

Expert Series

Progress Dynamics. Programming Handbook

John Sadd

PROGRESS
SOFTWARE

© 2005 Progress Software Corporation. All rights reserved.

Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. This manual is also copyrighted and all rights are reserved. This manual may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Progress Software Corporation.

The information in this manual is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

The references in this manual to specific platforms supported are subject to change.

A [Stylized], Allegrix, Allegrix & Design, Business Empowerment, eXcelon, ObjectStore, PeerDirect, Progress, Progress Dynamics, Powered by Progress, Empowerment Center, Progress Empowerment Center, Progress Empowerment Program, Progress Fast Track, Progress OpenEdge, Progress Profiles, Partners in Progress, Partners en Progress, Progress en Partners, Progress in Progress, P.I.P., Progress Results, Progress Software Developers Network, ProVision, ProCare, ProtoSpeed, SmartBeans, SpeedScript, Technical Empowerment, and WebSpeed are registered trademarks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and/or other countries. AccelEvent, A Data Center of Your Very Own, AppsAlive, AppServer, ASPen, ASP-in-a-Box, BusinessEdge, Cache-Forward, Fathom, Future Proof, IntelliStream, ObjectCache, ObjectStore Event Engine, ObjectStore RFID Accelerator, ObjectStore Trading Accelerator, OpenEdge, POSSE, POSSENET, ProDataSet, Progress Business Empowerment, Progress for Partners, PSE Pro, PS Select, SectorAlliance, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, WebClient, and Who Makes Progress are trademarks or service marks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and other countries.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Any other trademarks and service marks contained herein are the property of their respective owners.

February 2005



Product Code: 4494
Item Number: 103612;V2.1B

Acknowledgements

Much of the material in this book is derived from functional specifications for the various product areas it covers, and from white papers written by product developers. Many developers created that original material, without which this book would not have been possible. They include Anthony Swindells, Peter Judge, Mark Davies, Neil Bell, Chris Koster, Bruce Gruenbaum, Don Bulua, and Haavard Danielsen. Frank Beusenberg of Progress EMEA created the original code for the Browser popup example used in the chapter on Using the Dynamics Managers. Linda Gray provided invaluable editing assistance for the material. Ellen Marie Loycano, Linda Miller, and Ray Phillips assisted with the book's final production.

Contents

Preface	xv
Purpose	xv
Audience	xv
Organization of this manual	xv
Typographical conventions	xvii
Syntax notation	xviii
Progress messages	xxi
1. Writing Super Procedures for Progress Dynamics Objects	1-1
1.1 Client-side code	1-3
1.1.1 Client-side code in a distributed application	1-3
1.1.2 Using super procedures for client code	1-5
1.2 Using custom super procedures in Progress Dynamics	1-6
1.3 Defining a custom super procedure for an object	1-7
1.4 Writing code for a custom super procedure	1-10
1.4.1 Defining logic in a single custom super procedure	1-11
1.4.2 Creating a new super procedure	1-12
1.5 Defining user interface events in Progress Dynamics	1-13
1.5.1 Modifying Object Generator defaults for a dynamic object	1-14
1.5.2 Defining UI events for the viewer	1-16
1.5.3 Naming conventions for UI events and super procedures	1-20
1.5.4 Writing the supporting procedures for the UI event	1-21
1.5.5 Testing the dynamic viewer with the UI event	1-22
1.6 Moving client logic support to an extended class	1-22
1.7 Running a PLIP from client logic	1-35
1.7.1 Using launch.i or dynlaunch.i to run the PLIP	1-41
1.7.2 Testing the completed business logic procedure	1-43

1.8	Building a custom super procedure for a browser	1-45
1.8.1	Useful browser properties	1-46
1.8.2	Coding the rowDisplay procedure	1-47
1.8.3	Creating the other support functions	1-48
1.8.4	Defining the custom super procedure for a browser	1-49
1.8.5	Changing browser attributes for the master and instance.	1-49
1.9	Using the client logic API	1-50
1.9.1	Basic rules.	1-50
1.9.2	Logic	1-50
1.9.3	Object qualification	1-52
1.9.4	Client logic functions	1-56
1.10	Organizing logic at the level of procedures or functions	1-61
1.11	Customizing dynamic lookup behavior in viewers	1-62
1.11.1	LookupEntry event	1-62
1.11.2	LookupComplete event	1-63
1.11.3	LookupDisplayComplete event	1-65
2.	Extending Object Classes	2-1
2.1	Introduction	2-2
2.1.1	The Progress Dynamics object hierarchy	2-2
2.1.2	Understanding the nodes of the object hierarchy	2-3
2.2	Extending the behavior of standard objects	2-6
2.2.1	Extending the behavior of a single object	2-6
2.2.2	Extending the behavior of all objects in a class	2-7
2.2.3	Extending the behavior of some objects in a class	2-8
2.2.4	Extending object behavior in the class hierarchy	2-9
2.2.5	ADM customization	2-10
2.2.6	Instantiation order of super procedures	2-14
2.3	Defining new attributes and default values	2-15
2.3.1	ADM2 attribute support	2-15
2.3.2	Adding a new attribute to the repository	2-17
2.3.3	Providing distinctive names for attributes and other objects	2-18
2.4	Extending the class hierarchy	2-19
2.4.1	Understanding the rendering procedure	2-20
2.4.2	Thin rendering objects.	2-21
2.4.3	Defining support functions for new attributes	2-21
2.4.4	Adding the new class to the AppBuilder	2-21
2.4.5	Supporting your new attribute in standard SmartObjects	2-22

2.5	Changing the object type of a set of objects	2-23
2.5.1	Changing the object type	2-25
2.5.2	SCM integration considerations	2-25
2.5.3	Removing redundant attribute values	2-26
2.5.4	Removing invalid attribute values	2-26
2.5.5	Processing the changes.	2-27
2.5.6	Resetting the session cache to test the results	2-27
2.5.7	Modifying the classes cached at startup	2-28
2.6	Using and extending dynamic basic objects	2-29
2.6.1	Subclassing a button object	2-30
2.6.2	Adding a custom palette instance	2-33
2.7	Extending object classes tutorial	2-36
2.7.1	Adding a new attribute	2-37
2.7.2	Extending the class at the bottom of the hierarchy	2-39
2.7.3	Defining custom behavior for the subclass	2-41
2.7.4	Defining extended behavior for the new attribute	2-43
2.7.5	Defining support functions for new attributes	2-45
2.7.6	Adding the new class to the AppBuilder	2-46
2.7.7	Building an object of your extended class	2-51
2.7.8	Setting your extended attributes in an object	2-53
2.8	More complex customization	2-55
2.8.1	Customizing class behavior in the middle	2-55
2.8.2	Customizing a class with multiple classes	2-59
2.8.3	Migrating existing customizations	2-60
3.	Advanced User Interface Design In Progress Dynamics	3-1
3.1	Advanced support for object links	3-3
3.1.1	Using a single toolbar in a multi-page folder	3-3
3.1.2	Setting up your test window	3-3
3.1.3	Modifying the default link activation	3-13
3.1.4	Disabling data links to dependent SDOs	3-14
3.1.5	Using a GroupAssign link to group pages	3-15
3.2	Modifying visual properties at the master and instance level	3-17
3.2.1	Setting a property at the master level	3-17
3.2.2	Setting properties at the instance level	3-18
3.2.3	Using the Dynamic property sheet to set properties	3-19
3.2.4	Attribute value level summary	3-22
3.2.5	The attribute control and the object type control	3-23
3.3	Disabling and hiding buttons and menu items	3-28
3.3.1	Disabling actions based on a data value	3-28
3.3.2	Putting the value check into the window code	3-38

3.4	Defining action rules for menu and toolbar items	3-42
3.4.1	Understanding the role of the item link	3-46
3.4.2	Syntax of the action rules	3-47
3.4.3	Properties typically used in action rules	3-48
3.4.4	Functions typically used in action rules	3-50
3.4.5	Defining rules that use your own functions and properties	3-51
4.	Caching Application Data on the Client	4-1
4.1	Why cache?	4-2
4.2	Data types and cache types	4-2
4.3	Mechanics of caching	4-4
4.3.1	Cache versus a shared cache	4-5
4.3.2	Interaction of the CacheDuration and Shared properties	4-6
4.4	Enabling caching for SDOs	4-6
4.5	Turning on caching for dynamic lookups and combos in a session	4-7
4.6	Enabling caching for a dynamic lookups	4-8
4.7	Enabling caching for a dynamic combo	4-9
4.7.1	Enabling SDF-level caching	4-9
4.7.2	Enabling SDO-level caching	4-10
4.7.3	Enabling both	4-12
4.7.4	Synching instances	4-12
4.8	Using the Dynamic Launcher	4-14
4.9	Cache APIs	4-14
4.9.1	Programming notes	4-14
4.9.2	Cache key	4-15
4.9.3	registerCacheItem	4-15
4.9.4	findCacheItem	4-16
4.9.5	fetchDataFromCache	4-16
4.9.6	createSharedBuffer	4-16
4.9.7	destroySharedBuffer	4-17
4.9.8	clearCache	4-17
4.9.9	Modifying the instance properties	4-18
5.	Using ADM2 Properties and Methods In Progress Dynamics	5-1
5.1	Getting basic information	5-3
5.1.1	Object names and types	5-3
5.1.2	Testing whether objects and fields are enabled	5-4
5.1.3	Field and widget lists	5-5
5.1.4	Useful object handles	5-7
5.1.5	Functions that return field values and attributes	5-8
5.1.6	Other useful data object functions	5-11

5.2	Starting Progress Dynamics application windows	5-13
5.2.1	Initializing SmartObjects	5-19
5.2.2	Other methods related to startup and shutdown	5-24
5.2.3	Other initialization properties	5-25
5.3	Managing links in Progress Dynamics applications	5-27
5.3.1	SmartObject SupportedLinks and the addLink procedure	5-27
5.3.2	Defining custom SmartLinks and events	5-30
5.3.3	Link support methods	5-30
5.3.4	Miscellaneous link properties	5-32
5.4	Customizing and managing queries	5-33
5.4.1	Modifying the WHERE clause at runtime	5-33
5.4.2	Using addQueryWhere and assignQuerySelection	5-35
5.4.3	Repositioning the database query	5-39
5.4.4	Refreshing the database query	5-43
5.4.5	Using resortFiltering and sorting the RowObject query	5-45
5.4.6	Additional query methods	5-49
5.4.7	Other query properties	5-51
5.5	Paging methods and properties	5-51
5.5.1	Paging methods	5-51
5.5.2	Paging properties	5-53
5.6	Special functions that manage properties	5-54
5.7	General-purpose methods	5-56
6.	Using the Progress Dynamics Managers	6-1
6.1	Overview of the managers	6-3
6.1.1	Session Manager	6-5
6.1.2	Configuration File Manager	6-6
6.1.3	Connection Manager	6-9
6.1.4	Profile Manager	6-9
6.1.5	Localization Manager	6-11
6.1.6	Security Manager	6-11
6.1.7	Repository Manager and Repository Design Manager	6-12
6.1.8	General Manager	6-12
6.1.9	Request Manager	6-13
6.1.10	User Interface Manager	6-13
6.1.11	Referential Integrity Manager	6-13
6.1.12	Customization Manager	6-14
6.2	Manager architecture	6-14
6.2.1	Manager handles	6-14
6.2.2	Organization of the manager code	6-15
6.2.3	How the manager code runs on the server	6-21

6.3	Using the Session Manager	6-23
6.3.1	Context management	6-23
6.3.2	Property management	6-25
6.3.3	Managing procedures and containers	6-27
6.4	Using the Configuration File Manager	6-39
6.4.1	Using the isICFRunning function	6-39
6.4.2	Using the getManagerHandle function	6-40
6.4.3	Checking session types and session parameters	6-42
6.5	Using the Connection Manager	6-44
6.6	Using the Profile Manager	6-47
6.6.1	Creating user profile codes and types	6-48
6.6.2	Creating new browser properties	6-49
6.6.3	Creating a custom super procedure for the browser	6-54
6.6.4	Customizing the initializeObject procedure	6-56
6.6.5	Creating the Popup menu	6-60
6.6.6	Defining the Popup menu event procedures	6-62
6.7	Using the Localization Manager	6-72
6.7.1	Storing translated text strings in the Repository	6-73
6.7.2	Using the API calls translatePhrase and getTranslation	6-74
6.7.3	Using the multiTranslation call	6-76
6.8	Using the Security Manager	6-80
6.9	Using the General Manager	6-84
7.	Creating a New Manager In Progress Dynamics	7-1
7.1	The manager templates	7-3
7.2	Building a new manager in the AppBuilder	7-5
7.2.1	Separating client code from server code	7-5
7.3	Registering your manager	7-7
7.3.1	Creating a new manager type	7-8
7.3.2	Adding your manager to the session types	7-10
7.3.3	Invoking the session type in your startup command	7-15
7.4	Creating a new Field Edit manager	7-17
7.4.1	Adding a new table to the Repository	7-17
7.4.2	Defining the manager itself	7-20
7.4.3	Registering the new manager	7-31
7.4.4	Accessing the manager from SDOs	7-34
7.4.5	Accessing the manager from viewers	7-38
7.4.6	Invoking the field edits from an SDO	7-40
7.4.7	Testing the new manager	7-40
7.5	Customizing an existing manager	7-41

8.	Understanding the Object Tables In the Progress Dynamics Repository . . .	8-1
8.1	Architectural principles	8-4
8.1.1	Object IDs in the Repository	8-4
8.2	Object types, SmartObjects, and instances	8-5
8.2.1	Object diagram	8-6
8.2.2	The object type table	8-6
8.2.3	The SmartObject table	8-8
8.2.4	The object instance table	8-13
8.2.5	SmartObject table example	8-15
8.3	Attribute tables	8-17
8.3.1	The attribute group table	8-19
8.3.2	The attribute table	8-19
8.3.3	The attribute value table	8-23
8.3.4	Identifying the level of an attribute value	8-24
8.4	SmartLink tables	8-34
8.4.1	The SmartLink type table	8-35
8.4.2	The supported link table	8-35
8.4.3	The SmartLink table	8-36
8.5	Folder page tables	8-37
8.5.1	The layout table	8-38
8.5.2	The page table	8-39
8.5.3	The page object table	8-40
8.6	Customization tables	8-41
8.7	The customization type table	8-42
8.7.1	The customization result table	8-42
8.7.2	The customization table	8-43
8.7.3	Joining customizations to the SmartObject table	8-44
8.7.4	Tools to support customization maintenance	8-47
8.8	Using the Repository Manager	8-48
8.8.1	The client cache	8-48
8.8.2	Repository Manager API	8-53
8.8.3	Retrieving related information for an object	8-57
8.8.4	How objects are instantiated using prepareInstance	8-59
Index	Index	Index-1

Figures

Figure 1-1:	Property sheet	1-8
Figure 1-2:	Filtering objects for a new super procedure	1-12
Figure 1-3:	Customers with a dynamic viewer	1-22
Figure 1-4:	Relationships between objects and super procedures	1-26
Figure 1-5:	ProTools palette	1-27
Figure 1-6:	Prototype Generator window	1-27
Figure 1-7:	Improper function reference without TARGET-PROCEDURE	1-32
Figure 1-8:	Proper function reference with TARGET-PROCEDURE	1-33
Figure 2-1:	Class hierarchy	2-4
Figure 2-2:	Defining a super procedure	2-11
Figure 2-3:	Subclassing an object type	2-12
Figure 2-4:	Super procedure for individual objects	2-13
Figure 3-1:	Smart Toolbar Properties window	3-13
Figure 3-2:	Dynamic Properties window	3-14
Figure 3-3:	GroupAssign link example	3-16
Figure 3-4:	Property sheet for the CustNum field	3-18
Figure 3-5:	Dynamic properties sheet	3-19
Figure 3-6:	Viewer property sheet	3-20
Figure 3-7:	Attribute Maintenance window	3-24
Figure 3-8:	Object Type Control window	3-27
Figure 3-9:	Dynamic Properties window	3-29
Figure 3-10:	Container Builder for oemaintwin	3-31
Figure 3-11:	Toolbar and Menu designer	3-32
Figure 3-12:	Property sheet for frMain	3-36
Figure 3-13:	Test of window wltH DisabledActions	3-37
Figure 3-14:	Toolbar and Menu Designer	3-44
Figure 3-15:	Toolbar and Menu Designer: item link definition	3-46
Figure 3-16:	Defining an item link	3-46
Figure 5-1:	Customer record message box	5-6
Figure 5-2:	Customer selection window	5-17
Figure 5-3:	Property Sheet Master	5-26
Figure 5-4:	Test window for repositioning	5-40
Figure 5-5:	Customer maintenance window	5-50
Figure 6-1:	Progress Dynamics manager architecture	6-4
Figure 6-2:	Executable code in main block	6-17
Figure 6-3:	Data access file	6-18
Figure 6-4:	Context database table definitions	6-24
Figure 6-5:	Sample application wltH	6-31
Figure 6-6:	Static Handle field of the Manager Type Control wltH	6-40
Figure 6-7:	Manager Type Maintenance - Connection Manager	6-41
Figure 6-8:	Output of testRecordDetail.p	6-85
Figure 6-9:	FOR EACH query results	6-86

Figure 7-1:	Section Editor	7-5
Figure 7-2:	Standard Repository tables	7-18
Figure 7-3:	Field Properties window	7-19
Figure 7-4:	Table Triggers dialog box	7-19
Figure 7-5:	Section Editor	7-23
Figure 7-6:	Message Control window	7-37
Figure 7-7:	Field Edit Manager session type	7-41
Figure 7-8:	Error message	7-41
Figure 8-1:	Object diagram	8-6
Figure 8-2:	Order entry maintenance window records	8-15
Figure 8-3:	Attribute table relationships	8-18
Figure 8-4:	Attribute value and table relationships	8-25
Figure 8-5:	NavigationSourceEvents attribute	8-26
Figure 8-6:	Attributes defined at the object master level	8-27
Figure 8-7:	Relationships of attribute value records to other records	8-27
Figure 8-8:	Attributes defined at the object instance level	8-28
Figure 8-9:	Instance attribute value relationships	8-29
Figure 8-10:	Object instances in the container builder window	8-29
Figure 8-11:	Class level attribute value	8-31
Figure 8-12:	Objects with assigned minimum height values	8-32
Figure 8-13:	Classes with defined initial values for MinHeight	8-33
Figure 8-14:	DisableOnInit properties for oemaintwin	8-34
Figure 8-15:	Link, SmartObject, and object instance tables	8-37
Figure 8-16:	Folder page table diagram	8-40
Figure 8-17:	Customization tables	8-41
Figure 8-18:	Customization example	8-46
Figure 8-19:	Customization maintenance utility	8-47
Figure 8-20:	Levels of customization	8-48

Tables

Table 1–1:	Client logic functions	1–56
Table 1–2:	LookupCompleteEvent parameters	1–64
Table 1–3:	LookupDisplayCompleteEvent parameters	1–66
Table 4–1:	Caching feature combinations	4–12
Table 4–2:	Instance properties and defaults of an SDO data source	4–18
Table 5–1:	Default values of resize properties for visual objects	5–17
Table 7–1:	The gsc_entity_field_edit Table	7–18
Table 8–1:	The gsc_object_type table	8–7

Preface

Purpose

This handbook provides information about various programming topics in Progress Dynamics®. Use it along with the *Progress Dynamics Developer's Guide* as a guide and reference to programming with Progress Dynamics.

Audience

This guide is designed for any developer familiar with the Progress® 4GL who is interested in building a new Progress application, or rearchitecting an existing application to bring it to a distributed GUI environment.

Organization of this manual

This book is organized in the following manner:

[Chapter 1, “Writing Super Procedures for Progress Dynamics Objects”](#)

Describes how to create custom super procedures for dynamic client-side objects, including the kind of code and programming style to use, and introduces an API to simplify the code you write. There is also discussion of dynamic User Interface events, and of web development considerations.

[Chapter 2, “Extending Object Classes”](#)

Describes how to extend the Progress Dynamics class hierarchy by adding your own custom classes.

Chapter 3, “Advanced User Interface Design In Progress Dynamics”

Provides in depth coverage of important programming techniques that enrich the user interface.

Chapter 4, “Caching Application Data on the Client”

Describes how to take advantage of various framework features to cache application data.

Chapter 5, “Using ADM2 Properties and Methods In Progress Dynamics”

Provides an overview of the properties and methods of the Application Development Model that are essential to Progress Dynamics application builders. It also provides guidance on how properties and methods are typically used.

Chapter 6, “Using the Progress Dynamics Managers”

Describes the role of Progress Dynamics managers, which are a set of service procedures that support a wide range of application needs. There is a brief overview of the Managers, a description of how they are constructed, and some guidelines on how to use specific Manager API calls in your applications. In addition, there are some examples designed to give you a better understanding of how to use the Managers to provide support for special needs that they do not take care of automatically.

Chapter 7, “Creating a New Manager In Progress Dynamics”

Describes the new template procedures and how you can use them to create Managers of your own for your applications.

Chapter 8, “Understanding the Object Tables In the Progress Dynamics Repository”

Describes the Progress Dynamics repository tables and their fields in detail in order to provide you with a complete understanding of the repository database.

NOTE: There is a set of example files in a file called `DPH_Examples.zip`, that accompanies this manual. This file is available with the electronic version of the documentation. You can find this file on the Progress Dynamics documentation PDF CD and at the following URL:

<http://www.progress.com/products/documentation/index.ssp>.

Typographical conventions

This manual uses the following typographical conventions:

- **Bold typeface** indicates:
 - Commands or characters that the user types
 - That a word carries particular weight or emphasis
 - Names of user interface elements
- *Italic typeface* indicates:
 - Progress variable information that the user supplies
 - New terms
 - Titles of complete publications
- Monospaced typeface indicates:
 - Code examples
 - System output
 - Operating system filenames and pathnames

The following typographical conventions are used to represent keystrokes:

- Small capitals are used for Progress key functions and generic keyboard keys.
END-ERROR, GET, GO
ALT, CTRL, SPACEBAR, TAB
- When you have to press a combination of keys, they are joined by a hyphen. You press and hold down the first key, then press the second key.
CTRL-X
- When you have to press and release one key, then press another key, the key names are separated with a space.
ESCAPE H
ESCAPE CURSOR-LEFT

Syntax notation

The syntax for each component follows a set of conventions:

- Uppercase words are keywords. Although they are always shown in uppercase, you can use either uppercase or lowercase when using them in a procedure.

In this example, ACCUM is a keyword:

Syntax

`ACCUM aggregate expression`

- Italics identify options or arguments that you must supply. These options can be defined as part of the syntax or in a separate syntax identified by the name in italics. In the ACCUM function above, the *aggregate* and *expression* options are defined with the syntax for the ACCUM function in the [Progress Language Reference](#).
- You must end all statements (except for DO, FOR, FUNCTION, PROCEDURE, and REPEAT) with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon, as in this example:

`FOR EACH Customer:
 DISPLAY Name.
END.`

- Square brackets (`[]`) around an item indicate that the item, or a choice of one of the enclosed items, is optional.

In this example, STREAM *stream*, UNLESS-HIDDEN, and NO-ERROR are optional:

Syntax

`DISPLAY [STREAM stream] [UNLESS-HIDDEN] [NO-ERROR]`

In some instances, square brackets are not a syntax notation, but part of the language.

For example, this syntax for the INITIAL option uses brackets to bound an initial value list for an array variable definition. In these cases, normal text brackets ([]) are used:

Syntax

```
INITIAL [ constant [ , constant ] . . . ]
```

NOTE: The ellipsis (. . .) indicates repetition, as shown in a following description.

- Braces ({ }) around an item indicate that the item, or a choice of one of the enclosed items, is required.

In this example, you must specify the items BY and *expression* and can optionally specify the item DESCENDING, in that order:

Syntax

```
{ BY expression [ DESCENDING ] }
```

In some cases, braces are not a syntax notation, but part of the language.

For example, a called external procedure must use braces when referencing arguments passed by a calling procedure. In these cases, normal text braces ({ }) are used:

Syntax

```
{ &argument-name }
```

- A vertical bar (|) indicates a choice.

In this example, EACH, FIRST, and LAST are optional, but you can only choose one:

Syntax

```
PRESELECT [ EACH | FIRST | LAST ] record-phrase
```

In this example, you must select one of *logical-name* or *alias*:

Syntax

```
CONNECTED ( { logical-name | alias } )
```

- Ellipses (. . .) indicate that you can choose one or more of the preceding items. If a group of items is enclosed in braces and followed by ellipses, you must choose one or more of those items. If a group of items is enclosed in brackets and followed by ellipses, you can optionally choose one or more of those items.

In this example, you must include two expressions, but you can optionally include more. Note that each subsequent expression must be preceded by a comma:

Syntax

```
MAXIMUM ( expression , expression [ , expression ] . . . )
```

In this example, you must specify MESSAGE, then at least one of *expression* or SKIP, but any additional number of *expression* or SKIP is allowed:

Syntax

```
MESSAGE { expression | SKIP [ (n) ] } . . .
```

In this example, you must specify {*include-file*, then optionally any number of *argument* or *&argument-name* = "*argument-value*", and then terminate with }:

Syntax

```
{ include-file  
  [ argument | &argument-name = "argument-value" ] . . . }
```

- In some examples, the syntax is too long to place in one horizontal row. In such cases, **optional** items appear individually bracketed in multiple rows in order, left-to-right and top-to-bottom. This order generally applies, unless otherwise specified. **Required** items also appear on multiple rows in the required order, left-to-right and top-to-bottom. In cases where grouping and order might otherwise be ambiguous, braced (required) or bracketed (optional) groups clarify the groupings.

In this example, WITH is followed by several optional items:

Syntax

```
WITH [ ACCUM max-length ] [ expression DOWN ]
    [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
    [ STREAM-IO ]
```

In this example, ASSIGN requires one of two choices: either one or more of *field*, or one of *record*. Other options available with either *field* or *record* are grouped with braces and brackets. The open and close braces indicate the required order of options:

Syntax

```
ASSIGN { { [ FRAME frame ]
          { field [ = expression ] }
          [ WHEN expression ]
        } ...
      | { record [ EXCEPT field ... ] }
    }
```

Progress messages

Progress displays several types of messages to inform you of routine and unusual occurrences:

- Execution messages inform you of errors encountered while Progress is running a procedure (for example, if Progress cannot find a record with a specified index field value).
- Compile messages inform you of errors found while Progress is reading and analyzing a procedure prior to running it (for example, if a procedure references a table name that is not defined in the database).
- Startup messages inform you of unusual conditions detected while Progress is getting ready to execute (for example, if you entered an invalid startup parameter).

After displaying a message, Progress proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify, or that are assumed, as part of the procedure. This is the most common action taken following execution messages.
- Returns to the Progress Procedure Editor so that you can correct an error in a procedure. This is the usual action taken following compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

Progress messages end with a message number in parentheses. In this example, the message number is 200:

```
** Unknown table name table. (200)
```

Use Progress online help to get more information about Progress messages. Many Progress tools include the following Help menu options to provide information about messages:

- Choose **Help**→**Recent Messages** to display detailed descriptions of the most recent Progress message and all other messages returned in the current session.
- Choose **Help**→**Messages**, then enter the message number to display a description of any Progress message. (If you encounter an error that terminates Progress, make a note of the message number before restarting.)
- In the Procedure Editor, press the **HELP** key (**F2** or **CTRL-W**).

Writing Super Procedures for Progress Dynamics Objects

A basic principle of the Progress Dynamics® framework is to create an application from mostly dynamic, or data-driven components. In the *Progress Dynamics Developer's Guide*, you can see how to generate dynamic Browsers and Viewers, and how to assemble these into dynamic Windows with Folders and dynamic Toolbars.

A major goal of providing all these dynamic objects is to minimize the amount of code in your application that you need to write, maintain, and deploy. This leaves open the question of where you **do** put code when you have to write it. And of course you do have to write code of many kinds to complete any application. The Progress Dynamics framework provides most of the default behavior you need, but not application-specific business logic.

The discussion of the SDO logic procedure in the *Progress Dynamics Developer's Guide* answers the question for the SDO: in order to free most SDOs from static code and to be dynamic objects, you should write business logic and validation logic in a separate procedure that is run along with your dynamic SDO instance, and made a super procedure of the SDO.

The same technique applies to other dynamic objects as well. If you have dynamic Viewers and Browsers in your client user interface, and they need to respond to events with custom code, you can put that code into a separate procedure, which you designate as the *custom super procedure* for the object. At run time, the super procedure is run along with the instance of the dynamic object, and it responds to events in the object by running custom code.

This chapter describes how to create custom super procedures for dynamic client-side objects and what kind of code you should consider putting into them (and **not** putting into them), and suggests a programming style and introduces an API that together simplify the code you write. It also discusses dynamic user interface events, as well as how to anticipate providing your application with a Web browser user interface in Progress Dynamics Version 2. This chapter includes the following topics:

- [Client-side code](#)
- [Using custom super procedures in Progress Dynamics](#)
- [Defining a custom super procedure for an object](#)
- [Writing code for a custom super procedure](#)
- [Defining user interface events in Progress Dynamics](#)
- [Moving client logic support to an extended class](#)
- [Running a PLIP from client logic](#)
- [Building a custom super procedure for a browser](#)
- [Using the client logic API](#)
- [Customizing dynamic lookup behavior in viewers](#)

1.1 Client-side code

This section provides some examples and programming guidelines for you, including:

- [Client-side code in a distributed application.](#)
- [Using super procedures for client code.](#)

1.1.1 Client-side code in a distributed application

Progress Dynamics is designed in every way to support distributed applications, where the user interface runs in a client session separate from an AppServer™ session that is connected to the database. This means that direct database references must be avoided completely in any Progress code to be executed on the client. It means that client-side events that force an AppServer call to retrieve information should be kept to a minimum for the best performance. It also means that, in general, client-side executable code should be kept to a minimum to reduce the number of procedures that have to be deployed to client machines to run the application.

So if client-side code cannot reference the database, and should not call back to the AppServer except when necessary, what remains for the client code to do?

There are still likely to be many instances in which something particular to your application has to happen on the client, for example, an event that is triggered on the client and must be responded to on the client. On CHOOSE of a button, your code might need to check the value of the field and enable or disable other fields on the screen, or enable or disable menu items or buttons for certain related actions. You might need to modify the interface details of the screen in other ways, modifying colors depending on field values, displaying related text, etc. These are all legitimate actions that rightly belong in the client-side code.

In some cases it is also necessary to make a call back to the server to obtain information or cause a server-side database action. As noted, these calls should be kept to a minimum to keep application performance from suffering in a widely distributed application, where every AppServer hit is costly. Generally the amount of data passed between client and server is less of a performance issue than the number of AppServer requests, and this must be kept in mind when you are developing your code. If a data value that the user enters into a field requires that related data be retrieved from the server, and if this cannot be done until the field value is known, then go ahead and do it. If there are toolbar buttons or menu items, or other buttons in your application screens that retrieve data from the AppServer, then you can provide trigger code to handle this. Just ask yourself in each case whether the call is really necessary, and whether it can be combined with other calls.

This might mean a major adjustment in your thinking, especially for developers who have created host-based or client-server applications in earlier versions of Progress, where the user interface logic is usually compiled right into the user interface itself, and where the database is always available for immediate access. In this environment, it is simple and appropriate to write field validation trigger code in the Data Dictionary that is compiled into the procedure along with the frame that contains the field. This code is executed on LEAVE of every field in the frame that has validation logic. It makes a reference to the database, often to do a CAN-FIND to make sure that a value typed into a foreign key field (such as `Order.CustNum` in the Sports2000 database) is a valid value from another table (such as `Customer.CustNum`).

In a distributed environment, making a call to the database means making a call to a separate server session, and doing this on LEAVE of many fields on a screen would incur a terrible performance cost. This does not mean that users are left without the feedback and validation they are used to from older applications. It simply means that your application design has to take a different approach to providing this kind of feedback. The dynamic Lookups and Combos in Progress Dynamics are one mechanism for providing and improving on CAN-FIND support without the immediate database lookup. A dynamic Combo provides a list of all possible values for a field, assuring that the user will choose one correctly, and the Lookup does the same for larger data sets, going back to the server only if the data set is too large to retrieve in a single batch and is not already cached on the client.

In other cases, calculations that might have been done on the fly in the user interface code in an earlier application can be done in advance and sent as calculated fields in the SDO in a single call, along with the rest of the data.

Another consideration is that code that requires database access on the client, apart from the integrity checks that Combos and Lookups do, is likely really business logic of one kind or another. Putting code of this type on the client violates another principle of distributed application: that business logic belongs on the server, either in SDOs (and their logic procedures), SBOs, or other procedures that can be accessed from anywhere on the client. Embedding business logic in the UI means that it will not be automatically executed from some alternative UI, whether it is another screen in the same application or an alternative UI altogether, such as for a Web browser interface. It also adds to the maintenance headaches you face when you have to modify your application's behavior.

In any case, the basic goals remain the same:

- Minimize special code written for the client.
- Minimize unnecessary AppServer calls.
- Keep business logic out of the client code altogether whenever possible.

1.1.2 Using super procedures for client code

The basic reason for using super procedures in a dynamic application is obvious: if your application object is fully data-driven, there is nowhere else to put it! There are a number of advantages to putting your custom code into a small, specialized procedure, while leaving all the standard behavior to dynamic Objects.

First, the compiled static SDO is a large procedure, with a substantial amount of code providing a temp-table definition, query, and a number of update operations compiled specifically for that query and table. The internal procedures that provide custom validation logic are probably only a small percentage of the r-code generated for the SDO. You can greatly reduce the r-code that is deployed with your application by taking advantage of the dynamic SDO.

In addition, with your logic separate from the procedure that does the specific job of moving data back and forth between client and server, it can be executed more easily from elsewhere in your application, even when no SDO at all is involved. This is part of the reason why the SDO programming convention was modified to provide new validation hooks that can access an updated record by a more generally useful name of `b_` plus the primary table name, rather than using the SDO-specific naming convention of calling it `RowObjUpd`.

The SDO's logic procedure is just a specific example of a custom super procedure. It runs with some special wrapper code to make the updated record's buffer available to the validation procedures, but otherwise it runs as a super procedure of the SDO and can contain any other code that is useful.

The same principle of writing super procedures also applies to other kinds of objects, including dynamic SmartDataViewers™. First, the r-code footprint of the deployed application is greatly reduced. Secondly, because the Viewers and their fields are represented in the Repository, this abstract definition of the Viewer is available to the UI Manager as a basis for a Web browser interface or other alternative interface. Having all the data in the Repository is beneficial in other ways as well:

- To support various tools that need to know where fields are used.
- To help you understand what objects contain what other objects.
- To provide standard event behavior for a field no matter where it is used.

For these reasons, it is advisable to develop your applications with any required client-side logic in custom super procedures. In addition, other client-side objects like Browsers, as well as Windows, Folders, and Toolbars, are already dynamic, and the same techniques can apply to writing code to support these objects as well.

1.2 Using custom super procedures in Progress Dynamics

This section, which explains how to create super procedures in Progress Dynamics and attach them to your objects, starts with a review of just what super procedures are and how they work.

Super procedures provide a measure of inheritance and class behavior within the Progress 4GL. Code in a super procedure becomes an extension of the code written in another procedure, so to the extent that a single super procedure can be associated with many other procedures, it can give all of those other procedures the same standard behavior. The super procedure needs to be run only once within a session, where it can support any number of other procedures.

On the face of it, a super procedure looks just like any other Progress procedure. It contains no special code and no special declarations. It is run as a persistent procedure within a session, and can be started by the first procedure that needs it, or by any other mechanism. What makes it a super procedure is a method in the other procedure that wants to inherit its code, the `ADD-SUPER-PROCEDURE` method on the procedure handle. In this example, a procedure called `baseproc.p` runs the super procedure `superproc.p` and then makes the association:

```
DEFINE VARIABLE hSuper AS HANDLE NO-UNDO.  
  
RUN superproc.p PERSISTENT SET hSuper.  
THIS-PROCEDURE:ADD-SUPER-PROC(hSuper).
```

At this point, the Progress interpreter associates `superproc.p` and all its internal procedures and functions with `baseproc.p`. Any `RUN` statement in `baseproc.p` causes the interpreter to search first in `baseproc.p` and then in `superproc.p` for the entry point. It executes the first one it finds. In this way all the contents of the super procedure transparently act as if they were actually part of the base procedure.

In addition, because `baseproc.p` and `superproc.p` are in fact separately compiled procedures, each has its own namespace for procedure and function names, as well as for variables and other language constructs. This means that the same internal procedure or function name can appear in both procedures. The super procedure can provide standard behavior in the form of an internal procedure or function, and then the base procedure can override that behavior or augment it. To do this, the procedure or function in the base procedure must use the syntax `RUN SUPER` (for an internal procedure), or `SUPER()` (for a function) to invoke the standard behavior.

The interpreter locates and executes the occurrence of the procedure or function in the base procedure first, and then counts on the base procedure to invoke the same entry point in the super procedure if it wants to. Any other code can come before or after the `RUN SUPER` statement.

A base procedure can add any number of super procedures, just by invoking the `ADD-SUPER-PROC` method multiple times. The procedures are kept in a kind of Last In First Out (LIFO) stack, so the last one added is the first one searched for an entry point. When a super procedure is added to a stack, if that procedure also has super procedures, they too are added to the stack. Each one can pass control up the stack with its own `RUN SUPER` statement.

Adding a procedure to another object as a super procedure using the `ADD-SUPER-PROCEDURE()` method adds the contents of that procedure's super procedure stack to the object. This allows you, in effect, to have a super procedure of super procedures.

1.3 Defining a custom super procedure for an object

The super procedure mechanism has been used to provide SmartObjects™ with their standard behavior for Version 9, and this has been extended with Progress Dynamics. One extension is to provide a field in each SmartObject definition where the developer can define a custom super procedure for it. These are called custom super procedures precisely because the intention is to provide an individual application object with customized behavior not inherited from its standard super procedures and from other code in the framework. So as opposed to the model summarized in the previous section, where a single super procedure can provide common code to support many base procedures, the typical use of a custom super procedure in a Progress Dynamics application is likely to be to provide specific code needed by a single client-side Viewer, Browser, or other object.

NOTE: As on Version 2.1A, the `SuperProcedure` property on a class definition defines the common super procedure for all members of the class. On the master and instance level, this same property defines the custom super procedure for the master or instance.

To be sure, this is not always the case. A custom super procedure can provide common support for any collection of other objects, for example if they have fields in common or special field types and event handling needs in common. But if you find yourself attaching the same custom super procedure to a large number of application Objects, you can add that behavior to all the Objects as a class customization rather than to the individual objects.

If you want to add behavior to all objects of an existing type in your application, such as all Viewers or all Browsers, then you can define a super procedure in the `adm2/custom` directory to extend the class's behavior by adding another super procedure to every object of the type. On the other hand, if some group of objects needs the behavior but others do not, then you can use the Object Type Control to define a subclass, so you have two distinct types of object: one with the behavior and one without. See [Chapter 2, “Extending Object Classes,”](#) for more information about different techniques for extending the object hierarchy.

Some Progress Dynamics objects allow you to define a custom super procedure in the property sheet or other tool for creating objects of that type. For example, after you have created a set of Browsers using the Object Generator, you can open each one that requires custom code by using the Open Object button in the AppBuilder. Or you can define a Browser from scratch using the **New** button in the AppBuilder. Either way, you enter the AppBuilder's property sheet for the Browser, where you can name a custom super procedure, as you can see in [Figure 1–1](#).

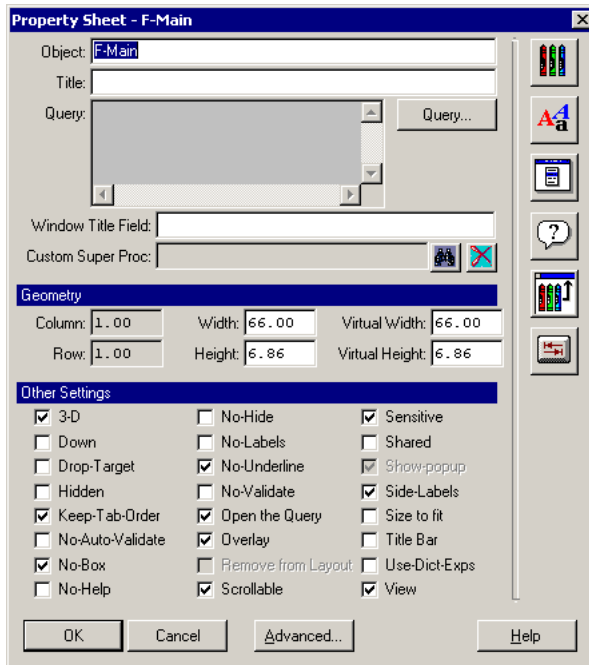


Figure 1–1: Property sheet

Custom super procedures can also be specified on the Save dialog for objects in AppBuilder.

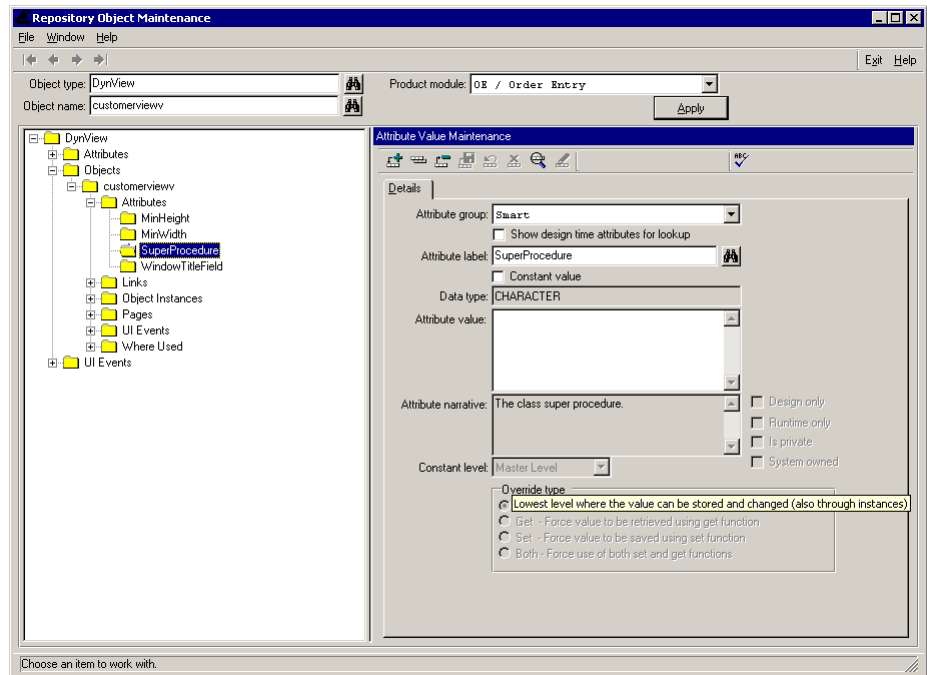
Remember that because the super procedure must be registered in the Progress Dynamics Repository like any other Object, you do not need to specify the relative pathname of the procedure. This is stored as part of the definition of the procedure as a Repository Object.

The AppBuilder allows you to set the custom super procedure for other Objects you edit there, such as SmartDataViewers. Likewise, the Container Builder lets you define the custom super procedure for a dynamic window.

To use the Repository Maintenance tool to define a super procedure:

- 1 ♦ Run the Repository Maintenance tool from the Appbuilder Build menu.
- 2 ♦ Enter the Object Name of the object and click **Apply**.

- 3 ♦ Expand both the top-level node in the TreeView, the Objects node, the Attributes node under that, and click SuperProcedure.
- 4 ♦ This brings you to the Attribute Value Maintenance window, where you can enter the name of the object's custom super procedure in the Attribute Value field:



Note that you can define custom super procedures in this way for both static and dynamic objects. If you want to associate extended behavior with multiple Objects of a type (as an alternative to truly subclassing the type), you can simply add the super procedure to all the Objects it should affect. Remember that if you want to add new attributes for the super procedure code to use, you must first add the attributes to the Repository (in the Attribute Maintenance), and then add them to the Object Type you are extending (in the Object Type Control or the Repository Maintenance Tool).

And remember that a dynamic object is simply an instance of a single procedure, such as `ry/obj/rydynview.w` for dynamic Viewers, that can read the right records out of the Repository to instantiate the object at run time. This one physical 4GL procedure serves all dynamic objects of the type. Thus there is always a procedure handle to associate with a super procedure, whether the object is dynamic or static. In the static case there is a separate source and compiled procedure for each object; in the dynamic case there is only one for all objects of a type.

1.4 Writing code for a custom super procedure

The nature of the super procedure also creates a programming challenge. Because the base procedure object and the super procedure are separately compiled objects, with their own name spaces, you cannot simply refer to frames un the base object. References to fields, variables, buffers, and other constructs are not automatically available to the super procedure, so the code that references them needs to change, and in general becomes somewhat more complex.

The Progress Version 9 ADM solves this problem for the SmartObjects themselves by defining a set of properties for each SmartObject type, whose values are stored in a temp-table record within the scope of the object. The {get} and {set} include files and the `get<property>` and `set<property>` functions provide access to those property values from other procedures, including in particular the object's super procedures.

In your application code, you need to do something similar to get access to the objects or *widgets* inside a Viewer or Browser.

NOTE: In the context of a Progress application, the term widget has a very specific meaning, namely, any basic 4GL object with its own object handle. Thus not only are fill-in fields and other visualizations of data fields widgets, but so are other types of controls such as buttons, as well as objects like rectangles that would never be thought of as controls.

You cannot simply refer to field names or button names from a super procedure attached to a Viewer or Browser. This section introduces a simple way to encapsulate that access in such a way that it can be used from any visual object's super procedure, and so acts as an API for writing super procedure code. The complete API, which will be explained later in this chapter, will be a standard part of Version 2 of Progress Dynamics, to help standardize and simplify writing client-side code.

To illustrate the different ways that you can write super procedure logic, parts of the API are built up in steps:

- [Defining logic in a single custom super procedure.](#)
- [Creating a new super procedure.](#)

1.4.1 Defining logic in a single custom super procedure

The first example is for a window managing the Customer table using the customerfullo SDO. The examples use objects from other parts of the documentation. If you need some of the base objects, you can simply build them. If your objects have different names, just adjust accordingly.

Suppose you want to define a user interface condition so that if the **Customer Credit Limit** does not exceed the **Balance** by at least \$5000, the **Balance** field is highlighted to show this. This check must be made whenever a row of data is displayed in the Viewer, and also when the **Credit Limit** or **Balance** field is modified.

In a static Viewer, you could simply write a simple statement into a local override of the displayFields procedure, which is executed each time a row is displayed:

```
PROCEDURE displayFields:
/*-----
Purpose:      Super Override of displayFields
Parameters:   pcColValues AS CHARACTER
Notes:        Highlights the Balance if it's too close to CreditLimit.
-----*/

DEFINE INPUT PARAMETER pcColValues AS CHARACTER NO-UNDO.

RUN SUPER( INPUT pcColValues).

DO WITH FRAME {&FRAME-NAME}:
  IF DECIMAL(RowObject.CreditLimit:SCREEN-VALUE) -
    DECIMAL(RowObject.Balance:SCREEN-VALUE) < 5000.00 THEN
    RowObject.Balance:BGCOLOR = 14. /* Highlight the field in yellow. */
  END.
END PROCEDURE.
```

Note that there are some complications to writing code like this, because the Viewer does not actually have access to the database record, or even the RowObject temp-table record. Because it is strictly a thin client object, the field values are simply copied into the screen values of the fields on display, and copied from there back to the associated SDO on Save. So your code needs to use the SCREEN-VALUE attribute of each field to get at its value, and then convert it to the proper data type. Also note how the code scopes the field references to the default Viewer frame name, which is represented by the preprocessor &FRAME-NAME.

If you want this same code to be executed on LEAVE of the Balance and CreditLimit fields, you can define a UI trigger for those events in the AppBuilder and associate the same code with them. In that case, it makes sense to move the code itself into a separate procedure to call from both places.

This kind of code is not available to a dynamic Viewer because there is no place to write the code. So you have to move the code to a custom super procedure you associate with the Viewer instance. In this case, the fields such as **RowObject.CreditLimit** are not directly available to the compiler when it compiles your super procedure, so you cannot reference them. Instead, your code must access the fields with the client API, which again is an instance (a running copy) of the single procedure `rydynview.w`, which reads data for your specific Viewer out of the Progress Dynamics Repository and creates the Viewer for you at run time.

1.4.2 Creating a new super procedure

To create a new super procedure, press the **New** button in the AppBuilder and select **Custom Super Procedure** from the list of Procedure objects. [Figure 1–2](#) shows how you can filter the kinds of objects the AppBuilder can create for you, in this case just showing Procedure files.

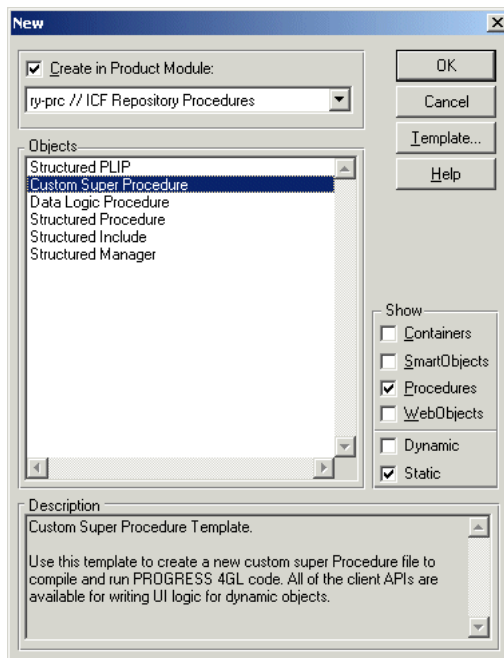


Figure 1–2: Filtering objects for a new super procedure

The brief procedure wizard simply prompts you for documentation information that it writes into the top of the procedure file. After you write your code and save the procedure, you must associate it with the Viewer in the AppBuilder's property sheet for the Viewer.

1.5 Defining user interface events in Progress Dynamics

The example you have created attaches an action to an ADM event, the `displayFields` event. This defines when it is executed relative to the overall sequence of SmartObject events. In addition, you will often want to define actions that occur when a user interface event occurs, such as choosing a button or changing a value. Progress Dynamics supports the definition of these UI Events, so that you can associate code with events in dynamic objects.

You can define UI events for objects such as a dynamic Viewer using the dynamic Property Sheet in the AppBuilder. For more information, see the [Progress Dynamics Developer's Guide](#).

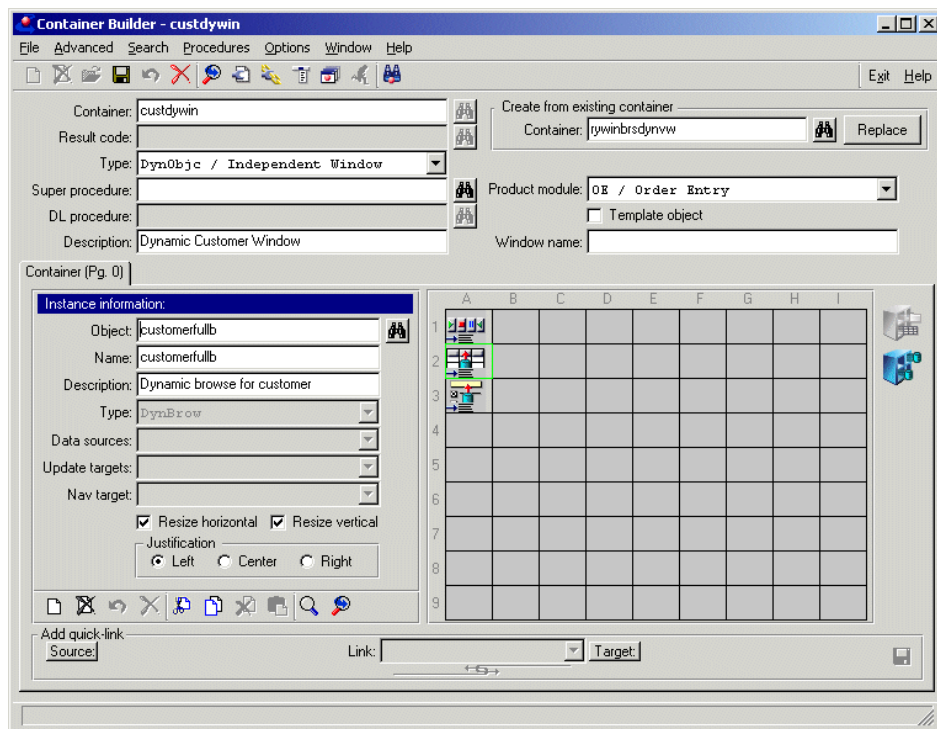
Note that Repository data is not created for the fields in a static Viewer, although the Viewer itself and its attributes are registered in the Repository, allowing you, for example, to define a custom super procedure for the static Viewer. When you create a dynamic Viewer using the Object Generator or the AppBuilder, not only is the Viewer itself registered in the Repository, but there is a whole set of object and attribute values created for each of the fields and other objects in the Viewer. It is this data that the dynamic Viewer procedure uses to instantiate the Viewer at run time. If you want to define UI events for field events such as `VALUE-CHANGED`, you can do this only for dynamic Viewers, not static ones.

Next you must construct a window with a dynamic Viewer in it to use to continue the client logic example. There is a layout template already defined in the Repository that you can use as a basis for a simple test window. It has the rather dense but all-inclusive name `rywinbrsdynvw`. This window has slots for an SDO, a dynamic Browser, and a dynamic Viewer, in addition to the Standard Toolbar.

To create a Customer test window:

- 1 ♦ Select **New→Independent Window** in the AppBuilder.
- 2 ♦ Give the window an Object Name such as **custdynwin**.
- 3 ♦ Specify **rywinbrsdynvw** as the Container Template.

- 4 ♦ Replace the template SDO, Browser, and Viewer with the ones the Object Generator created for the Customer table:



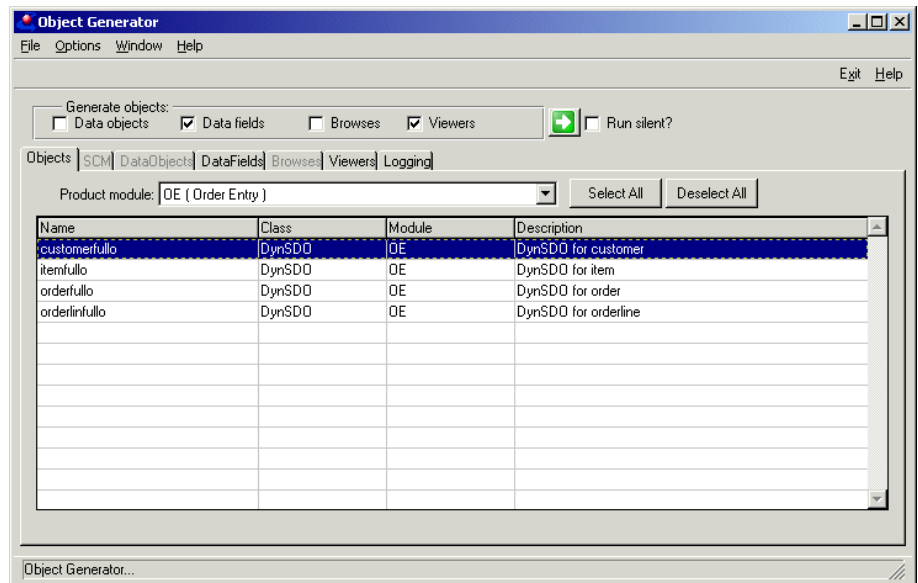
- 5 ♦ Save the new container.

1.5.1 Modifying Object Generator defaults for a dynamic object

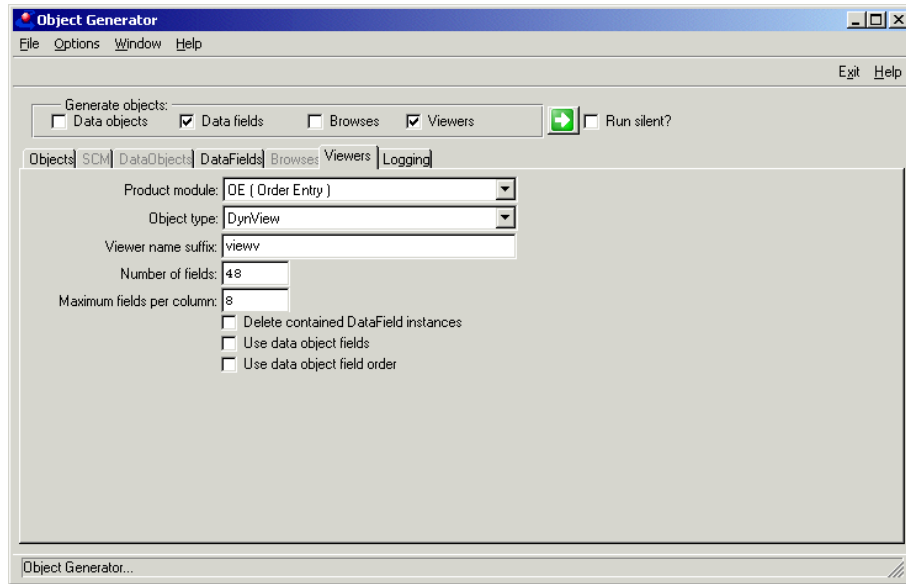
A slight digression is needed to get a reasonable looking dynamic Viewer. If you run the dynamic window you just created, you might see that all the Customer fields in the Viewer are in a single column, making for a very tall and awkward-looking Viewer. This is a result of the Viewer Settings you chose when you ran the Object Generator. You can edit the Viewer in the AppBuilder to arrange the fields as you want them, but it is still helpful to improve the default layout so that there is less to change later on.

To reduce the number of fields in a column:

- 1 ♦ Return to the Object Generator by choosing **Build→Object Generator** in the AppBuilder menu.
- 2 ♦ Set the **OE Product Module**.
- 3 ♦ Check the **Data Objects** toggle box off. Existing SDOs in your selected Module are displayed in the browse.
- 4 ♦ Select just the **Customer SDO** in the browse, and check the **Viewers** toggle box on:



- 5 ♦ Choose the **Viewers** folder tab. Choose the **OE Product Module** and set the **Max Fields Per Column** to a smaller number, perhaps 6 or 8:



- 6 ♦ Press the **Start** button to recreate the dynamic Viewer.

When the generation is complete you have a new dynamic Customer Viewer with three columns, a better default appearance for most windows. [Figure 1–3](#) shows how this Viewer will look when you run the completed example later on.

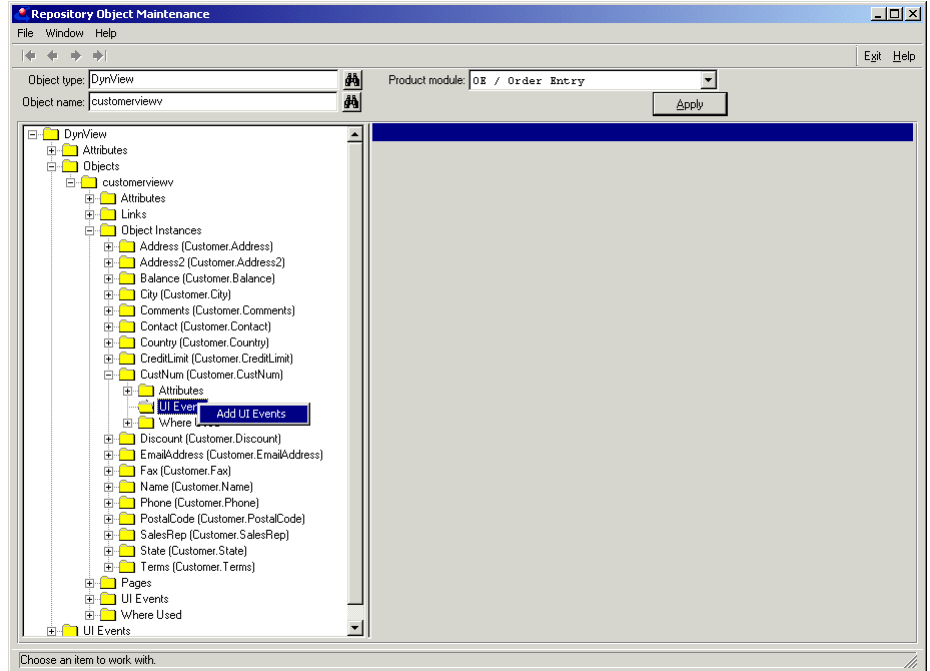
1.5.2 Defining UI events for the viewer

You can define UI Events to which you associate code to execute when the event occurs. This is how you get a user interface trigger in a dynamic Object to run procedural code in a custom super procedure that you attach to the Object. You can define these events either in the Repository Maintenance Tool, or, more simply, in the dynamic property sheet.

To define UI Events for the window in the Repository Maintenance tool:

- 1 ♦ Open the **Repository Maintenance** tool.
- 2 ♦ Select the dynamic Viewer **customerviewv**, and expand it down to its **Object Instances**. When you expand **Object Instances**, you see the field-level objects representing all the fields in the Viewer.

- 3 ♦ Expand the **Customer.CreditLimit** node, and you see subnodes for its attributes, and for its UI Events.
- 4 ♦ Expand the **UI Events** node and you see that none are defined:



- 5 ♦ Right-click on the **UI Events** node and press the **Add UI Events** pop-up button that appears.

The maintenance frame that then appears on the right allows you to define one or more UI Events for the **CreditLimit** field. These are the fields you must fill in:

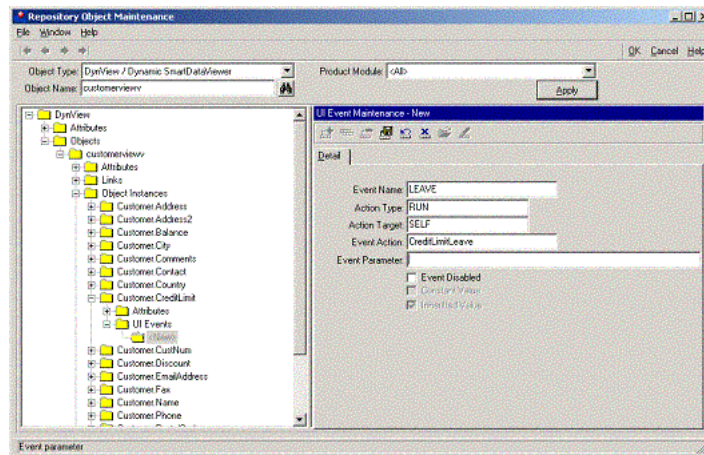
- **Event Name** — This is the name of the Progress 4GL event you want to associate code with, for example: ENTRY, LEAVE, VALUE-CHANGED.
- **Action Type** — Enter **RUN** if you want to run an internal procedure when the event occurs. Enter **PUBLISH** if you want to publish a Progress named event when the UI event occurs (which could be an ADM2 event such as fetchNext or some named event you have defined yourself, and to which your object subscribes). These are the only two valid values for the **Action Type**.

- **Action Target** — This defines the context within which the event should occur. You can specify one of the following values for the **Action Target**:
 - **SELF** — This means that the event should be run or published in the context of the object the code is serving, which effectively means it will be executed in the TARGET-PROCEDURE of the custom super procedure.
 - **CONTAINER** — This means that the event should be run or published in the context of the object that contains the object the custom super procedure is attached to; this is effectively the Container-Source of the object, normally its dynamic SmartWindow.
 - **ANYWHERE** — This value is valid only if the **Action Type** is PUBLISH, and means that the PUBLISH will be done from the custom super procedure itself, not FROM TARGET-PROCEDURE, and not FROM <ContainerHandle>. Since objects do not (and should not) normally subscribe to named events in super procedures directly, this effectively means that the event will be received only by an object that subscribes to it with the keyword ANYWHERE, meaning from any publisher.
 - **AS** — This and all the remaining values are valid only if the **Action Type** is RUN. The value AS means that the action will be run in the default AppServer handle for the client session.
 - **SM** — This value means that the action will be run in the Session Manager.
 - **SEM** — This value means that the action will be run in the Security Manager.
 - **PM** — This value means that the action will be run in the Profile Manager.
 - **RM** — This value means that the action will be run in the Repository Manager.
 - **TM** — This value means that the action will be run in the Translation/Localization Manager.
 - **GM** — This value means that the action will be run in the General Manager.

- **Event Action** — This is the name of the internal procedure to be RUN, or the named event to be published.
- **Event Parameter** — This is the value of an optional parameter of type CHARACTER that will be passed to the procedure that is RUN or published, if specified. If this is not specified, then the procedure is RUN or the event is published with no INPUT parameter.
- **Event Disabled** — If this toggle box is checked on, then the event is disabled and will not occur.

To define a LEAVE event for the **CreditLimit** field that will execute the same check as happens on row display:

- 1 ♦ Type **LEAVE** for the **Event Name**, **RUN** for the **Action Type**, **SELF** for the **Action Target**, and **CreditLimitLeave** for the name of the **Event Action** procedure. Leave the **Event Parameter** blank and of course leave the **Event Disabled** toggle box unchecked:



- 2 ♦ Press the **Save** button to register your event in the Repository.

To use the Dynamics property sheet to define events:

- 1 ♦ Open the **Dynamic Viewer** in the AppBuilder and select the **CreditLimit** field in the design window.
- 2 ♦ Open the **Dynamics** property sheet from the Windows menu.
- 3 ♦ Select the **Events** folder tab in the property sheet and enter the same values as described above for the Leave event of the **Credit Limit** field.

1.5.3 Naming conventions for UI events and super procedures

There is a reason for the form of the name **CreditLimitLeave**. The Progress Dynamics Migration Utility converts static Viewers to dynamic ones. In doing this, it not only creates all the appropriate Repository data to register the Viewer and to represent its fields and all of their attributes, but it also strips any custom code from the static Viewer and writes it to a custom super procedure. Any trigger code defined for user interface events is placed into internal procedures using the naming convention of <WidgetName><EventName>. Hence a UI trigger that is defined ON LEAVE OF CreditLimit in a static Viewer will cause the Migration Utility to generate an internal procedure called CreditLimitLeave with the same code in it, and place this into the custom super procedure for the Viewer. There is no absolute need for you to use the same naming convention, since if you follow the guidelines of this chapter, you will be creating super procedures for your static Viewers yourself, so that there will be no code left for the Migration Utility to convert. Nonetheless, if you follow this naming convention in creating procedures in your custom super procedures, your code will be consistent with what the Migration Utility does for static procedures that need converting.

Obviously, the code the Migration Utility strips out of existing Viewers and puts into a super procedure is not likely to compile or run correctly without modification, because of the issues discussed in this chapter. The intention is simply to salvage the existing code and move it to a place where the developer can then edit it as necessary to make it work in the context of a super procedure.

In addition, the Migration Utility creates a default super procedure for each converted static object that has the base table name of the SDO for the object plus the extension `super.p`. So in this case, if you call your super procedure `customersuper.p`, it will match the naming convention used for converted objects.

1.5.4 Writing the supporting procedures for the UI event

Now you must edit the `customersuper.p` procedure to add the internal procedure to be executed when the UI event occurs so that `CreditLimitLeave` checks first to see if the **CreditLimit** field has been modified. First create a support function, called `widgetModified`, to take care of that check. This function checks the Progress 4GL `MODIFIED` attribute for the field and returns `True` or `False` accordingly:

```
FUNCTION widgetModified RETURNS LOGICAL
( pcField AS CHARACTER ) :
/*-----
Purpose: Returns TRUE if the widget value has been changed.
Params: pcField AS CHARACTER
-----*/
DEFINE VARIABLE hField AS HANDLE NO-UNDO.

hField = widgetHandle(pcField).
IF VALID-HANDLE(hField) AND CAN-SET (hField, 'MODIFIED') AND
hField:MODIFIED THEN
RETURN TRUE.
ELSE RETURN FALSE.
END FUNCTION.
```

The next step is to code the `CreditLimitLeave` procedure. In this example the same condition is duplicated from the `rowDisplay` procedure. Obviously, in a properly completed example this code should be pulled out into a separate internal procedure called from both places:

```
PROCEDURE CreditLimitLeave:
/*-----
Purpose: Check the CreditLimit against the balance on LEAVE.
Parameters: <none>
-----*/
IF widgetModified('CreditLimit') THEN
IF DECIMAL(widgetValue('CreditLimit')) -
DECIMAL(widgetValue('Balance')) < 5000
THEN highlightWidget('Balance').
ELSE unhighlightWidget('Balance').
END PROCEDURE.
```

NOTE: When events are overridden by custom code, the developer needs to ensure that the standard Dynamics behavior is invoked. This typically involves a `RUN SUPER` call, but for widget event overrides other calls must be made. For example, if the `VALUE-CHANGED` event is overridden for a `DataField` on a dynamic viewer, the override code needs to call `valueChanged` in order to ensure that the toolbar state is correctly set.

1.5.5 Testing the dynamic viewer with the UI event

Now if you launch your container window with the dynamic Viewer, you can position to a Customer satisfying the condition and see the result. You can also see the effect of creating a dynamic Viewer with three columns of fields, as shown in [Figure 1–3](#).

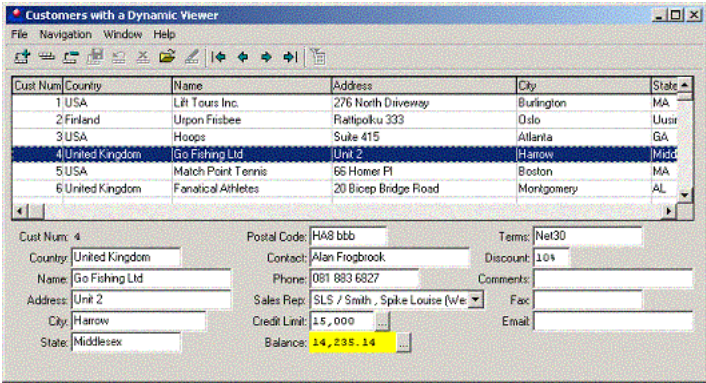


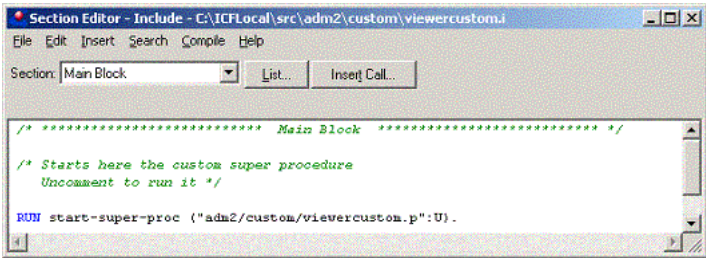
Figure 1–3: Customers with a dynamic viewer

1.6 Moving client logic support to an extended class

This chapter begins with a simple example with all client-side code in a custom super procedure for a single Viewer. Realistically, much of that support code belongs in an extended Viewer class, so that it can serve all Viewers. This section illustrates the distinction between a single custom super procedure and a more generic procedure that serves a whole class of objects.

To extend the Viewer class:

- 1 ♦ Copy src/adm2/custom/viewercustom.i and viewercustom.p to a local directory with the same relative path.
- 2 ♦ Edit viewercustom.i to uncomment the line in the **Main Block** that starts the super procedure viewercustom.p:



- 3 ♦ Move the generically useful code in your custom super procedure to your local copy of `viewercustom.p`. These are all the support functions shown so far:
 - `widgetHandle`
 - `widgetValue`
 - `widgetModified`
 - `highlightWidget`
 - `unhighlightWidget`
- 4 ♦ Move the code that sets the variables where the lists of widget names and handles are stored. In the original `customersuper.p` example, this is done in `initializeObject`. But will this still work when the code is in a super procedure that is part of an extension to the whole Viewer class? No, it will not, because the new `viewercustom.p` procedure might be supporting multiple Viewers at once. As a result, these lists and any other Viewer attributes that the code accesses need to be re-established each time a request comes from a Viewer. This is the first of several basic principles to keep in mind when creating a super procedure.

NOTE: When writing code for a super procedure, always consider whether it will be attached to a single other procedure, or whether it can be attached to multiple other procedures.

In your `customersuper.p`, the super procedure would only be attached to a single instance of one specific Viewer, so it was sufficient to establish needed attribute values on startup, in `initializeObject`. In a more general-purpose procedure, you have to support switching the context of the object the code is supporting on each request, so that it will work properly when there is more than one object using the single super procedure instance. In this case, this means that the code to save off attribute values must go somewhere where it will be executed each time there is a request. In this case, this can be a local version of the `displayFields` procedure, since the `rowDisplay` procedure that does any and all checks on display of a new row is called from there.

So here is a version of `displayFields` for the `viewercustom.p` procedure that gets the attribute values, saves them off, calls `rowDisplay`, and then resets the local variables so there is no danger of a stray call using stale values:

```
PROCEDURE displayFields:
/*-----
  Purpose:      Runs a rowDisplay procedure if there is one to allow
                  custom logic to support any Viewer.
  Parameters:   pcColValues AS CHARACTER.
  Notes:
-----*/
DEFINE INPUT  PARAMETER pcColValues AS CHARACTER  NO-UNDO.

RUN SUPER (INPUT pcColValues). /* Execute the standard display behavior. */

/* Establish the list of fields and handles. */
{get AllFieldNames  gcFields}.
{get AllFieldHandles gcHandles}.

RUN rowDisplay IN TARGET-PROCEDURE NO-ERROR.
ASSIGN gcFields = ""
      gcHandles = "".

END PROCEDURE.
```

When you create this, define the variables `gcFields` and `gcHandles` in the Definitions section of the procedure, so they are scoped to the procedure. Note that because of the principle of designing super procedures so they can support multiple other procedures, this must be done very cautiously. Generally it is not good practice to have **any** variables or other constructs scoped to the super procedure itself, because the values might not apply to the next request of the super procedure, which might come from a different object. In this case, it is something of an optimization to write the code in such a way that, within the context of a single call to `displayFields`, all references to these variables will be valid, so it is better to retrieve them once rather than in every single call to the `widgetHandle` function.

The block of code that sets the variables and calls `rowDisplay` first checks to see whether that procedure has been implemented at all. This is always good practice when you are extending the behavior of any object or class, and could be stated as a second super procedure principle.

NOTE: When extending the behavior of an object or a class of objects using a super procedure, be sensitive to making the additional behavior both optional and efficient.

In this case, the specific rowDisplay for a Viewer will probably be implemented in a custom super procedure for that Viewer. Some Viewers might have this procedure, and some might not. So the behavior is made optional here because the statement to RUN rowDisplay is done NO-ERROR so that there is no error if it is not defined in the TARGET-PROCEDURE. Simply adding a NO-ERROR qualifier on a RUN statement will often be sufficient to make sure that your extension will not break the code in objects that do not use your extension.

Note that you cannot check for rowDisplay by looking for it in the 4GL procedure handle attribute TARGET-PROCEDURE:INTERNAL-ENTRIES because this Progress attribute returns only those entry points that are actually coded in the TARGET-PROCEDURE, or for which there are prototypes compiled into the procedure. In this case, the TARGET-PROCEDURE is the Viewer procedure, and rowDisplay will normally be implemented in its custom super procedure, so the entry point won't be found, as shown in [Figure 1-4](#).

There is one more critical detail in the RUN statement, and that is the reference to TARGET-PROCEDURE. This brings up the third key point in working with super procedures.

NOTE: Always remember to invoke internal procedures and functions IN TARGET-PROCEDURE from a super procedure. The same applies to PUBLISH and SUBSCRIBE statements in super procedures.

This is one of the most common errors made when people write code for super procedures, or even worse, when people move code from an object procedure such as a static Viewer to a super procedure that supports it. Code that worked fine before can simply stop working (that is, stop being executed at all) for this reason. Always keep this relationship between object and super procedure in mind.

Figure 1–4 shows the relationships between the procedures in this case.

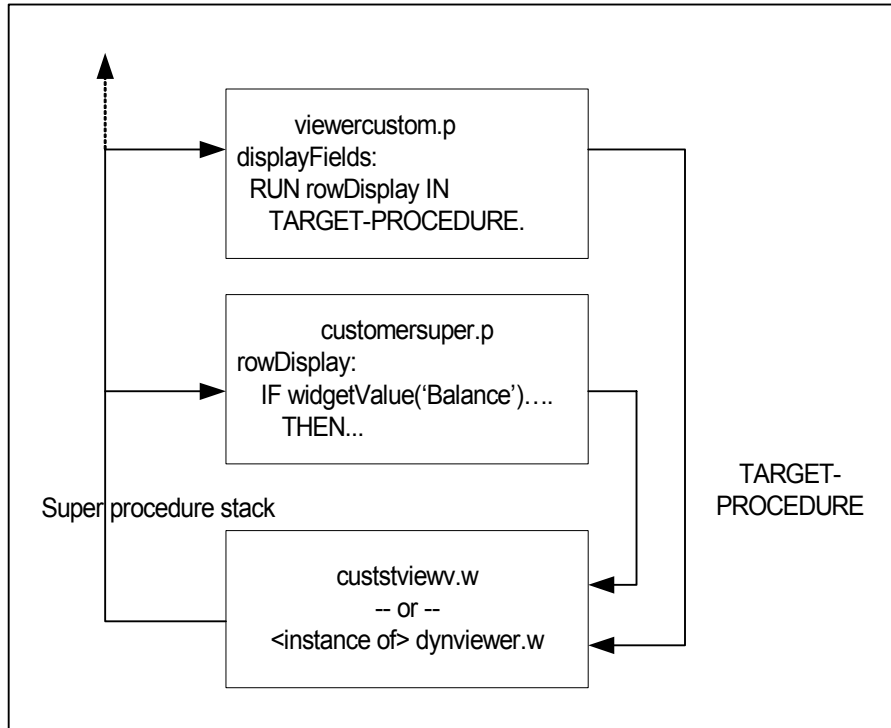


Figure 1–4: Relationships between objects and super procedures

In addition to the standard set of super procedures starting with `smart.p` and ending with `viewer.p`, the Customer Viewer has two additional super procedures, `viewercustom.p`, which applies to the whole class, and `customersuper.p`, which applies just to the one Viewer. This is the case whether the base object procedure is a static Viewer such as `custstvieww.w`, or an instance of the generic dynamic Viewer procedure `rydynvieww.w`. Any statement invoked `IN TARGET-PROCEDURE` starts at the Viewer procedure itself and works its way up the stack until it finds an implementation of the entry point being run. If this entry point has a `RUN SUPER` statement in it, then it continues up the object's super procedure stack looking for the next version of it.

Note that the super procedures themselves are not super procedures of each other. So any call that wants to find an implementation of an entry point in the stack must be made `IN TARGET-PROCEDURE`. The `TARGET-PROCEDURE` of the customer-specific super procedure `customersuper.p` is the Viewer itself. The `TARGET-PROCEDURE` of the generic `viewercustom.p` is likewise the Viewer itself.

The statement `RUN rowDisplay IN TARGET-PROCEDURE` attempts to run it in the Viewer procedure. According to recommended coding practice, `rowDisplay` will not be found there, because there **is** no custom code in the Viewer itself, even if it is a static procedure. But the Progress interpreter then looks up the Viewer's super procedure stack and finds `rowDisplay` in `customersuper.p`, and executes it there.

This leads to an interesting problem with this structure where 4GL functions are concerned. The Progress compiler must have a function definition for any static function reference in a procedure's code. The only alternative to this is to use the `DYNAMIC-FUNCTION` syntax, which is rather clumsy for a model where you want the simplest possible coding style in your client logic.

So you want to be able to reference functions like `widgetHandle` directly in code in `customersuper.p`, even though the functions are no longer defined there. To do this without error, you must include function prototypes of all the support functions in your super procedure.

There is a Progress tool to generate the prototypes for you, called `ProtoGen`. It is available from the `ProTools` palette, as shown in [Figure 1-5](#).



Figure 1-5: ProTools palette

In the Prototype Generator, you first enter the name of the super procedure it should generate prototypes for. In this case, it is the local version of `src/adm2/custom/viewercustom.p`. You then enter the name of the include file that the tool generates containing the prototype definitions. In the example it should be called `viewcustomprto.i`, as shown in [Figure 1-6](#).

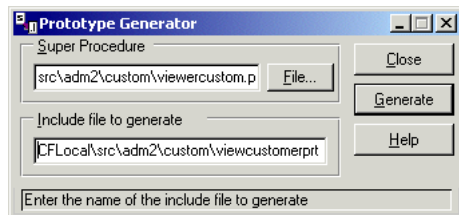


Figure 1-6: Prototype Generator window

Press the **Generate** button to create the include file.

In its default form, however, it is not exactly what you need. This is because the generator assumes that the include file will be added to an object procedure such as the Viewer procedure, so that functions in its super procedures can be referenced. For this reason, it creates function prototypes defined as being IN SUPER, meaning that the interpreter should invoke the function, and it will be found in a super procedure. For example:

```
FUNCTION widgetHandle RETURNS HANDLE
  (INPUT pcWidget AS CHARACTER) IN SUPER.
```

But in the present case, the function references are in another super procedure, `viewercustom.p`, which, as we explained, is not a super procedure of `customersuper.p`. So the function prototypes need to be modified to tell the interpreter to execute the function IN TARGET-PROCEDURE. When this happens, the interpreter searches up the stack and finds them in another super procedure of the Viewer, namely `viewercustom.p`:

```
/*
* Prototype include file: C:\ICFLocal\src\adm2\custom\viewcustomprto.i
* Created from procedure: C:\ICFLocal\src\adm2\custom\viewercustom.p at 13:31
on 06/17/02
* by the PROGRESS PRO*Tools Prototype Include File Generator
*/

FUNCTION disableWidget RETURNS LOGICAL
  (INPUT pcField AS CHARACTER) IN TARGET-PROCEDURE.
FUNCTION enableWidget RETURNS LOGICAL
  (INPUT pcField AS CHARACTER) IN TARGET-PROCEDURE.
FUNCTION highlightWidget RETURNS LOGICAL
  (INPUT pcField AS CHARACTER) IN TARGET-PROCEDURE.
FUNCTION setWidgetAttr RETURNS LOGICAL
  (INPUT pcField AS CHARACTER,
   INPUT pcAttr AS CHARACTER,
   INPUT pcValue AS CHARACTER) IN TARGET-PROCEDURE.
FUNCTION unhighlightWidget RETURNS LOGICAL
  (INPUT pcField AS CHARACTER) IN TARGET-PROCEDURE.
FUNCTION widgetHandle RETURNS HANDLE
  (INPUT pcWidget AS CHARACTER) IN TARGET-PROCEDURE.
FUNCTION widgetModified RETURNS LOGICAL
  (INPUT pcField AS CHARACTER) IN TARGET-PROCEDURE.
FUNCTION widgetValue RETURNS CHARACTER
  (INPUT pcField AS CHARACTER) IN TARGET-PROCEDURE.
```

NOTE: You can remove the prototype for the internal procedure `displayFields`, which is not needed. This is what the resulting include file should look like. The exact function prototypes depend on how many supporting functions you have in `viewercustom.p`.

Next, modify the Definitions section of your `customersuper.p` to include the prototype file:

```
{src/adm2/custom/viewcustomprto.i}
```

Now any references to the supporting functions in your custom super procedure will compile and execute successfully.

This entire exercise of moving commonly used code into a higher level in the super procedure stack illustrates a fourth principle to keep in mind whenever you are developing an application.

NOTE: Always factor out common code into the highest possible point in the super procedure stack.

A key aspect of Progress Dynamics as an application framework is that it provides a great deal of standard behavior that can be used in all objects of a given type. Whenever you develop code of your own, try to use it to extend that standard behavior whenever possible. In a sense this is just good normal programming practice, employing the concept of modular programming. However, Progress Version 9 with its super procedures, published events, and object attributes provides a specific structure within which you can implement modular code. Whenever you write 4GL code, always consider what general use it can be, and always take the time to remove commonly useful code and move it up to a higher level in the tree. When you do this, consider the range of objects that might want to take advantage of it. Does the code apply to just a subset of Viewers, for example, so that it would be worthwhile to create a new specialized Viewer class for just those objects? Or does it apply to all Viewers? To all Datavis objects, which are visual objects that display database data? To all visual objects? To all client-side objects? This same set of questions can be asked about behavior you are adding to any part of the overall SmartObject class tree. Taking the time to write code that can then be reused by many other objects is an investment that will pay off many times over. The continuation of the Viewer event example below is another illustration of this.

If you look at the UI Event example, you will see that the supporting code needs a further refinement. In the case of the `rowDisplay`, this is always called in the context of a single row's display event, and putting code into `displayFields` to set and unset the local context (the values of `AllFieldNames` and `AllFieldHandles`) can safely be done there. But a UI Event can occur on any event for any widget, so it is not as easy to encapsulate the context setting in such a general way.

The code for the example event procedure CreditLimitLeave needs to be modified to set and unset the context. Because this might be happening all over the place in your code, you should extract the code out into another pair of functions implemented in `viewercustom.p`, which you can call `setDisplayContext` and `clearDisplayContext`:

```
FUNCTION setDisplayContext RETURNS LOGICAL
( ) :
/*-----
Purpose: Establish the list of fields and handles in the current
Viewer.
Notes:
-----*/

{get AllFieldNames gcFields}.
{get AllFieldHandles gcHandles}.
RETURN TRUE.

END FUNCTION.
```

```
FUNCTION clearDisplayContext RETURNS LOGICAL
( ) :
/*-----
Purpose: Clears any local storage of Viewer context information.
Notes:
-----*/

ASSIGN gcFields = ""
      gcHandles = "".
RETURN TRUE.

END FUNCTION.
```

Now the displayFields procedure in viewercustom.p can be cleaned up a little to use these two functions:

```

PROCEDURE displayFields:
/*-----
  Purpose:      Runs a rowDisplay procedure if there is one to allow
                  custom logic to support any Viewer.
  Parameters:   pcColValues AS CHARACTER.
  Notes:
-----*/
DEFINE INPUT  PARAMETER pcColValues AS CHARACTER  NO-UNDO.

  RUN SUPER (INPUT pcColValues).
  DYNAMIC-FUNCTION ('setDisplayContext' IN TARGET-PROCEDURE).
  RUN rowDisplay IN TARGET-PROCEDURE NO-ERROR.
  clearDisplayContext().
END PROCEDURE.

```

So why does the reference to setDisplayContext have to execute IN TARGET-PROCEDURE (which in turn requires the use of the DYNAMIC-FUNCTION syntax, which allows you to specify where the function is invoked), even though the setDisplayContext function is implemented in the same source procedure as this version of displayFields? It is because of another aspect in the use of TARGET-PROCEDURE that it is important to go over.

In addition to using IN TARGET-PROCEDURE in code references that are written in the object the code supports, or that are written in another super procedure in the stack, it is important to remember to use IN TARGET-PROCEDURE even if the entry point your code is running is implemented in the super procedure itself, **if** the entry point makes another reference to the TARGET-PROCEDURE. This is because the TARGET-PROCEDURE reference in the called entry point only evaluates properly if it has bounced back from the base object itself. If the entry point is instead invoked directly in THIS-PROCEDURE (which is implicit if there is no qualifier on the RUN statement or function invocation), then within the called entry point, the value of TARGET-PROCEDURE is the same as THIS-PROCEDURE, since that is where the call originated.

Figure 1–7 illustrates the difference between the references.

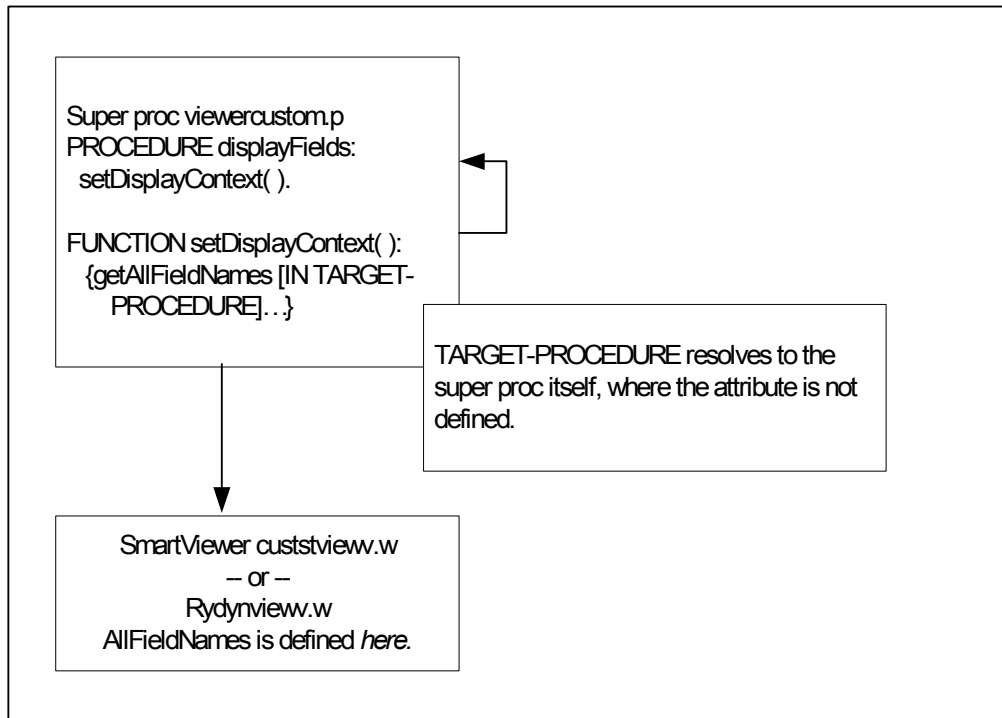


Figure 1–7: Improper function reference without TARGET-PROCEDURE

In contrast, the `setDisplayContext` reference in the example uses the special `{get}` pseudo-syntax to retrieve the attribute values, and this includes the necessary reference to `TARGET-PROCEDURE` to point at the object where the attribute is defined, so any reference to the function must itself be made `IN TARGET-PROCEDURE` for things to resolve properly.

So with the function reference explicitly made IN TARGET-PROCEDURE, the function itself will be able to resolve it properly, as [Figure 1–8](#) shows.

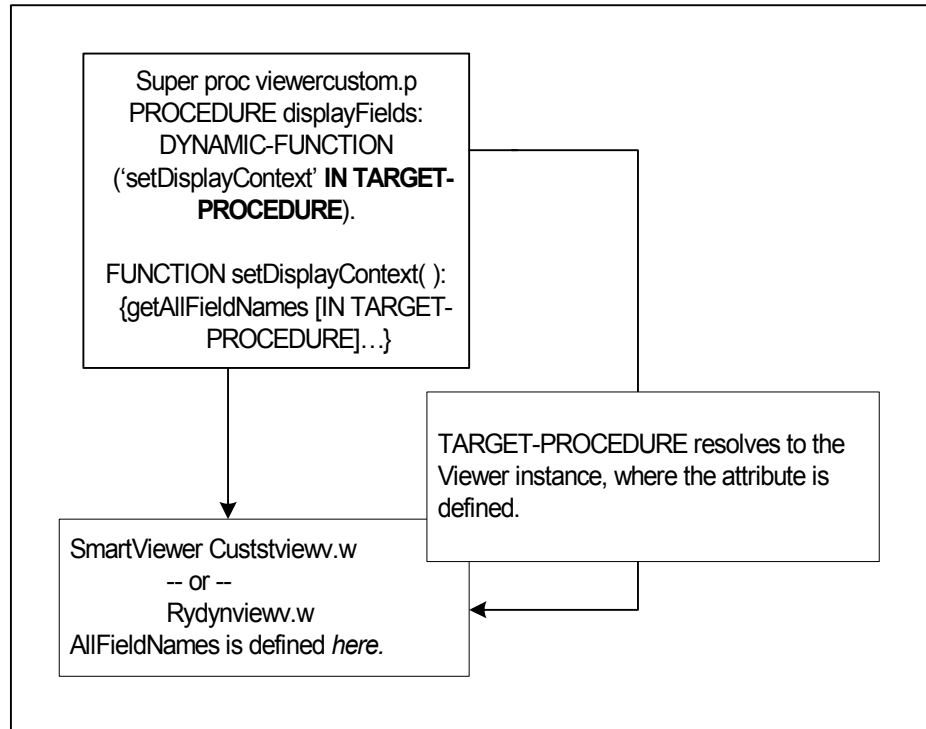


Figure 1–8: Proper function reference with TARGET-PROCEDURE

Now each event procedure can set and clear the context itself, so that all references to the fields and their handles within all the functions that are invoked resolve properly without the attributes having to be retrieved each time:

```
PROCEDURE CreditLimitLeave:
/*-----
Purpose:      Check the CreditLimit against the balance on LEAVE.
Parameters:  <none>
-----*/
setDisplayContext().

IF widgetModified('CreditLimit') THEN
IF DECIMAL(widgetValue('CreditLimit')) -
    DECIMAL(widgetValue('Balance')) < 5000
    THEN highlightWidget('Balance').
ELSE unhighlightWidget('Balance').

clearDisplayContext().

END PROCEDURE.
```

To bring all this into sync:

- 1 ♦ Make the changes to the `viewercustom.p` procedure, save it and compile it.
- 2 ♦ Regenerate the `viewcustomprto.i` prototype include file using the ProtoGen tool.
- 3 ♦ Edit the `viewcustomprto.i` file to change `IN SUPER` to `IN TARGET-PROCEDURE`.
- 4 ♦ Compile all the Viewers that need to use this support procedure. This includes any static Viewers such as `custstvieww.w` from the static example, as well as the generic dynamic Viewer procedure `rydynvieww.w`.

As noted before, the formula for the Credit Limit check itself, which is duplicated in `rowDisplay` and `CreditLimitLeave`, should be removed to another function where it only needs to be defined once.

There is another way in which this example is simplified to the point where it does not actually set a good example: The formula that says that a Customer Balance within \$5000 of the Credit Limit represents a warning condition is really business logic, even though the way that condition is dealt with is simply to change something in the user interface. If that logic needs to be changed, or modified depending on the user organization or other conditions, then you have a maintenance headache to deal with because the code that expresses the condition is compiled into client user interface procedures and deployed to all the client systems.

It is much better to bring as much of the logic as possible over from the server at run time. At a minimum this means that the \$5000 figure should be turned into an application property that is retrieved as needed, perhaps at startup along with a whole set of other client logic properties, in a single call to a business logic procedure running on the server. In that way the logic that maintains that figure and determines what the proper figure is for this user or this organization, can be maintained on the server along with other business logic.

1.7 Running a PLIP from client logic

A Progress Dynamics *Persistent Library of Internal Procedures (PLIP)* is simply a Progress 4GL procedure that is designed to hold business logic and run on the server side of a distributed application, to be invoked from other code either on the server or on the client side. Naturally, you want to avoid making calls from client code to the server whenever possible, in order to maximize application performance, but sometimes it is necessary to make extra calls beyond those that the framework uses to populate the client side objects and return data to the server. This section illustrates how you can use the Progress Dynamics Session Manager to invoke server-side logic that accesses the application database.

In this example, you want to allow the user to make a call back to the server to obtain the latest Credit information for a Customer during OrderLine maintenance. Again, whenever possible, this kind of information should always be brought over along with other information the framework objects are already handling for you, using a device such as calculated fields in an SDO. But for the sake of the example, we presume that this must indeed be an immediate call back to the server.

To run a Progress Dynamics PLIP:

- 1 ♦ Build the business logic procedure itself. Press the **New** button in the AppBuilder menu, and select **Structured PLIP** as the object type. See the [Progress Dynamics Developer's Guide](#) for more information on the structure of the PLIP. In the Section Editor, add a new internal procedure called AvailCredit, which looks like this:

```
PROCEDURE AvailCredit:
/*-----
Purpose:
Parameters:  <none>
Notes:
-----*/
DEFINE INPUT  PARAMETER pcKeyType      AS CHARACTER  NO-UNDO.
DEFINE INPUT  PARAMETER piKeyNum       AS INTEGER    NO-UNDO.
DEFINE OUTPUT PARAMETER pdAvailCredit AS DECIMAL     NO-UNDO INIT ?.

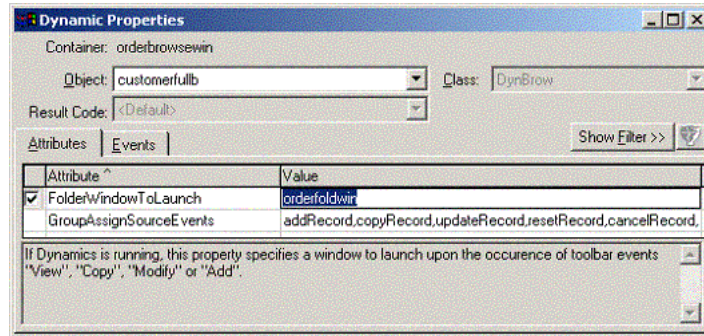
IF pcKeyType = "OrderNum" THEN
DO:
    FIND Order WHERE Order.OrderNum = piKeyNum NO-LOCK NO-ERROR.
    IF AVAILABLE (Order) THEN
        FIND Customer WHERE Customer.CustNum = Order.CustNum NO-LOCK.
    END.
ELSE IF pcKeyType = "CustNum" THEN
    FIND Customer WHERE Customer.CustNum = piKeyNum NO-LOCK NO-ERROR.
IF AVAILABLE (Customer) THEN
    pdAvailCredit = Customer.CreditLimit.
END PROCEDURE.
```

This takes a KeyType parameter to tell it whether the key being passed in is a Customer Number or an Order Number, and then uses the key value itself to retrieve the appropriate Customer record. It returns that Customer's CreditLimit as the Output parameter. Again, real-life PLIPs will normally do more serious work that could not be returned in a simple field as part of an SDO.

- 2 ♦ Save this PLIP in the oe_ directory (or wherever you are organizing your objects) as customerplip.p.

Now you need to build a window where you support Order and OrderLine maintenance. Any combination of objects that manages these tables will do, but the step describes the combination used for the present example. The example presumes that you have generated SDOs and dynamic Viewers and Browsers for the Sports2000 tables.

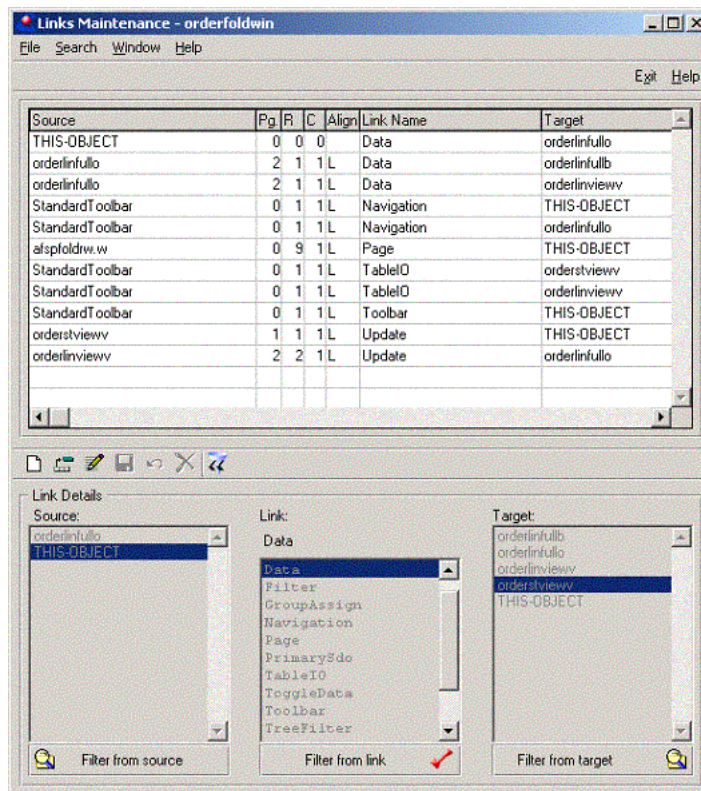
- 3 ♦ Build an Order browse window as an Independent Window. (For more information on building a browse window, see the [Progress Dynamics Developer's Guide](#).) Use the layout called **rywinObjCont** as the Container Template, and substitute the SDO **orderfullo** for the template SDO, and the dynamic Browser **orderfullb** for the template Browser. Select the dynamic Browser in the layout grid, right click to bring up the Property Sheet, and type **orderfoldwin** as the FolderWindowToLaunch property. This is the maintenance window that starts when the user double-clicks on a row in the browse or selects one of the update-related buttons in the toolbar:



- 4 ♦ Save this container as **orderbrowsewin**.
- 5 ♦ Create the Order Maintenance window. Create a new Dependent Window, which gets the Order Number as input from the order browse window. Use the folder window **rywinFolder** as the existing container to create this from.
- 6 ♦ Select Page 1 in the Container Builder, which is part of the container template. This page layout is designed for a single static Viewer. If you have a dynamic Order Viewer you would like to use instead, you can unsubstitute the page layout **rypagDynView** as the template. Substitute the Order Viewer (our example static Viewer is **orderstvieww**, previously built in the AppBuilder) for the template Viewer.
- 7 ♦ Add a page 2 to display and maintain OrderLines. Add the dynamic OrderLine SDO **orderlinfullo** to this page, along with the dynamic Browser **orderlinfullb**, and the dynamic Viewer **orderlinvieww**.
- 8 ♦ Select the **orderlinfullo** SDO in the grid, open the property sheet for it, and type **ordernum,ordernum** as the **Foreign Fields**. This will filter the OrderLine SDO for OrderLines of the currently selected Order.
- 9 ♦ Fill out the necessary Data and Update links between the OrderLine Objects on Page 2 so that records are passed back and forth properly.

- 10 ♦ Create a TableIO link from the StandardToolbar to the static Order Viewer `orderstvieww`. This allows the Toolbar on page 0 to maintain the Order displayed on page 1. Create another TableIO link from it to the OrderLine Viewer `orderlinvieww` as well.
- 11 ♦ Create a Data link from THIS-OBJECT (which represents the Container itself) to the static Order Viewer. This populates the Viewer with data from the record selected in the browse window.
- 12 ♦ Create an Update link from the static Order Viewer to THIS-OBJECT. This will pass the update back to the Order SDO maintained in the browse window.
- 13 ♦ Create a Data link from THIS-OBJECT to the OrderLine SDO **orderlinfullo**. This will pass the current Order number in to the OrderLine SDO, which, using the **Foreign Fields** information you entered, will filter the OrderLines for the current Order.

When you are done, your Links Maintenance window should look like this:



- 14 ♦ Save this as **orderfoldwin**, and you are ready to proceed.

- 15 ♦ The next task is to add a fill-in to the dynamic OrderLine Viewer. Open the **orderlinviewv** Viewer in the AppBuilder, using the **Open Object** dialog box. Add the fill-in to the layout just as you would for a static Viewer. Name it **AvailCredit** and give it a label of **Available Credit** and a format of >>>>**9.99**.
- 16 ♦ Define a UI Event for this new field, using the dynamic property sheet from the AppBuilder, to trigger the internal procedure inside the custom super procedure that will in turn execute the server-side code to retrieve the Credit information.

It might be a better user interface if there was a button for the user to press to retrieve the Credit information, but to keep things simple, retrieve the information on LEAVE of the dynamic fill-in itself by continuing the procedure.
- 17 ♦ Type an **Event Name** of **LEAVE**, an **Action Type** of **RUN**, an **Action Target** of **SELF**, and for the **Event Action** the internal procedure name **ShowAvailCredit**. This defines a UI event that, on Leave of the dynamic fill-in, will run the procedure ShowAvailCredit in the TARGET-PROCEDURE.
- 18 ♦ Save this new UI Event.
- 19 ♦ Create the custom super procedure for the OrderLine Viewer by selecting **New**→**Structured Procedure** in the AppBuilder. Use the Section Editor to create an internal procedure called **ShowAvailCredit**. This procedure first retrieves the lists of field names and field handles from its TARGET-PROCEDURE, as the earlier example did.

Note that these two statements are effectively identical to this {get} include file syntax:

```
{get AllFieldNames cFieldNames}.  
{get AllFieldHandles cFieldHandles}.
```

```
{get DataSource hDataSource}.  
{get DataSource hDataSource hDataSource}.
```

This version might look confusing because the handle is reused to retrieve the second Data Source from the first. The second statement amounts to this:

NOTE: Get the value of the DataSource attribute and put it into the variable hDataSource. Use hDataSource itself as the handle of the object to query for its DataSource.

If this optional third argument to the {get} include file is not there, it uses the TARGET-PROCEDURE by default.

In some cases there is no Order SDO above the OrderLine SDO. Because you should always write code that is as general-purpose as possible, it is good to allow for any combination of Objects in any custom code you write. So the block of code checks to see whether this second DataSource in fact returned a valid procedure handle. If it did, then it uses an SDO function to retrieve the value of the **CustNum** field for the current row:

```
iKeyNum = INTEGER(DYNAMIC-FUNCTION  
    ('columnStringValue':U IN hDataSource, "CustNum":U)).
```

It sets another variable to indicate that it retrieved the CustNum. But if that second handle is not valid, then the code tries another approach, instead retrieving the value of the **Order Number** field from the OrderLine Viewer, using the field names and handles lists:

```
ASSIGN iLookup = LOOKUP("OrderNum", cFieldNames)  
    hField = WIDGET-HANDLE(ENTRY(iLookup, cFieldHandles))  
    iKeyNum = INTEGER(hField:SCREEN-VALUE)
```

It sets the other variable to indicate that what it retrieved is the Order Number.

Save this procedure and register it in the Repository. Then associate it with the dynamic OrderLine Viewer in the AppBuilder. To do this, open the OrderLine Viewer, double-click on it to bring up its App Builder property sheet, and set the Custom Super Procedure field to **order1nsuper.p**. Note that you do not specify the .p filename extension or the relative pathname because this information was placed in the Repository when you registered the procedure.

This simplified example might seem a bit contrived, but it demonstrates a couple of basic principles that are important to keep in mind when you write this kind of code:

- **Always write your code to work in as many client Object configurations as possible** — Do not assume a particular combination of Objects in the client container, and do not assume what the exact contents of those objects will be. This makes your code more flexible and reusable, and less prone to errors if other Objects in the user interface change.

- **Retrieve information from other Objects if you need to** — Remember that you can get data from anywhere else on the client, as long as you are flexible about how you look for it. The link functions in each SmartObject will connect you to other Objects, and various types of functions can retrieve information from those Objects. Getting the Object's ContainerSource and then searching the container's ContainerTargets can allow you to locate any other Object in the same container window as the Object you started from. If a search such as this saves you from an extra AppServer hit to retrieve data that might be elsewhere on the client, it is well worth it.

1.7.1 Using launch.i or dynlaunch.i to run the PLIP

Next comes the key part of the exercise: running the business logic procedure that is located where the data is, and which will return the needed Credit value. The standard Progress Dynamics include file `launch.i` takes a series of named arguments that are explained in the *Progress Dynamics Developer's Guide*. In this example it runs the internal procedure `AvailCredit` in the persistent procedure `customerplip.p`, passing it the parameter list specified, deleting the running instance of `customerplip.p` when it is done:

```
{!launch.i &PLIP = 'oe/customerplip.p'
  &IProc = 'AvailCredit'
  &PList = "(INPUT cKeyType,
            INPUT iKeyNum,
            OUTPUT dAvailCredit)"
  &AutoKill = YES
  &Perm = NO}
```

There is now a new alternative to `launch.i` that is more efficient in cases where you do not need to retain the procedure handle of the business logic procedure after your request has returned. This include file is `dynlaunch.i`, and it takes advantage of the dynamic CALL object added to the Progress 4GL in Progress Version 9.1D to carry out the entire request in a single AppServer call, running code on the server that starts the PLIP if it is not already running, runs the requested internal procedure, passes in any INPUT parameters, receives back any OUTPUT parameters, and deletes the PLIP if it was started by the call.

The `dynlaunch.i` file uses essentially the same named arguments as `launch.i`, except that INPUT and OUTPUT parameters are expressed as named arguments rather than as a single quoted string. This is because the arguments are actually evaluated on the server dynamically using a CALL object to construct the call at run time. For each parameter to the internal procedure you are running, you must specify three named include file arguments. In each case *n* represents the position of the parameter in the calling sequence:

- **Moden** — INPUT, OUTPUT, or INPUT-OUTPUT.
- **&Parmn** — The parameter name as a single-quoted string.
- **&DataTypen** — The data type of the parameter as an unquoted string.

Also, because the handle of the server-side procedure cannot be made available to the client after the call is complete, the `&AutoKill` and `&Perm` arguments do not apply. A call to `dynlaunch.i` to accomplish the same request as the previous call to `launch.i` looks like this:

```
{dynlaunch.i &PLIP = 'oe/customerplip.p'  
             &IProc = 'AvailCredit'  
             &mode1 = INPUT   &parm1=cKeyType &dataType1 = CHARACTER  
             &mode2 = INPUT   &parm2=iKeyNum &dataType2 = INTEGER  
             &mode3 = OUTPUT  &parm3=dAvailCredit &dataType3 = DECIMAL  
}
```

If you use `dynlaunch.i`, you must also include this reference in the **Definitions** section of your procedure so that the variables it uses are properly defined:

```
{dynlaunch.i &define-only=YES}
```

Next comes the standard error checking include file, `checkerr.i`, which is described in the [Progress Dynamics Developer's Guide](#):

```
{checkerr.i &display-error = YES  
           &return-only = YES}
```

These arguments mean that an error will be displayed if there is one (because the code is being executed on the client, where the error dialog can be displayed), and the code will return out of the procedure if there is an error, but will not pass a specific error condition to its caller.

Finally, the code locates the new dynamic fill-in field AvailCredit in the Viewer, and sets its SCREEN-VALUE to the Credit limit that came back from the serve:

```
ASSIGN iLookup = LOOKUP("AvailCredit", cFieldNames)
      hField = WIDGET-HANDLE(ENTRY(iLookup, cFieldHandles))
      hField:SCREEN-VALUE = STRING(dAvailCredit).
```

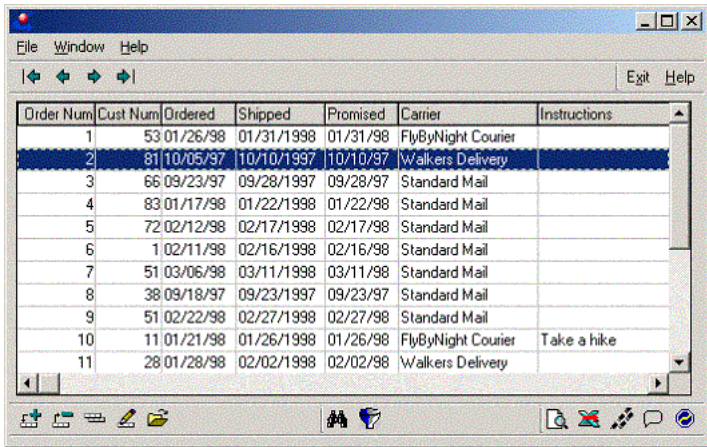
1.7.2 Testing the completed business logic procedure

Here is the complete procedure:

```
PROCEDURE ShowAvailCredit:
/*-----
Purpose:      Retrieves the Customer's Credit Limit from the server
               and displays it in the Viewer.
-----*/
DEFINE VARIABLE hDataSource AS HANDLE NO-UNDO.
DEFINE VARIABLE cFieldNames AS CHARACTER NO-UNDO.
DEFINE VARIABLE cFieldHandles AS CHARACTER NO-UNDO.
DEFINE VARIABLE iLookup AS INTEGER NO-UNDO.
DEFINE VARIABLE hField AS HANDLE NO-UNDO.
DEFINE VARIABLE iKeyNum AS INTEGER NO-UNDO.
DEFINE VARIABLE cKeyType AS CHARACTER NO-UNDO.
DEFINE VARIABLE dAvailCredit AS DECIMAL NO-UNDO.
  {get AllFieldNames cFieldNames}.
  {get AllFieldHandles cFieldHandles}.
  hDataSource = DYNAMIC-FUNCTION('getDataSource':U IN TARGET-PROCEDURE).
  hDataSource = DYNAMIC-FUNCTION('getDataSource':U IN hDataSource).
  IF VALID-HANDLE (hDataSource) THEN
    ASSIGN cKeyType = "CustNum"
           iKeyNum = INTEGER(DYNAMIC-FUNCTION
                             ('columnStringValue':U IN hDataSource, "CustNum":U)).
  ELSE
    ASSIGN iLookup = LOOKUP("OrderNum", cFieldNames)
           hField = WIDGET-HANDLE(ENTRY(iLookup, cFieldHandles))
           iKeyNum = INTEGER(hField:SCREEN-VALUE)
           cKeyType = "OrderNum".
  {dynlaunch.i &PLIP = 'oe/customerplip.p'
    &IProc = 'AvailCredit'
    &mode1 = INPUT &parm1='cKeyType' &dataType1 = CHARACTER
    &mode2 = INPUT &parm2='iKeyNum' &dataType2 = INTEGER
    &mode3 = OUTPUT &parm3='dAvailCredit' &dataType3 = DECIMAL
  }
  {checkerr.i &display-error = YES
    &return-only = YES}
  ASSIGN iLookup = LOOKUP("AvailCredit", cFieldNames)
        hField = WIDGET-HANDLE(ENTRY(iLookup, cFieldHandles))
        hField:SCREEN-VALUE = STRING(dAvailCredit).
END PROCEDURE.
```

To try out the finished application window:

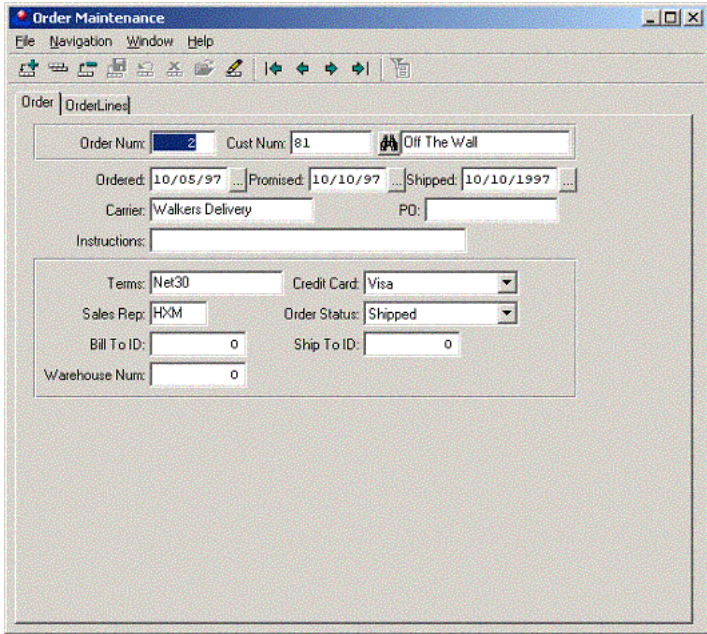
- 1 ♦ Launch **orderbrowsewin** and select an Order:



The screenshot shows the 'orderbrowsewin' application window. It has a menu bar with 'File', 'Window', and 'Help'. Below the menu bar is a toolbar with navigation icons and 'Exit' and 'Help' buttons. The main area contains a table with the following columns: Order Num, Cust Num, Ordered, Shipped, Promised, Carrier, and Instructions. The table lists 11 orders, with the second order (Order Num: 2, Cust Num: 81) highlighted in blue. The bottom of the window features a status bar with various icons.

Order Num	Cust Num	Ordered	Shipped	Promised	Carrier	Instructions
1	53	01/26/98	01/31/1998	01/31/98	FlyByNight Courier	
2	81	10/05/97	10/10/1997	10/10/97	Walkers Delivery	
3	66	09/23/97	09/28/1997	09/28/97	Standard Mail	
4	83	01/17/98	01/22/1998	01/22/98	Standard Mail	
5	72	02/12/98	02/17/1998	02/17/98	Standard Mail	
6	1	02/11/98	02/16/1998	02/16/98	Standard Mail	
7	51	03/06/98	03/11/1998	03/11/98	Standard Mail	
8	38	09/18/97	09/23/1997	09/23/97	Standard Mail	
9	51	02/22/98	02/27/1998	02/27/98	Standard Mail	
10	11	01/21/98	01/26/1998	01/26/98	FlyByNight Courier	Take a hike
11	28	01/28/98	02/02/1998	02/02/98	Walkers Delivery	

- 2 ♦ Double-click on the Order to bring up **orderfoldwin**:



The screenshot shows the 'Order Maintenance' application window. It has a menu bar with 'File', 'Navigation', 'Window', and 'Help'. Below the menu bar is a toolbar with various icons. The main area is divided into two tabs: 'Order' and 'OrderLines'. The 'Order' tab is active, showing a form with the following fields: Order Num (2), Cust Num (81), Off The Wall (checkbox), Ordered (10/05/97), Promised (10/10/97), Shipped (10/10/1997), Carrier (Walkers Delivery), PO (empty), Instructions (empty), Terms (Net30), Credit Card (Visa), Sales Rep (HXM), Order Status (Shipped), Bill To ID (0), Ship To ID (0), and Warehouse Num (0).

Remember that some of the links you defined in the Container Builder set up a connection from the browse window to the maintenance window. These are the links that have **THIS-OBJECT** (the maintenance window) as the Source for the link. These pass-through links allow the Order data to be retrieved and displayed by the Order Viewer on Page 1, and also allow the OrderLine SDO on Page 2 to retrieve the Order Number to use for filtering its own query.

- 3 ♦ Select **Page 2**. Then select the **Available Credit** field you created and tab out of it. The Leave trigger fires, which runs showAvai1Credit in the Viewer's custom super procedure order1insuper.p. This in turn runs AvailCredit in the customerplip.p procedure, which returns the Customer Credit Limit for display:

The screenshot shows the 'Order Maintenance' window. It contains a table with the following data:

Order Num	Line Num	Item Num	Price	Qty	Discount	Extended Price	Order Line Status
2	1	19	2.75	48	25%	99.00	Shipped
2	2	49	6.78	14	25%	71.19	Shipped
2	3	13	10.99	79	25%	851.16	Shipped
2	4	37	12.50	23	25%	215.63	Shipped
2	5	42	37.00	64	25%	1,776.00	Shipped

Below the table, there is a summary section with the following fields:

- Order Num: 2
- Line Num: 1
- Item Num: 19
- Price: 2.75
- Qty: 48
- Discount: 25%
- Extended Price: 99.00
- Order Line Status: Shipped
- Available Credit: 6200.00

1.8 Building a custom super procedure for a browser

Any Progress Dynamics object can have a custom super procedure, not just a Viewer. To illustrate this, you can build one for a dynamic Browser to do the same thing the first Viewer example did: to highlight a browse cell if the Balance is within \$5000 of the Credit Limit.

Remember that there are only limited visual modifications that can be made to an individual browse cell on display of a particular row, including changing the background color, and the font or the format of the data. So for your purposes, the highlight action works well, since it changes the background color.

You can move some of the same code from the Viewer example into a custom super procedure for a Browser. To the extent that the same actions apply, this code could be factored out all the way up to the level of the `datavis` or even the `visual` class, but you do not go that far in this example.

To build the example, create a new structured procedure called `custbsuper.p`. First it needs a local version of `initializeObject`. Like the first version of the Viewer super procedure, this code assumes that it is serving only a single browser instance, so `initializeObject` stores attribute values locally for other functions to use. First the code sets the `ScrollRemote` attribute in the Browser to `True`. This attribute causes the Browser to define the internal procedure `rowDisplay` as a trigger procedure for the Browser's `ROW-DISPLAY` Progress event, which occurs each time a row of data is displayed to the browse widget. This event gives you the opportunity to intercept the display, check data values, and make limited changes (as noted above) to the cells in the browse.

This section contains information on custom super procedures for browsers, including:

- [Useful browser properties](#)
- [Coding the `rowDisplay` procedure](#)
- [Creating the other support functions](#)
- [Defining the custom super procedure for a browser](#)
- [Changing browser attributes for the master and instance](#)

1.8.1 Useful browser properties

After this the code runs the standard `initializeObject` behavior and then captures three property values locally:

- The **DisplayedFields** property holds a comma-separated list of all the column names in the browse. Since you are only interested (for the purposes of this example at least) in looking at values in the browse itself, this is all you need. Note that the `AllFieldNames` and `AllFieldHandles` properties are defined for the Browser, as they are for Viewers, but they hold only the name and handle of the browse widget itself, along with any additional fields or other widgets that you might add to the Browser's frame, not the names and handles of the browse columns themselves. So they are not useful for your purposes here.
- The **FieldHandles** property holds a comma-separated list of the widget handles of the browse cells, in the same order as the `DisplayedFields` list.

- The **QueryRowObject** property holds the handle of the record buffer for the RowObject table in the SDO that is used to populate the browse. This is needed because, within the ROW-DISPLAY event trigger where the custom code executes, it is not possible to inspect the SCREEN-VALUES of the cells themselves, so the widgetValue function looks at the corresponding RowObject fields instead:

```

PROCEDURE initializeObject
/*-----
----
Purpose:   ScrollRemote enables the rowDisplay trigger on the browse's
           ROW-DISPLAY event, used to customize cell color etc.
Parameters: <none>
Notes:
-----
--*/

DYNAMIC-FUNCTION("setScrollRemote" IN TARGET-PROCEDURE,
                 TRUE).

RUN SUPER.

{get DisplayedFields gcFields}.
{get FieldHandles gcHandles}.
{get QueryRowObject ghBuffer}.

END PROCEDURE.

```

The variables gcFields, gcHandles, and ghBuffer must be defined in the **Definitions** section of the procedure, so that they are scoped to the procedure as a whole:

```

DEFINE VARIABLE gcFields AS CHARACTER NO-UNDO.
DEFINE VARIABLE gcHandles AS CHARACTER NO-UNDO.
DEFINE VARIABLE ghBuffer AS HANDLE NO-UNDO.

```

1.8.2 Coding the rowDisplay procedure

Next you need to code the **rowDisplay** procedure where your custom logic is written. This corresponds to the rowDisplay procedure in the Viewer custom super procedure. The name of this procedure is significant because, as described earlier, the standard Browser code automatically defines this internal procedure as the event procedure for the ROW-DISPLAY event when the ScrollRemote property is set on.

There is a small amount of standard Browser code for rowDisplay, so it must first RUN SUPER. Then it can contain exactly the same statements as a corresponding Viewer. You will create equivalent support functions to make this work properly:

```
PROCEDURE rowDisplay:
/*-----
  Purpose:   This procedure holds any custom client-side display logic
              for the Browser.
  Parameters: <none>
  Notes:
-----*/

RUN SUPER.

IF DECIMAL (widgetValue('CreditLimit')) - DECIMAL (widgetValue('Balance'))
    LT 5000 THEN
    highlightWidget('Balance').

END PROCEDURE.
```

1.8.3 Creating the other support functions

Now create the support functions to make this work.

The widgetHandle function can be the same as in the Viewer example, because it uses the same gcFields and gcHandles variables, even though those are derived from different object properties.

The widgetValue function, however, looks a little different because, as noted, you cannot reference the cell values of the browse cells from within the ROW-DISPLAY event. For this reason the code refers back to the SDO's RowObject record buffer, captured from the QueryRowObject property, to get the field value from there:

```
FUNCTION widgetValue RETURNS CHARACTER
( pcColumn AS CHARACTER ) :
/*-----
  Purpose:   Returns the value of the requested browse column
              from the RowObject buffer.
  Params:   INPUT pcColumn AS CHARACTER
  Notes:
-----*/

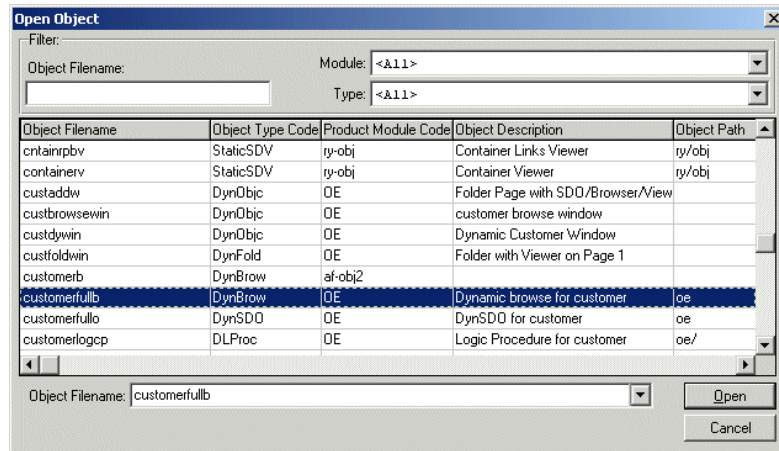
DEFINE VARIABLE hField AS HANDLE      NO-UNDO.
  hField = ghBuffer:BUFFER-FIELD(pcColumn).
  RETURN STRING(hField:BUFFER-VALUE).
END FUNCTION.
```

The `highlightWidget` function, like any other functions that just use `widgetHandle` and `widgetValue`, can be the same code as for the Viewer example.

1.8.4 Defining the custom super procedure for a browser

To make this new procedure a custom super procedure for a Customer Browser:

- 1 ♦ In the AppBuilder, use the **Open Object** dialog box to open the Browser property sheet:



- 2 ♦ Set the custom super procedure in the AppBuilder's property sheet for the Browser.

1.8.5 Changing browser attributes for the master and instance

To see the **Credit Limit** and **Balance** columns more easily, rearrange the column order in the **Selected Fields** list to move them to the front of the column list. Launch your dynamic window containing the Customer Browser to see the effect of your custom super procedure logic:

Cust Num	Name	Credit Limit	Balance	Address
1	Lift Tours Inc.	66,700	903.64	276 North Driveway
2	Uipon Frisbee	27,600	437.63	Raitapolku 333
3	Hoops	75,000	1,199.95	Suite 415
4	Go Fishing Ltd	15,000	14,235.14	Unit 2
5	Match Point Tennis	11,000	0.00	66 Homer Pl
6	Fanatical Athletes	38,900	1,202.66	20 Bicep Bridge Road
7	Aerobics valine Ky	13,500	1,112.44	Peltolantie 3
8	Game Set Match	15,000	8,254.00	Box 60
9	Pihlputaan Pyora	29,900	1,242.14	Puitkontie 2
10	Just Joggers Limited	22,000	1,222.11	Fairwind Trading Est
11	Keitalu ja Biliardi	10,900	1,186.80	Valturiemenkuja 8 A

1.9 Using the client logic API

The client logic API is a collection of functions (extending the visual class) and a programming standard that support easier development of client-side user interface logic in containers, browses, viewers, and other visual objects.

The client logic API essentially supports widget manipulation in containers, browses, viewers, and visual objects in both static and dynamic objects. For example, you might want to enable some fields in a dynamic viewer based upon the value of another field. The API, in particular, makes it much easier to associate code with one browse or viewer and have that code manipulate widgets in a different browse or viewer (as long as both are in the same container).

You can use the client logic API in static object code or in custom super procedures for dynamic objects.

You need to write client logic for each rendering engine separately. An application for both the GUI and DHTML clients must maintain two versions of client logic, one written in 4GL for GUI rendering and another written in JavaScript for Web rendering. See the [Progress Dynamics Web Development Guide](#) for information on DHTML client development.

1.9.1 Basic rules

The API uses **widget** to identify an object that can be accessed or modified using the API. These objects include any object with a widget handle in the Progress environment, plus SmartDataFields (which use the procedure handle).

NOTE: The client logic API does not support toolbar button manipulation.

The functions of the client logic API are, for the most part, intended for use in comparisons and assignments. For example:

```
IF widgetIsModified("customerviewv.discount":U) THEN
    enableWidget("customerviewv.terms":U).

tempDiscount = INTEGER(widgetValue("discount":U)) * 1.10.
```

Restricting your use of the API to comparisons and assigns will assure you of the best results for migrations to future versions of Progress Dynamics.

1.9.2 Logic

The client logic API supports code written in the following locations:

- Static object code.
- Custom super procedure logic associated with containers, browses, and viewers.

For dynamic objects, while you can use the API in any function or internal procedure of the custom super procedure, the following are usually the best places to put client logic code:

- rowDisplay internal procedures.
- Widget event procedures.
- ADM override procedures and functions.

NOTE: You cannot use the client logic API in the main block of the custom super procedure.

rowDisplay internal procedures

For your custom super procedures, take advantage of this procedure and place appropriate client logic code inside. This technique is parallel to the validation procedures of the SDO, in that it provides a hook where you can put code and benefit from the supplied standard behavior.

Unless noted in the description of the function, the API only supports actions within a browse rowDisplay internal procedure that are supported by the 4GL for the browse ROW-DISPLAY event.

Widget event procedures

The standard name for UI event procedures is <widget><event>. For example, CreditLimitLeave, indicates ON LEAVE OF CreditLimit.

The Migration Utility converts existing embedded trigger code blocks into super procedure internal procedures with this naming convention, so this is the recommended standard for new UI event procedures (the Event Action).

Note that you first need to define your own UI events using the Dynamics Repository Maintenance Tool or the dynamic property sheet.

ADM overrides

Use the API in ADM overrides as you would use any of the APIs described in the ADM documentation. See the ADM documentation for more about overrides.

Error handling

The client API does not raise errors, since errors at the client level are not desirable. If Dynamics cannot locate a referenced object, or it does not support the specified operation, then the function returns FALSE or unknown if the function has a character or handle return. Dynamics raises no other visible error condition.

TARGET-PROCEDURE

When using the client API from a super procedure, always remember to invoke internal procedures and functions IN TARGET-PROCEDURE from a super procedure.

1.9.3 Object qualification

In the API signatures, *name* is the widget name or SmartDataField name of the object you want to manipulate. In some methods, you can supply a *namelist*, which is a CHARACTER string of one or more widget or SmartDataField names. Separate multiple names with commas without intervening spaces.

Client logic code on the GUI client supports both qualified and unqualified names, as described in the next sections.

Unqualified names

When you supply just the widget names, these are considered unqualified names. The APIs have the ability to resolve unqualified names where there is no ambiguity. Using unqualified names speeds code writing and promotes reuse of code. For example, you can move code with unqualified names from one object to a similar object without changing it.

In the GUI client, unqualified widgets are qualified by a search algorithm based on where the code containing the unqualified reference is written.

The following table describes how each type of object finds objects with unqualified instance names:

Object type	Qualification method	Unqualified examples
Container	Does not support unqualified names.	NA
Browse	Checks for object in the browse first. Then checks for widgets (data fields only) in viewers with the same data source. Dynamics uses the first widget found.	ordernum orderdate
Viewer	Checks for a widget in that object first (data fields and local fields). Then checks for widgets (data fields only) in sibling viewers (viewers linked to the viewers data source). Dynamics uses the first widget found.	orderfullo.ordernum orderlinefullo.ordernum

The search algorithm is complicated a bit by references to SBO fields. Suppose a viewer displays fields from an SBO that draws data from several SDOs. To fully qualify which data source (SDO) fields are sources for viewer fields, the field is qualified by the SDO name: SDOname.fieldname. For example, orderfullo.ordernum and orderlinefullo.ordernum.

So, to deal effectively with this use case, first, the API treats names with a single qualifier as a field name qualified with an SDO name. If it fails to find the named widget, it then treats the first qualifier as an SDO name qualifier.

In an SBO-based viewer, an unqualified widgetname will find a local widget with a matching name before looking in the SBO for a qualified field. The search to locate the widget in data-source siblings is done after checking whether the unqualified field matches an SBO field. The unqualified search will return a handle even if the request is ambiguous and more than one field matches the unqualified name, so the qualifier should be used to guarantee precision. The qualified search is also faster.

Qualified names

Use qualified names when writing complex code or when the rules for resolving unqualified names does not meet the needs of your application.

Object names can be qualified by one of the methods described in the table below. They are searched depending upon the location of the client logic. If the first entry in a list of objects is qualified and others in the list are not, that first qualifier extends as a default to other unqualified items in the list:

Qualification	Description	Examples
With the Self keyword.	For a browse or viewer, Dynamics checks for a widget (data fields and local fields) in that object only. It uses everything after <code>self.</code> as the field name. In the second example, <code>orderfullo.orderdate</code> is the field name. For a container, there is no support for the self qualifier.	<code>Self.custnum</code> <code>Self.orderfullo.orderdate</code>
With the Browse keyword.	For a browse, Dynamics checks for a widget in that object only. For a viewer, Dynamics gets the viewer's data source and checks for a widget in its browse data target. For a container, there is no support for the browse qualifier. NOTE: BROWSE is an allowed 4GL variable. You cannot use a widget name of "Browse" with the client logic API.	<code>Browse.contact</code> <code>Browse.empnum</code>
With an SDO name and Browse keyword.	For a browse or viewer, Dynamics finds the SDO in the object's container source and checks for a widget in the SDO's browse data target. For a container, Dynamics checks for a widget in the SDO's browse data target.	<code>benefitsfullo.Browse.empnum</code> <code>customerfulllo.Browse.name</code>
With an SDO name.	For a browse or viewer, Dynamics finds the SDO in the object's container source and checks for a widget in the SDO's data targets, but checks the update source first. The first widget found is used. For a container, Dynamics checks for a widget in the SDO's data targets but checks the update source first. Dynamics uses the first widget found.	<code>salesrepfullo.region</code> <code>customerfullo.discount</code>

Qualification	Description	Examples
<p>With an SBO name and Browse keyword.</p>	<p>Any use of a field name from an SBO must be qualified with an SDO name: <code>SBOname.SDOname.fieldname</code>.</p> <p>In a browse or viewer, Dynamics finds the SBO in the object container source and checks for a widget SBO's browse data target</p> <p>In a container, Dynamics checks for a widget in the SBO's browse data target.</p>	<p><code>ordersbo.Browse.orderfullo.orderdate</code></p> <p><code>employeesbo.Browse.employeefullo.name</code></p>
<p>With an instance name.</p>	<p>The instance name needs to be a browse, viewer, or visual object that supports the <code>internalWidgetHandle</code> function.</p> <p>It uses everything after the <code>instancename</code>. as the field name. In the second example, <code>orderfullo.shipdate</code> is the field name.</p> <p>For a browse or viewer, Dynamics gets the object's container source and checks for a widget in that object instance within its container.</p> <p>For a container, Dynamics checks for a widget in that object instance of the container.</p>	<p><code>customerviewv.name</code></p> <p><code>orderviewv.orderfullo.shipdate</code></p>
<p>With an SBO name.</p>	<p>Any use of a field name from an SBO must be first qualified with an SDO name: <code>SBOname.SDOname.fieldname</code>.</p> <p>If the viewer qualifies the fields with an SDO name (because they have been defined for an SBO), Dynamics uses the SDO name in the search, otherwise it uses only the field name.</p> <p>In a browse or viewer, Dynamics finds the SBO in the object container source and checks for a widget in the SBO's data targets that map to the SDO, but checks the update source first. The first widget found is used.</p> <p>In a container, Dynamics checks for a widget in the SBO's data targets that map to the SDO, but checks the update source first. The first widget found is used.</p>	<p><code>ordersbo.itemfullo.itemnum</code></p> <p><code>employeesbo.benefitsfullo.ext</code></p>

1.9.4 Client logic functions

Table 1–1 summarizes the functions and procedures that make up the client API. For complete documentation, see the *Progress Dynamics ADM2 API Reference*.

Table 1–1: Client logic functions (1 of 6)

Function	Description
assignFocusedWidget	Sets focus to the named object. Not supported for SmartDataFields and returns FALSE if attempted for SmartDataFields.
assignWidgetValue	Takes the name of one object and a character screen value as input and sets the SCREEN-VALUE of the object. The DataValue is set for a SmartDataField. (This would always be the key field for a lookup even where the display field is different.) This function sets DataModified to make the toolbar enable saving data (it behaves as if the user actually changed the field value manually). DataModified is set to TRUE whether or not the field is enabled.
assignWidgetValueList	<p>Takes the name of one or more objects and character screen values and a delimiter as input and sets the SCREEN-VALUE of the objects. The DataValue is set for a SmartDataField (this would always be the key field for a lookup even where the display field is different).</p> <p>This function sets the DataModified attribute to force the toolbar to enable saving data (it behaves as if the user actually changed the field value manually). The DataModified attribute is set to TRUE whether the field is enabled or not.</p>

Table 1–1: Client logic functions*(2 of 6)*

Function	Description
blankWidget	<p>Blanks the SCREEN-VALUE of the objects in the namelist. The DataValue is blanked for a SmartDataField (this would always be the key field for a lookup even where the display field is different).</p> <p>This function sets the DataModified attribute to force the toolbar to enable saving data (it behaves as if the user actually changed the field value manually).</p> <p>This does nothing to objects that do not support SCREEN-VALUE and to objects where a blank screen value does not make sense, such as toggle boxes. It blanks a combo-box by setting its list items to null.</p>
clearWidget	Clears any value currently in the widget. Currently, if a Save is attempted on a widget before a new value is provided, the widget will revert to its last saved value.
disableRadioButton	Disables the specified radio button of the radio set objects identified in the namelist. Returns FALSE if a widget in the list is not a radioset, if a widget in the list is not found, or if the button number is invalid.
disableWidget	<p>Disables the objects identified in the namelist.</p> <p>For SmartDataFields, it runs the disableField function. Note that disableField is not a generic SmartDataField function, therefore this API is only supported for SmartDataFields that have disableField defined. If a field in the list is not found, or a SmartDataField without disableField is included in the list, disableWidget returns FALSE.</p>
enableRadioButton	Enables the specified radio button of the radio set objects identified in the name list. Returns FALSE if a widget in the list is not a radioset, if a widget in the list is not found, or if the button number is invalid.

Table 1–1: Client logic functions*(3 of 6)*

Function	Description
enableWidget	Enables the objects identified in the name list. For SmartDataFields, it runs the enableField function. Note that enableField is not a generic SmartDataField function, therefore this API is only supported for SmartDataFields that have enableField defined. If a field in the list is not found, or a SmartDataField without enableField is included in the list, enableWidget returns FALSE.
formattedWidgetValue	Returns the SCREEN-VALUE of the object, or in the case of a browse column when in a ROW-DISPLAY trigger, the STRING-VALUE from the RowObject buffer field. The DataValue is returned for a SmartDataField (this would always be the key field for a lookup even where the display field is different). For example, you could use this function to get the formatted value of a single field to use for comparisons.
formattedWidgetValueList	<p>Takes the name of one or more objects and returns the SCREEN-VALUE of the object or objects. For example, use this function to retrieve the formatted values of several fields to assign their screen values to other fields.</p> <p>If the object is a browse column (called from within a ROW-DISPLAY trigger), the STRING-VALUE from the RowObject buffer field is added to the list of returned values.</p> <p>If the object is a SmartDataField, the DataValue field is returned. The DataValue field is always the key field, even when the display field is different.</p> <p>If a field in the list is not found or a field has an unknown value, the function returns “?” for its value in the returned character list.</p>
hideWidget	Hides the objects identified in the namelist (and their popup buttons). For SmartDataFields, it invokes the hideObject method.

Table 1–1: Client logic functions*(4 of 6)*

Function	Description
highlightWidget	Sets the background and foreground colors (FGCOLOR and BGCOLOR widget attributes) of the named objects to a standard highlight color, depending on the value of the highlightType argument. For example, you might want to change the background color of a field with an invalid value to red.
resetWidgetValue	Resets the SCREEN-VALUE of the objects identified in the name list back to its original value from its data source. If a field in the list is not found or there is no data source for a field in the list, FALSE is returned and it will process all others in the list.
toggleWidget	<p>Reverses the value of one or more objects of type LOGICAL in the name list. The format of the widget is used to reverse its SCREEN-VALUE. For example, a logical with a format of credit/debit would change a “credit” screen-value to “debit”. A null value is not changed and FALSE is returned.</p> <p>This function sets the DataModified attribute to force the toolbar to enable saving data (it behaves as if the user actually changed the field value manually).</p>
viewWidget	Views the object or objects identified in the namelist (and their pop-up buttons). For SmartDataFields, it invokes the viewObject method.
widgetHandle	Returns the handle of the requested object. For a basic object it returns the widget-handle. For a SmartDataField it returns the procedure handle.
widgetIsBlank	Returns TRUE if the widget is blank, otherwise FALSE. If the namelist contains more than one object, then the function returns TRUE if all of them are blank, otherwise FALSE.
widgetIsFocused	Returns TRUE if the widget has focus. This is not supported for SmartDataFields. This returns unknown if the field is not found or if the widget is a SmartDataField.

Table 1–1: Client logic functions

(5 of 6)

Function	Description
widgetIsModified	<p>Returns TRUE if the MODIFIED attribute or its equivalent is set for the object, otherwise FALSE. If the name list contains more than one object, then the function returns TRUE if any of them have changed, otherwise FALSE. If any field in the name list is not found, unknown is returned.</p> <p>For example, use this function to check if any of multiple values involved in a calculation or expression have been modified.</p>
widgetIsTrue	<p>Returns TRUE if the value of a LOGICAL object is TRUE, otherwise FALSE. This function does not support SmartDataFields. This function returns unknown if the field is not found, if the value of the LOGICAL is unknown, if the widget is not a LOGICAL field, or if the widget is a SmartDataField. Contrast with the widgetValue function, which returns a CHARACTER value.</p>
widgetLongCharValue	<p>Returns the longchar value of a field. The widgetValue function cannot return longcar values.</p>

Table 1–1: Client logic functions*(6 of 6)*

Function	Description
widgetValue	For most objects, returns the INPUT-VALUE of the object. For a browse column within a ROW-DISPLAY trigger, the function returns the BUFFER-VALUE from the RowObject buffer field. If INPUT-VALUE returns a 4GL error because the value is actually blank, widgetIsBlank will be invoked and blank will be returned. For SmartDataFields, which do not have a standard function for returning an unformatted value, this function does nothing and returns unknown.
widgetValueList	Takes the name of one or more objects and a delimiter and returns the INPUT-VALUE of the object. In the case of a browse column reference from within a ROW-DISPLAY event, it returns the BUFFER-VALUE from the RowObject buffer field. SmartDataFields do not have a standard function for returning an unformatted value so this function does nothing for SmartDataFields and returns unknown. If a field in the list is not found or a field has an unknown value, the function returns “?” for its value in the returned character list.

1.10 Organizing logic at the level of procedures or functions

As a general rule when writing your client-side logic, whether it is invoked from the rowDisplay procedure or from a UI Event Action procedure or both, it is advisable to keep it as modular as possible, so that a single logical action is coded as a single internal procedure or function. This makes it simpler to invoke the same logic from both the Display event and a particular UI Event. In addition, it makes it more straightforward to convert logic as needed to another form, such as JavaScript for a Web interface, and also facilitates the conversion of client code to rules data at a later time. For example, if the code must invoke a specialized block of code on some event, for example to check a credit card number, then if you write just the credit card checking code as a distinct function or procedure, it will be simpler to separate that code, which you must either write by hand for the Web or replace with a standard Web routine to do the same thing, while preserving the standard client API code for the rest of the logic. At a later date, then, it will be more realistic for a migration tool to convert the standard API logic to data, while leaving in calls to specific additional functions you have written.

1.11 Customizing dynamic lookup behavior in viewers

The dynamic Lookup supports three custom event “hooks” designed to allow developers to extend the behavior of the Lookup in various ways. This section describes three named events that the Lookup support code publishes, and which custom client code can subscribe to in order to get behavior that goes beyond what the standard Lookup provides. You can learn more about the dynamic Lookup and the many features of its property sheet in the Developer’s Guide.

1.11.1 LookupEntry event

On entry of the lookup field, the enterLookup procedure in the Lookup support procedure `lookup.p` publishes `lookupEntry` with the following parameters:

DEFINE INPUT PARAMETER	pcScreenValue	AS CHARACTER	NO-UNDO.
DEFINE INPUT PARAMETER	phLookup	AS HANDLE	NO-UNDO.

A description of the parameters is as follows:

- **pcScreenValue** — This is the dynamic Lookup’s current screen value
- **phLookup** — This is the handle of the dynamic Lookup itself (the SmartDataField™ instance)

The handle of the Lookup instance is useful when your viewer contains multiple Lookups and you need to determine which Lookup caused your hook to fire.

To use this event, add a procedure to your Viewer’s custom super procedure with the name **lookupEntry** and the above parameters.

Using PUBLISH and SUBSCRIBE properly in super procedures

In **initializeObject** in the super procedure supporting the Lookup, before the RUN SUPER statement, subscribe your Viewer to the event using the statement:

SUBSCRIBE PROCEDURE	TARGET-PROCEDURE	TO "lookupEntry":U	IN TARGET-PROCEDURE.
---------------------	------------------	--------------------	----------------------

Note the format of this SUBSCRIBE statement carefully. It is essential that you always keep in mind that events should almost never be published by or directly subscribed to in super procedures. The super procedure acts in the background on behalf of application objects such as Viewers. Thus a SUBSCRIBE statement in a super procedure will normally be qualified by PROCEDURE TARGET-PROCEDURE, meaning that the subscription is registered on behalf of the Viewer or other object, not the super procedure itself. And the event will also be qualified by IN TARGET-PROCEDURE, meaning that the interpreter will respond when the event occurs in the Viewer or other object, not in the super procedure. Likewise, a PUBLISH statement in super procedure code should normally be of the form:

```
PUBLISH <event-name> FROM TARGET-PROCEDURE.
```

In this way the interpreter responds to the event as if it had actually come from the supported SmartObject, and not from the super procedure itself. Forgetting these forms can cause frustrating problems, when events seem not to occur or seem not to be responded to properly.

The lookupEntry event is used within the Lookup super procedure lookup.p itself to save the current screen value of the Lookup so that on leave of the Lookup you can see if the Lookup has been programmatically changed and allow the standard Lookup code to fire, validating the new value. Possible uses in application-specific code could be for programmatic manipulation of properties at run time, or possibly customization of the value based on the value of other fields on your viewer.

1.11.2 LookupComplete event

This event occurs on leave of the Lookup (published by the procedure leaveLookup in lookup.p and also on return from the selection of a row in the Lookup browse in the procedure rowSelected in lookup.p. LookupComplete has the following parameters:

DEFINE INPUT PARAMETER	pcFieldNames	AS CHARACTER	NO-UNDO.
DEFINE INPUT PARAMETER	pcFieldValues	AS CHARACTER	NO-UNDO.
DEFINE INPUT PARAMETER	pcKeyFieldValue	AS CHARACTER	NO-UNDO.
DEFINE INPUT PARAMETER	pcNewScreenValue	AS CHARACTER	NO-UNDO.
DEFINE INPUT PARAMETER	pcOldScreenValue	AS CHARACTER	NO-UNDO.
DEFINE INPUT PARAMETER	plBrowseUsed	AS LOGICAL	NO-UNDO.
DEFINE INPUT PARAMETER	phLookup	AS HANDLE	NO-UNDO.

Chapter 1, “LookupDisplayCompleteEvent parameters,” describes the lookupCompleteEvent parameters.

Table 1–2: LookupCompleteEvent parameters

Parameter	Description
pcFieldNames	A comma-delimited list of field names (in the form table.fieldname) which includes the key field, the displayed field, all the selected browse fields and the selected linked fields, each field appearing only once in the list.
pcFieldValues	A CHR(1)-delimited list of the corresponding values for the fields in pcFieldNames.
pcKeyFieldValue	The key field value of the selected record .
pcNewScreenValue	The currently displayed field SCREEN-VALUE
pcOldScreenValue	The previously displayed field SCREEN-VALUE before the Lookup was done, which you use to check if the value changed.
plBrowseUsed	This logical parameter equals YES if the new record was selected from the Lookup browse, or NO if the new record was selected as a result of manually entering a value without using the Lookup browse.
phLookup	The handle of the Lookup SmartDataField instance, used to determine which Lookup on your Viewer caused the event to occur (if your Viewer contains multiple Lookups).

To use this event, simply add a procedure to your Viewer's custom super procedure with the name **lookupComplete** and the above parameters. Then in **initializeObject** of your custom super procedure, before the RUN SUPER statement, add the following line:

```
SUBSCRIBE PROCEDURE TARGET-PROCEDURE TO "lookupComplete":U
  IN TARGET-PROCEDURE.
```

This hook is useful if you need to control the updating of related fields to the Lookup field whenever a new value is selected. Remember that the linked fields and widgets instance properties of the Lookup can be used to automatically update related fields, so this event is only really for exceptional circumstances where perhaps specific formatting is required of the related fields. It might also be necessary to pass new or linked values on to other viewers, in which case this event could also be useful.

1.11.3 LookupDisplayComplete event

This event occurs as soon as the Lookup has displayed its values into the Lookup's fill-in and linked widgets. The difference between the lookupComplete and lookupDisplayComplete hooks is mainly that the lookupDisplayComplete hook fires *every time* the lookup displays something. This includes when a viewer initializes.

The lookupDisplayComplete event is published from displayLookup in `lookup.p` with the following parameters:

```
DEFINE INPUT PARAMETER pcFieldNames      AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER pcFieldValues     AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER pcKeyFieldValue   AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER phLookup          AS HANDLE      NO-UNDO.
```

Table 1–3 describes the lookupDisplayCompleteEvent parameters.

Table 1–3: **LookupDisplayCompleteEvent parameters**

Parameter	Description
pcFieldNames	A comma-delimited list of field names (in the form table.fieldname) which includes the key field, the displayed field, all the selected browse fields and the selected linked fields, each field appearing only once in the list
pcFieldValues	A CHR(1)-delimited list of the corresponding values for the pcFieldNames.
pcKeyFieldValue	The key field value of the selected record .
phLookup	The handle of the Lookup SmartDataField instance, and is used to determine which Lookup on your Viewer caused the event to occur (in the case where your Viewer contains multiple Lookups).

To use this event, simply add a procedure to your Viewer’s custom super procedure with the name **lookupDisplayComplete** and the above parameters. Then in **initializeObject** of your super procedure, before the RUN SUPER statement, add the following line:

```
SUBSCRIBE PROCEDURE TARGET-PROCEDURE TO "lookupDisplayComplete":U
IN TARGET-PROCEDURE.
```

This event is useful if you want to use the values in the linked fields when the Viewer starts up. The lookupComplete hook would not be available at this time, as the Lookups have not actually been used yet. It might also be necessary to pass new or linked values on to other Viewers, in which case this event would also be useful.

Extending Object Classes

The Progress Dynamics™ framework provides you with a number of Objects and Object types (classes) you can use to build applications. Each of these has its own attributes or properties, and its own behavior. These objects are organized into a class hierarchy, and objects of a given class inherit attributes and behavior from all the classes above them in the hierarchy.

You can extend this class hierarchy to create custom objects of your own and create new attributes and associate them with your newly created classes. You can either extend off the bottom of the class hierarchy (by extending a class at the bottom of the hierarchy tree) or in the middle of the class hierarchy. This chapter provides a prescribed approach for performing these tasks and discusses the pros and cons of both approaches. It includes these topics:

- [Introduction](#)
- [Extending the behavior of standard objects](#)
- [Defining new attributes and default values](#)
- [Extending the class hierarchy](#)
- [Changing the object type of a set of objects](#)
- [Using and extending dynamic basic objects](#)
- [Extending object classes tutorial](#)
- [More complex customization](#)

2.1 Introduction

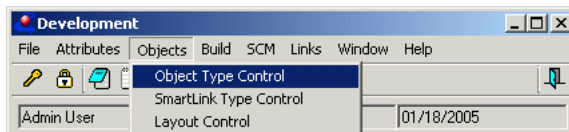
In this chapter, the term *Object* generally refers to Progress® SmartObjects™ (both static and dynamic) and other procedure-based objects or dynamic versions of basic objects, such as fill-in fields and buttons. The term *Object type* refers to a *Class*. Object type and Class are used interchangeably in this chapter.

The first part of the chapter describes the class structure of Progress Dynamics. At the end of the chapter, a step-by-step guide takes you through the process of creating new attributes, extending the class hierarchy, specifying custom behavior for the subclass and for custom attributes, and setting up your custom classes to work with the AppBuilder.

NOTE: This chapter describes the prescribed approach supported as of Version 2.1A02 (Service Pack 2).

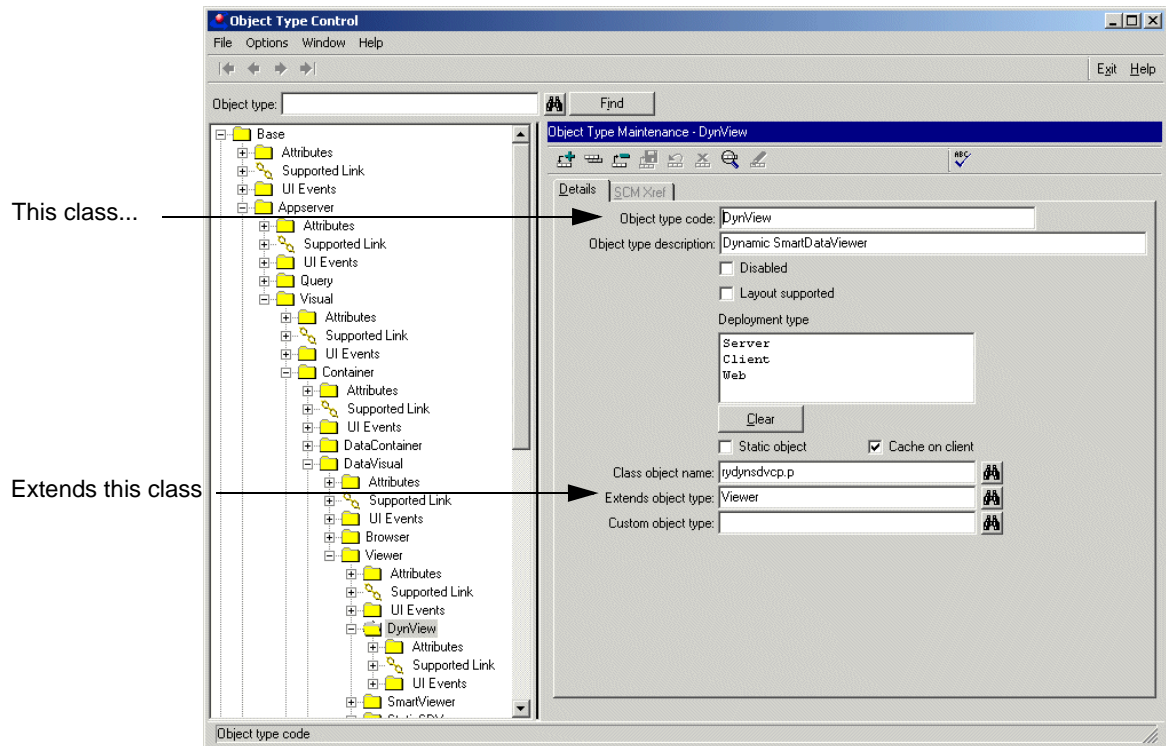
2.1.1 The Progress Dynamics object hierarchy

You can view the Progress Dynamics Object hierarchy by selecting **Objects→Object Type Control** from the **Development** menu:



There are three top-level nodes in the Object type display. Progress SmartObjects are all represented under the Base node of the treeview that is the first top-level node of the hierarchy. You can open individual nodes in this tree to drill down into the hierarchy, but if you know the Object type you want to see you can simply enter its name in the **Object Type** field and press **Find**.

This display for example shows the Object Type hierarchy for the dynamic Viewer, or the DynView class:



Classes are structured in a one-to-many relationship. The viewer above in the left-side panel shows the current class (DynView) which corresponds to the highlighted node in the treeview. The class that the DynView class extends is shown in the 'Extends object type' field (Viewer).

2.1.2 Understanding the nodes of the object hierarchy

If you are familiar with Progress SmartObjects, you already understand the basics of the Object class hierarchy. However, there are some special cases that need to be explained for you to understand all the nodes that are involved in locating an Object type.

Figure 2-1 is a partial graphical representation of the Object type hierarchy.

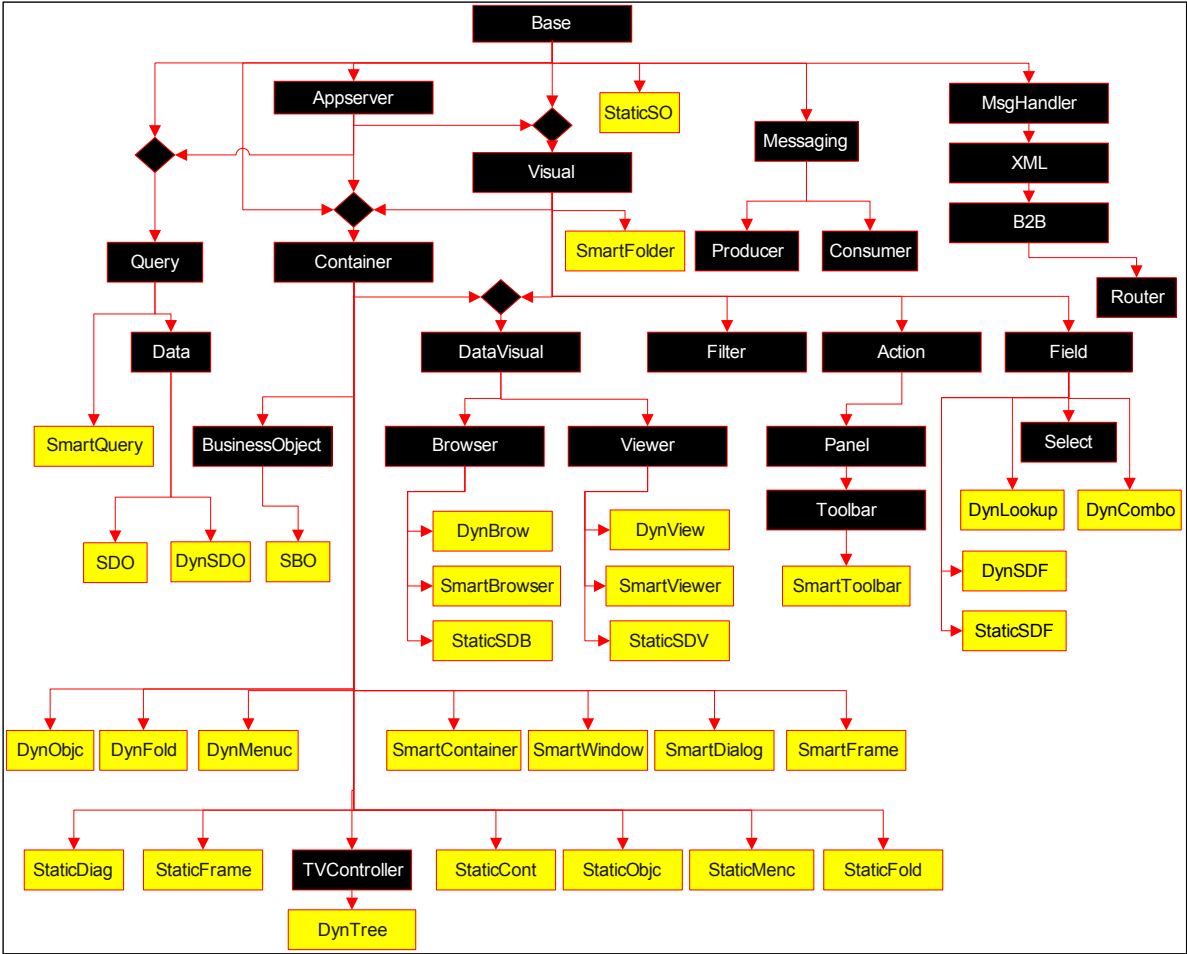
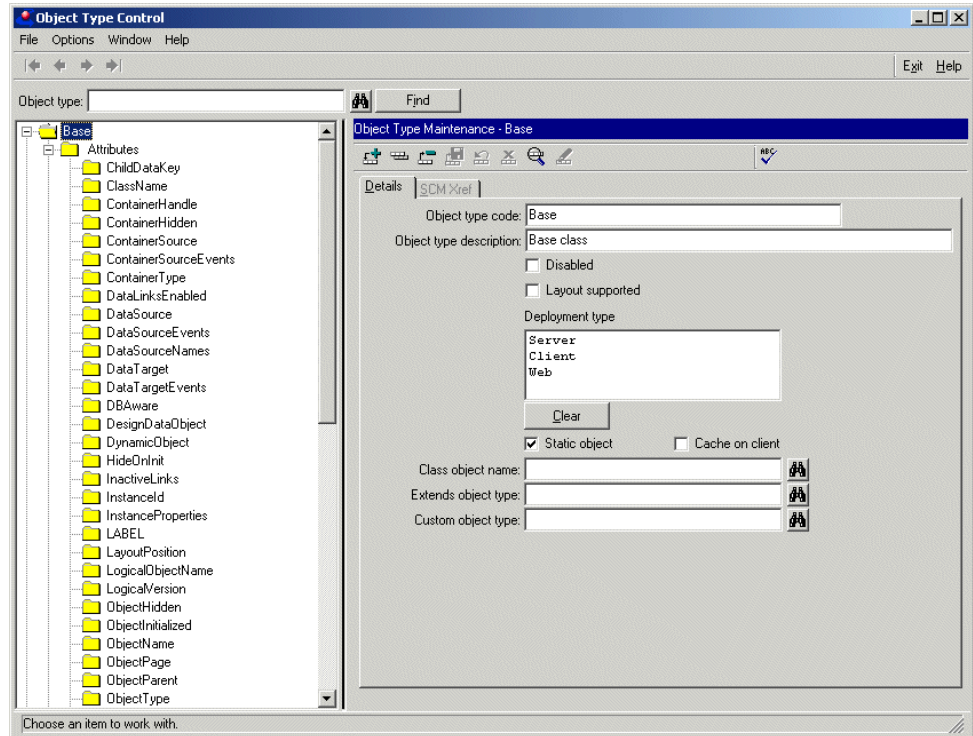


Figure 2-1: Class hierarchy

First, the Base at the top of the tree represents the ADM2 Smart class that is the basis for all SmartObjects, with the procedure smart.p and the properties or attributes defined in the include file smrtprop.i.

Under each node in the tree that represents an Object type, there is an **Attributes** node, a **Supported Links** node, and a **UI Events** node. The **Attribute** node groups all the attributes that are defined at that level of the hierarchy. If you expand this node, you can see each attribute and its default value for that class. For example, this is a partial list of all the attributes defined at the highest level of the hierarchy, the Base level:



Every Object in the tree under Base has these attributes, because they are defined at the top of the tree.

After the **Supported Links** node and the **UI Event** node, all remaining nodes represent an Object type in the hierarchy. The next level of the hierarchy is the AppServer level. Not all SmartObjects actually use the AppServer class, but because it is conditional for some types of Objects, the class is inherited here in the tree.

Underneath the AppServer class, the Query class which contains the Data class which leads to SmartDataObjects (SDOs).

The Visual class leads to all visual objects.

Beneath the Visual class is the Container class. Again, not all visual objects can be containers, but because some, like the SmartViewer, can be containers if they include SmartDataFields, the Container class is inserted here in the hierarchy.

Beyond this the class hierarchy continues as you would expect it to for the various kinds of SmartObjects.

The second major node at the top of the tree is labeled (somewhat misleading at this point) DynamicObject. Under this node you will mostly find templates for procedural objects.

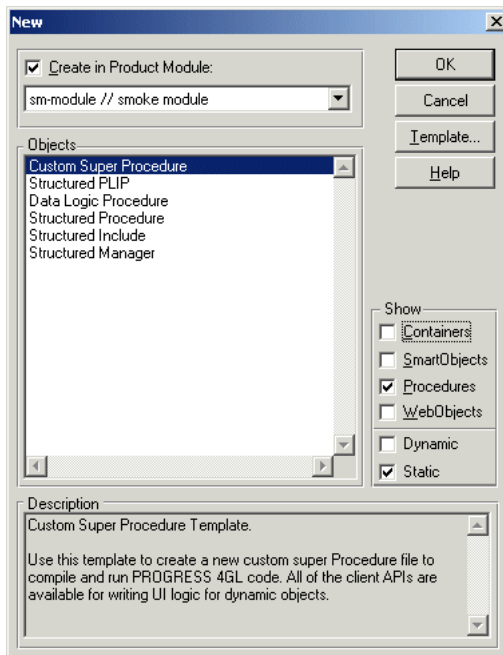
The third major node is labeled **ProgressWidget**. Under this node are all the dynamic objects that represent basic objects such as fill-ins, buttons, and rectangles. These are discussed later.

2.2 Extending the behavior of standard objects

If you want to extend the behavior of a class of Objects by adding executable code for its standard events, you can do that without making any changes to the Object type hierarchy. There are basically two levels (at an object level and a class level) at which you can do this, both of which are summarized in the following sections.

2.2.1 Extending the behavior of a single object

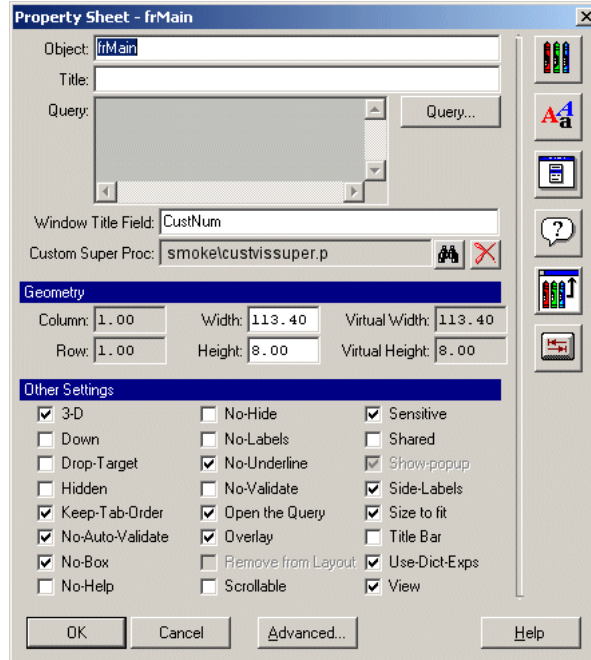
If you simply want to extend the behavior of a single Object, such as a dynamic SmartViewer for the Customer table, then you should create a new Progress procedure starting with the Custom Super Procedure template in the AppBuilder:



You can define local versions of any of the standard event procedures for the Object, or you can define client side logic such as UI behavior.

Ensure that you register the procedure upon saving it. If it is already created but not registered in the repository, use the **File→Register in Repository** menu item in the AppBuilder.

To add your custom procedure to the procedure stack of the Object, go into the Object's property sheet and select the custom super procedure in the lookup labeled for that use, such as this example from the Viewer property sheet:



This custom procedure is saved in an attribute called SuperProcedure which is stored against the object in the Repository.

The custom super procedure is then started for you at runtime and added to the bottom of the super procedure stack for just this one Viewer, so that its custom code is executed before any of the standard event procedure code.

2.2.2 Extending the behavior of all objects in a class

The second level at which you can extend behavior is for all Objects in a class. There are two different ways of modifying class behavior and each has different implications. The first is extending the class at the bottom of the class hierarchy, also known as subclassing (or extending off the bottom). The second is extending the behavior of a class anywhere in the class hierarchy.

Extending a class at the bottom implies creating your own class that extends an existing Progress Dynamics class that is a bottom level node. The DynView class is one such class. You could create your own class that might extend this class and have specific custom attributes associated with it. You could then create your own custom dynamic viewers of this new class that would inherit the behavior of the DynView class as well as your own custom behavior and attributes. Only your custom dynamic viewers would inherit your custom behavior. As mentioned above, this is known as subclassing.

Extending a class in the middle implies creating your own class, or set of classes, that extend an existing class in the middle of other Progress Dynamics supplied classes. For example, if you want to apply common behavior to all dynamic and static viewers, you would have to extend the Viewer class. This is done by adding a custom class to the Viewer class.

2.2.3 Extending the behavior of some objects in a class

If you want to apply behavior to a specific set of objects in a class that is defined at the bottom of the hierarchy tree as shipped with Progress Dynamics, you can simply create your own class and extend the Progress Dynamics class. For example, to extend the DynView class, you could create a class called myDynView and use the Object Type Control tool to specify this extension. Ensure you provide a unique name for all classes and Objects created (See the [“Providing distinctive names for attributes and other objects”](#) section.) The process is described in the bullets below:

- Create a super procedure starting with the Custom Super Procedure template in the AppBuilder and ensure it is registered in the repository. You can define local versions of any of the standard ADM2 procedures or you can define client side logic that would apply to all dynamic viewers.
- In the Object Type Control tool, find the DynView class. Create a new class that extends the viewer class and add the attribute SuperProcedure to that class and specify the relative path of the procedure you created in the step above. You can also create attributes and apply those attributes against the custom class.

The advantage of this type of extension is that it will only apply to specific objects that you create of the extended object type, that is, the change is focused to a specific set of objects and for existing objects to inherit the new behavior, they would need to have their object type modified to use the extended object type. The standard objects shipped with the Progress Dynamics framework would therefore be unaffected by this type of extension.

A possible issue is that if the framework introduces a new class in a future version that extends one of the classes you extended, your customizations will not apply to these new classes. For example, say you extended the DynViewer Class with myDynViewer Class and created objects against that class. If Progress Software extends the DynViewer class in a future version and introduces a new class (that is, a dynamic filter viewer), then your class would not inherit the behavior of the new introduced class. This may or may not be what you want. If it is not, then you need to manually modify your extended class to extend from the new class (using the Object Type Control tool).

2.2.4 Extending object behavior in the class hierarchy

If you want to apply behavior to a subset of classes, you can extend a class in the middle of the hierarchy so that all classes inheriting from that class will inherit your custom behavior and custom attributes. To accomplish this, you can modify the class hierarchy using the Object Type Control tool. If you need to extend or customize behavior for static objects, you may still need to perform some ADM customization.

It is quite simple to extend behavior to all data visual objects (browsers, viewers):

- Create a super procedure starting with the Custom Super Procedure template in the AppBuilder and ensure it is registered in the repository. You can define local versions of any of the standard ADM2 procedures or you can define client side logic that would apply to all viewers and browsers.
- In the Object Type Control tool, find the DataVisual class. Create a new class that extends the DataVisual class and add the attribute SuperProcedure to that class and specify the relative path of the procedure you created in the step above.
- Now add the new class to the DataVisual class as a custom class (the **Custom Object Type** field).

The overridden behavior will apply to dynamic viewers and browsers. It will also apply to static viewers and browsers as long as they are being run within a dynamic container. However, if you want to apply your custom behavior for viewers in a static container, you need to customize the ADM2 files as described in the [“ADM customization”](#) section.

The advantage of extending in the middle using the custom object type is that you can alter the behavior and add custom attributes to an entire subset of new or existing classes. The changes you make here will impact all objects of any classes below where you made your changes, and so will also impact the standard objects shipped with Progress Dynamics.

The method described above extends the behavior by adding a custom class to an existing Progress Dynamics classes, and is the recommended way of extending class behavior.

Note that even though the information in this section refers to customizing in the middle, the same principles and methods apply when extending the behavior of all objects in a class, even a “bottom” class such as DynView. It is possible to extend class behavior by creating a child class and inserting it between existing Progress Dynamics classes. This was the only method of extending class behavior prior to Progress Dynamics version 2.1A02. This method of changing a class behavior is discouraged because of the fact that such customizations will be overwritten when a new deployment is made from a central repository.

Modifying class behavior by adding a custom class is the recommended approach.

2.2.5 ADM customization

It is currently necessary to define ADM2 customization both through the **adm2/custom** include file and in the Progress Dynamics repository. This allows ADM2 customized procedures to run interchangeably in non-repository environments (including static containers in Progress Dynamics) and in repository driven dynamic containers. Note that this is also necessary if you are customizing data logic that runs on the server—such as business logic for SmartDataObjects (SDOs) or SmartBusinessObjects (SBOs), since these objects currently run statically on the Progress AppServer™ and on Webspeed® Transaction Server. The following steps are required:

- Start with the appropriate procedure in the `src/adm2/custom` directory, for example the procedure `viewercustom.p` for an extended Viewer class. Copy this procedure from the install directory to a local directory with the same path. (that is, `work/adm2/custom`).
- Edit this procedure and add the same kind of code to it that you would add to a custom super procedure for a single Object.
- Copy the associated include file from `src/adm2/custom` to your local directory, for example, the file `viewercustom.i` for an extended Viewer class.
- Edit this include file to remove the comments from the line of code that runs `start-super-proc` for the custom .p file, such as `viewercustom.p`.
- Recompile all the static Objects that you want to inherit the new behavior. The static objects will include the custom include file in the procedure, which in turn will cause the custom super procedure to be started at runtime and added to each Object’s stack.

This set of diagrams illustrates ADM customization, using a Viewer as the example. If you want to extend the behavior of every object of this type, then defining a super procedure for it in the `adm2/custom` directory called `viewercustom.p` does this. That super procedure is added to the end of the stack for every instance of an object of that type, as shown in [Figure 2-2](#).

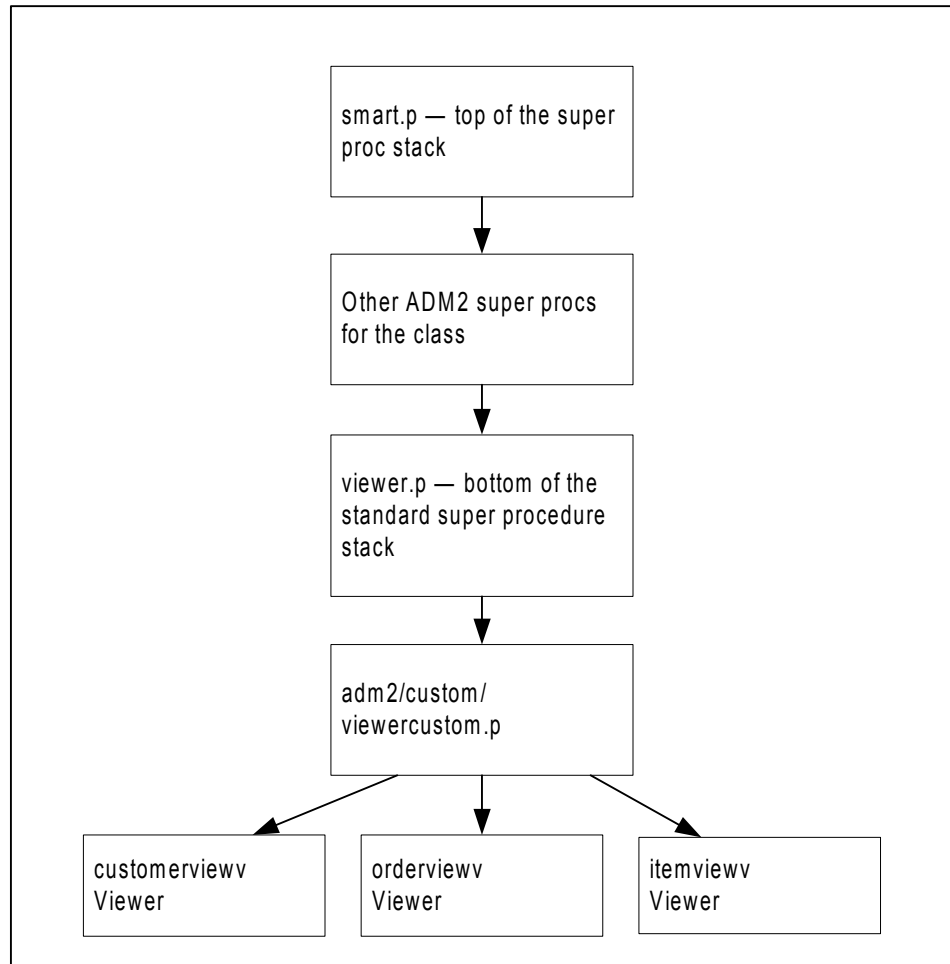


Figure 2-2: Defining a super procedure

Every Viewer you create will now inherit the behavior defined in `viewercustom.p`. Remember that you have to recompile the physical procedure that is the driver for dynamic Objects of the type. For Viewers, for example, this is the file `ry/obj/rydynviewv.w`.

An alternative is to subclass the static Viewer class for a special group of objects, so that they can have extra behavior that does not need to be added to every single object of that type. To do this, use the Add New Class utility in the AppBuilder to define the whole set of procedures that make up the object's definition, with a new object name such as MyViewer. Then you can define standard Viewer objects and also MyViewer objects, and each would in effect be a separate object type, with MyViewer inheriting from Viewer, as shown in [Figure 2–3](#).

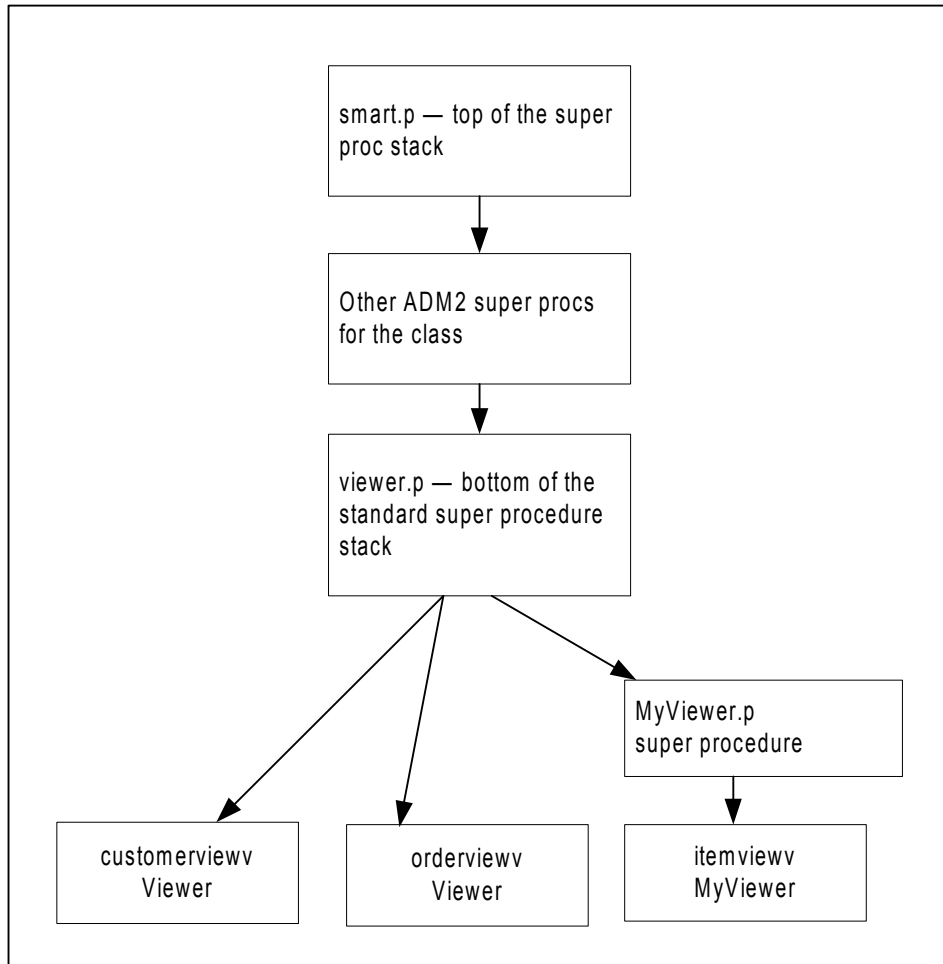


Figure 2–3: Subclassing an object type

The Customer and Order Viewers are built from the standard template or Repository object type, and the Item Viewer from a variation of that object. The Object Type Control in Progress Dynamics allows you to define new Object Types and indicate what other type they extend. The Attribute Maintenance utility allows you to define new attributes for new Object Types. You can add those new attributes to new Object Types either in the Object Type Control or in the Repository Maintenance Tool. Because this method of actually subclassing an Object Type is more the exception than the rule, it will not be explained in further detail here. In most cases, you will want to extend the behavior of either **all** your Objects of a type, as described above, or of individual Objects, as described next.

The third case is the one that will normally apply for custom super procedures. Each individual object that needs extended behavior has its own super procedure, which is added to the end of the super procedure stack for that object when it is instantiated, as shown in [Figure 2-4](#).

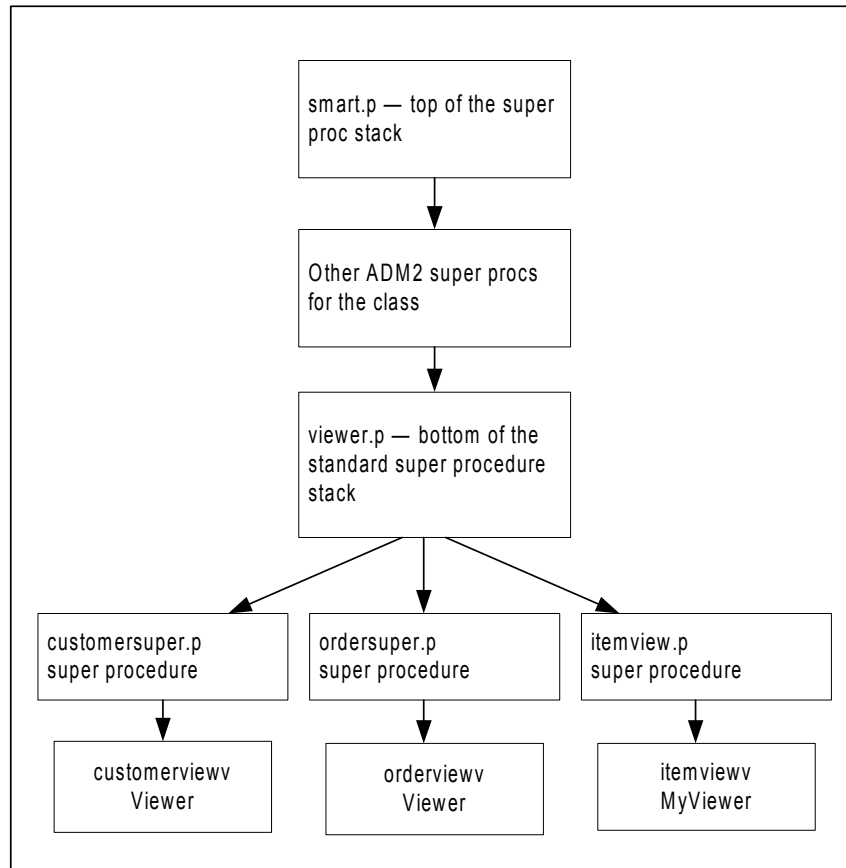


Figure 2-4: Super procedure for individual objects

2.2.6 Instantiation order of super procedures

When an object is launched, all of the super procedures associated with the object's class and its inherited classes are added to its stack. These super procedures are based on the value of the SuperProcedure attribute as defined in the repository. The order of the instantiation of these super procedures is based on the class hierarchical structure. Typically, the stack of a dynamic viewer might look something as follows:

```
ry/prc/rydynview.p
adm2/viewer.p
adm2/datavis.p
adm2/containr.p
adm2/visual.p
adm2/smart.p
```

If you were to extend a class in the middle of the hierarchy, it is important that you define your class in the repository and assign the SuperProcedure attribute. For example, if you extend the DataVisual and Viewer class, you may have customized procedures `datavisualcustom.p` and `viewercustom.p`. You would then have to create custom classes for both of these classes, extending the hierarchy accordingly and specify these super procedures in the SuperProcedure attribute. The stack could then appear as follows:

```
ry/prc/rydynview.p
adm2/custom/viewercustom.p
adm2/viewer.p
adm2/custom/dataviscustom.p
adm2/datavis.p
adm2/containr.p
adm2/visual.p
adm2/smart.p
```

CAUTION: If you intend to run the static objects in both a non-Dynamics and Dynamics environment, or if you want to run static objects in both a dynamic or static container, you need to modify the ADM code to maintain the customization, as described below.

Dynamics has modified all of the class include files to conditionally run the custom super procedures based on the value of the pre-processor ADM-LOAD-FROM-REPOSITORY. This ensures that you only run the super procedure if it hasn't already been loaded from the repository. Add the following code to accomplish this:

```
IF NOT {%ADM-LOAD-FROM-REPOSITORY} &THEN
  RUN start-super-proc ("adm2/custom/viewercustom.p":U)
```

If you do not qualify the starting of the super-procedure, the stack order will not be as indicated above. Instead, all of the custom super procedures will be added on top of the stack as follows:

```
adm2/custom/viewercustom.p
adm2/custom/dataviscustom.p
ry/prc/rydynviewp.p
adm2/viewer.p
adm2/datavis.p
adm2/containr.p
adm2/visual.p
adm2/smart.p
```

2.3 Defining new attributes and default values

The topics below describe how to add new attributes to your classes and set default values for them.

2.3.1 ADM2 attribute support

In the standard ADM2 support code, SmartObject attributes or properties (the terms are synonymous as used here) are defined in include files that are part of what is compiled into each static SmartObject. The top include file, called `src/adm2/smrtprop.i`, creates a dynamic temp-table named `ADMProps` and adds basic attributes to it as fields in that temp-table. Each temp-table field defines the name, data-type, and (if appropriate) the initial value of the attribute. Here is the beginning of that list of basic attributes used by all SmartObjects:

```
CREATE TEMP-TABLE ghADMProps.
  ghADMProps:UNDO = FALSE.
  ghADMProps:ADD-NEW-FIELD('ObjectName':U, 'CHAR':U, 0, ?, '':U).
  ghADMProps:ADD-NEW-FIELD('ObjectVersion':U, 'CHAR':U, 0, ?,
    '{&ADM-VERSION}':U).
  ghADMProps:ADD-NEW-FIELD('ObjectType':U, 'CHAR':U, 0, ?,
    '{&PROCEDURE-TYPE}':U).
...
```

Each of the include files in the Object hierarchy that make up an object then adds more fields to the temp-table, one for each attribute in that class. The end result is a temp-table for each procedure-based SmartObject that has exactly one record. That record holds the attribute values for that particular SmartObject. The initial values of some of the attributes are set by the ADD-NEW-FIELD methods that add fields to the temp-tables dynamically. Other attribute values are assigned either at design time or at runtime.

Each of these include files also defines a number of preprocessor values that begin with “xp” followed by the name of the attribute. These act as flags to indicate which of the attribute values can safely be set or retrieved at runtime simply by looking directly at the field in the temp-table record. If the “xp” preprocessor is defined for that attribute, then this is done. Otherwise setting and retrieval must be done by executing functions whose names are “get” and “set” plus the name of the attribute. Typically, the get and set functions are defined in the super procedure of the class. This would be appropriate for cases where the attribute value must be calculated before it is retrieved, or when setting the attribute value has some side effect that must be executed beyond just setting the value in the temp-table.

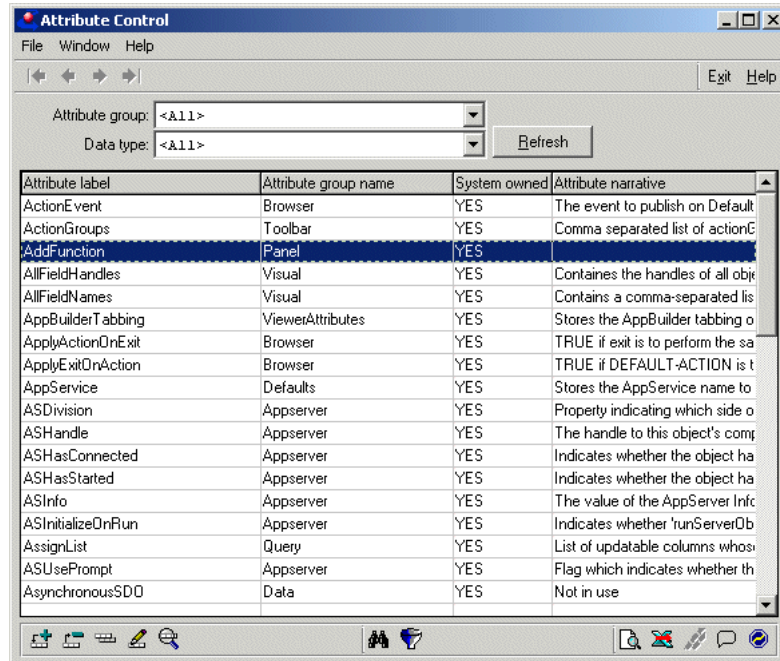
Here is the start of that list of “xp” preprocessors from smrtprop.i:

```
&GLOB xpObjectName  
&GLOB xpObjectVersion  
&GLOB xpObjectType  
...
```

When each static SmartObject is compiled, all of the include files that define all of the attribute fields get compiled into it. Runtime code then sets and manages the values in the temp-table record.

2.3.2 Adding a new attribute to the repository

If you need to extend the hierarchy because of new attributes, the first step is to add the attributes to the repository. To do this, choose the **Attributes→Attribute Control** menu item in the **Development** menu:



Select the add button to add a new attribute:

The screenshot shows the 'Attribute Maintenance' dialog box. It has a menu bar with 'File', 'Window', and 'Help'. Below the menu bar is a toolbar with various icons, including a plus sign for adding attributes. The dialog is divided into several sections: 'Details' with fields for 'Attribute group' (set to 'DataVisual'), 'Attribute label' (set to 'MandatroyField_Ext'), and 'Narrative' (containing a text area with the text 'Developer-specified list of mandatory fields: to highlight beyond those derived from the schema.'). Below these are 'Data type' (set to 'Character') and 'Object types' (an empty list). The 'Runtime and Realization Settings' section includes 'Constant level' (set to 'Instance Level'), checkboxes for 'Private', 'Runtime only', 'Derived value', 'System owned', and 'Design only', and an 'Override type' section with radio buttons for 'None - Allow direct access to attribute value' (selected), 'Get - Force value to be retrieved using get function', 'Set - Force value to be saved using set function', and 'Both - Force use of both set and get functions'. The 'Property List Settings' section includes 'Lookup type' (set to 'Free Text') and 'Lookup value' (an empty text area).

CAUTION: It is very important that you do not add new attributes to the classes already in the Object type hierarchy as it is shipped with the product, or modify the initial value of an attribute in an existing class. If you do this, then your extensions may be overwritten and lost when you upgrade to the next new version of Progress Dynamics.

2.3.3 Providing distinctive names for attributes and other objects

There is no strict naming convention for custom attributes or custom object types in Progress Dynamics. Names within the repository data as shipped with the product are typically “plain text” names that don’t have any distinctive pattern, beyond these basic characteristics:

- Most names do not contain punctuation such as hyphens or underscores, but rather use mixed case to separate the sub-parts of a compound name such as ObjectType.

- The principal exception to this is attribute names that map directly to native widget attributes in Progress, such as WIDTH-CHARS. These attributes are in uppercase and use hyphens in the same way that the native Progress attributes do, so that the repository values for the attributes can be mapped automatically and dynamically to the native attributes when objects are created at runtime. The upper or lower case letters of the name are not significant, of course, as Dynamics names, like Progress names in general, are not case sensitive. The convention of using upper case or mixed case just helps to identify the origin and use of the name.

This means that the most effective way for you to create distinctive names for attributes, object types and objects is to use an underscore somewhere in the name, possibly between the identifying part of the name and a prefix or suffix that distinctively identifies your custom attribute or object. Dynamics will not use the underscore character in names added to the repository in the future, so this type of convention will assure that your custom objects don't collide with those introduced into the product in future releases.

NOTE: There are at present a number of session property names that use underscores; it seems to be rather the norm there rather than the exception, so this rule will not work effectively for new session properties.

Perhaps the most distinctive prefix or suffix you could use, and one that would provide the most assurance that there will be no collision with names possibly introduced in integrating your application modules with other application modules in the future, is the site ID of the primary development site for your application. If others obey that convention, or at least have no reason for using that particular integer as an identifier, then your names will remain unique. We hesitate to make this a blanket recommendation as it may be odious to have to type your site ID every time your code references one of your attributes. In any case, be aware of the potential for conflict.

2.4 Extending the class hierarchy

In order to use your new attribute, you have to extend the class hierarchy to provide a new level where the attribute is used. You may use the Object Type Control or the Repository Maintenance tool. The approach is to create a new class using these tools that extends an existing class, and then add the new attributes to this extended class. (See the tutorial at the end of this chapter for step-by-step instructions on adding the class and attribute using the tools.)

If you're familiar with standard Version 9 SmartObjects, you know that all of the attributes as well as the class hierarchy are defined through a series of nested include files that make up each static SmartObject. When the Object is compiled, the statements in the series of include files define both the list of super procedures that are started and added to the Object's super procedure stack at runtime, and also the list of fields to be added to a dynamic temp-table that holds a field for each attribute the Object supports.

Dynamics does the same thing, but the list of attributes is obtained from the repository rather than from the include files. The same temp-table is built up, but from a Dynamics-specific function called `prepareInstance`.

The repository also holds the hierarchy of classes that make up each Object type. At runtime a properties temp-table record is created for each Object, with a field for every attribute inherited from all of the classes that make up the object type hierarchy for the Object. This includes everything from the “base” class (representing `smart.p` and all the ADM properties from `smrtprop.i`) down through the subclass that defines the Object type directly.

This repository-based attribute definition is for Objects used in Progress Dynamics applications only. If you are extending an Object type in the middle of the hierarchy that needs to function in a standard Progress Version 9 ADM2 application with static SmartObjects, then you need to define the attribute in the standard way as well, by editing its definition into the `<class>propcustom.i` file for the class and recompiling Objects with that file. If you are extending a class at the bottom of the hierarchy class, you would use the New ADM class tool to create the necessary include, property, super procedure and prototype files. See the ADM2 documentation for more detail.

2.4.1 Understanding the rendering procedure

Beginning with Progress Dynamics Version 2.1A, you can now define the rendering procedure for a class. This is the procedure that creates the dynamic Object at runtime from data in the repository. For example, for a dynamic viewer (the `DynView` class) this is `rydynview.w`. It is unlikely that you would need to define a different rendering procedure for a subclass of a standard class, but if you did, there now is a means to do so. The rendering procedure used in the framework for a specific object is based on a new property called `RenderingProcedure`. If you wanted to change the rendering procedure for your extended class, you could simply change the value of this attribute to point to the new procedure by entering the relative path filename of the new procedure.

Also, starting with Progress Dynamics Version 2.1A, you can associate a super procedure with your extended class. A new property called `SuperProcedure` has been introduced which allows you to specify the super procedure for a class, or for a specific object. You can specify a super procedure (relative path file name) for a class and also for an object, and both supers will be loaded into the stack

If you need to change the rendering procedure or the custom super procedure for a class, you can do it in the Object Type Control tool.

2.4.2 Thin rendering objects

There exist special thin versions of most SmartObject renderers that contain almost no r-code as they exclude logic from their compilations that are not required for dynamic objects, such as logic to create ADMProps and custom include file inclusion. They also exclude the prototype definitions, which really is independent of whether the object is dynamic or not. The session attribute `UseThinRendering` is used to determine which session that will use thin rendering. Thin Rendering can also be turned on or off for a class of objects or for individual object masters or object instances.

If only thin renderers are used there is no need to implement the `start-super-proc` in the custom include or any properties in the custom prop files. Since the prototypes are excluded, direct references to functions in thin objects would need to be changed to use `DYNAMIC-FUNCTION`.

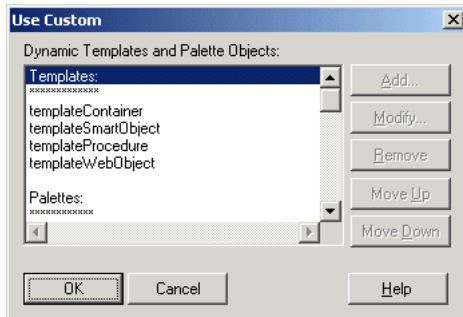
2.4.3 Defining support functions for new attributes

When you define a new attribute that is required for an extended class, you can set a value that registers in the repository whether the attribute can be accessed directly without going through the standard get and set functions. However, this information is used only by the Repository Manager and not by references to the attributes in other code. For this reason, you either need to define functions for the attributes or set up a property include file as described in the [“Supporting your new attribute in standard SmartObjects.”](#) Since the functions are needed for other objects to be able to access your attributes, and for the attributes to work with standard static SmartObjects, it is best to include this file in your compilation in any case.

2.4.4 Adding the new class to the AppBuilder

When you bring up the AppBuilder to do development, you have available both the AppBuilder’s **Palette** window, which lets you choose existing objects to add to a design window, and also the **New...** menu item and the **New** button to let you create new objects. In standard Progress ProVision and in Progress Dynamics, the definitions of all the objects on the **Palette** and in the **New** dialog have been defined in a set of text files the AppBuilder reads in when it starts up. These are called “custom” files, since they can be extended in a text editor to define new types of objects and also individual specialized objects you can choose. They all have the filename extension `.cst` and are typically located in the `src/template` directory. If you select the **Menu→Use Custom** option in the **Palette**, you see a list of the files the AppBuilder reads in to build the **Palette** and the **New** dialog.

Beginning in Progress Dynamics Version 2.0A02, all of this information is also stored in the repository database, which allows you to maintain it through repository tools rather than manually editing these text files. If you create a new repository database in SP2, the information represented in the standard .cst files is already in the repository. If you choose the **Menu→Use Custom** option in the **Palette** window in SP2 or later, you see a list of repository classes instead of text files:



The Templates are repository object records (even though they don't all necessarily represent Progress SmartObjects). These are used to populate the **New** dialog. The Palettes are another repository class used to populate the **Palette** window.

In order for you to be able to use the new Object Type in your applications, you need to make it available from the AppBuilder's **New...** menu item or the **New** button in the toolbar. To do this, there needs to be a template object record in the repository representing your new class. You can add the new template record in the Repository Maintenance tool, which you can select from the **Build** menu in the AppBuilder. You will need to add a new object for your extended class which will act as a template for the appBuilder. You will then have to create an object within the Template class to be able to add your own objects and object instances.

2.4.5 Supporting your new attribute in standard SmartObjects

As noted, you must create a `src/adm2/custom/viewpropcustom.i` file if you want to define an "xp" preprocessor for the new attribute or if you want the code to work in standard ADM2 static SmartObjects that don't have access to the Progress Dynamics repository.

In this file you add a `&GLOBAL-DEFINE` for each attribute that should be accessible directly, without going through the get and set functions.

Within the preprocessor block that follows this, you also need to add a statement to add the new attribute temp-table field to your objects. This is the step that is done dynamically by the prepareInstance function using information in the repository, when you are using dynamic Objects and the Dynamics repository:

```
/* ***** Main Block ***** */

/* Include the file which defines prototypes for all of the super
   procedure's entry points.
   And skip including the prototypes if we are *any* super procedure. */

&IF "&ADMSuper":U EQ "":U &THEN
  {src/adm2/custom/viewprtocustom.i}
&ENDIF

&GLOBAL-DEFINE xpMandatoryFields_Ext

&IF "&ADMSuper":U = "":U &THEN
  ghADMPProps:ADD-NEW-FIELD('MandatoryFields_Ext':U, 'CHARACTER':U, 0, ?,
  "").
&ENDIF
```

After you've added new attributes and extended the Object type hierarchy as described, you need to restart your session to capture the effects of your changes.

2.5 Changing the object type of a set of objects

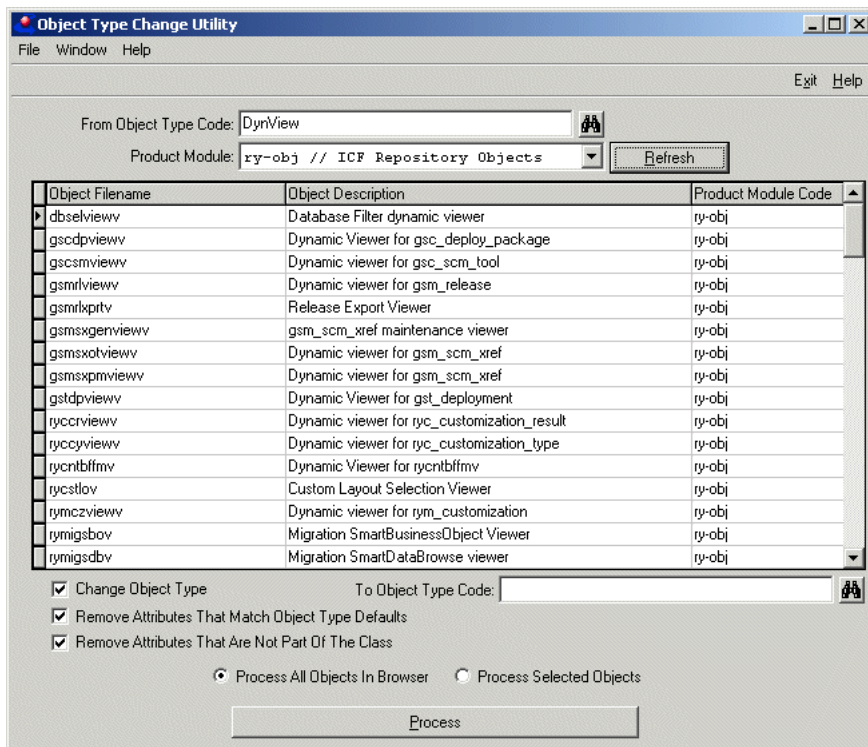
If you are extending a class at the bottom of the hierarchy, you may want to change the object type of a whole set of objects at one time. For example, you may define a new subclass of a standard object type such as DynView and then move either all of your objects of that type, or perhaps a subset of your objects that require the extended behavior, to the new class. Alternatively, you may find that you extended a class in an earlier release in order to provide behavior that is now part of the standard object, and you want to return your objects to the standard type.

You can move objects to a new type using a utility available from the **Object Type Control** window. In that window, select **Class Options**→**Object Type Change Utility**.

To operate on a set of objects, enter their object type in the fill-in marked **From Object Type Code**, or use the lookup button to select an Object type.

You can also select a **Product Module** to filter objects on that basis.

After you select one or both of these values, press the **Refresh** button to see the list of all objects that match your selected values, as in this example:



The utility allows you to change Object types, and/or to make other changes to the objects you select. If you want your changes to apply to all the Objects in the browse, then leave the radio set at the bottom of the window at its default value of **Process All Objects In Browser**. If you want to further subset the Objects, then select the **Process Selected Objects** radio set value and use the multi-select browse to select those specific Objects you want to operate on. If you don't change the radio set to **Process Selected Objects**, then selecting individual rows in the browse has no effect.

Once you have identified a set of Objects to work with, the following sections describe the things you can do to modify them.

2.5.1 Changing the object type

If you want to change the Object type of the Objects, then check the **Change Object Type** toggle on and enter the target Object type in the **To Object Type Code** field, or select it using the lookup button. You can change the Object type to any other type either above or below the current Object type in the Object type hierarchy. If you want to change the Object type to a type somewhere else in the hierarchy, for example to an Object type that is a sibling, parallel to the current Object type, then you need to do this in two stages, first moving the Objects up the hierarchy to a common ancestor, and then back down the tree to the proper destination.

Having said this, it is very important that you keep in mind that you cannot arbitrarily assign new Object types to Objects without considering the effects of this. Changing the Object type could render some of the attributes useless if the Object type it is being changed to does not have those attributes. Generally it will only make sense to change an Object type to another closely related type, typically to change Objects to an immediate new sub-type or from a sub-type back to a standard type. A two-step move would therefore likely be necessary only when you need to move a set of Objects to a sibling position under the same parent type as before.

In considering the relationship between Object types, use the treeview representation in the **Object Type Control** window itself. This is the definition of the hierarchy that matters where changing Object types is concerned, because this reflects the actual representation of Object types in the repository.

2.5.2 SCM integration considerations

When extending a class at the bottom of a hierarchy and changing the object types of objects, there are certain SCM implications that need to be considered. The new Object Type has to be registered in the SCM Object Type Xref control, which is accessible from the SCM menu off the Dynamics Development menu. Any required Roundtable subtype change also needs to be taken into consideration.

For more information, refer to the white paper *Progress Dynamics® Version 2.1 Source Code Management: Roundtable™ TSMS Installation and Setup*. (You can find this document on www.progress.com.)

2.5.3 Removing redundant attribute values

The utility allows you to do more than just change Object types. You can also use it for two types of repository cleanup work relative to attribute values. Attribute value records are normally stored only at the highest level at which they have a distinct value. For each Object attribute, an attribute value record with the type's default value is stored at the level of the class. If that value is not changed in a master Object created from the type, then no attribute value record is stored with the master. Likewise, if an attribute for an instance of that master (a Viewer in a particular window, for example) does not have a distinct value for an attribute, then no record is stored at that level.

Over time, and especially in the course of moving Objects from one type to another, you may wind up with redundant or invalid attribute value records in the repository as part of your Object definitions. Redundant attribute value records are those that have a value identical to the default value for Object type. If you have redundant attribute value records, you can improve the performance of your application and reduce the size of the repository data by removing these records. You can also improve maintainability of the class attribute if you want to modify the value. To do this, check on the toggle labeled **Remove Attributes That Match Object Type Defaults**. If the **Change Object Type** toggle is also checked, the result will be the removal of attribute values for all the Objects in your list that match the default attribute values of the Object type you are changing to. If you simply want to remove these attribute values without changing the Object type, then check the **Change Object Type** toggle off. In this case the utility will remove attribute values for the object that match the default attribute values of its current Object type.

2.5.4 Removing invalid attribute values

When you change Objects from one type to another, you may also wind up with attribute values for attributes that are no longer a part of the class at all. These attributes will not be retrieved by the runtime API and so should be removed. They may, however, be left where they are for informational purposes.

To remove them, check on the toggle labeled **Remove Attributes That Are Not Part of the Class**. If you are changing Object type at the same time, this will remove attribute values for each Object that are not for attributes of the new Object type. If you are not changing the type, the utility will remove values that are not for attributes for its current Object type. Presumably this would be the result of an earlier change of Object type or some other data conversion.

By default all these toggles are checked on. This means that as you move Objects from one type to another, both redundant and invalid attribute value records will be removed at the same time.

2.5.5 Processing the changes

Before you use the Change Object Type utility, it is wise to make sure you have a current backup of your repository database in case the changes do not wind up being those that you intended. Once you have chosen the list of Objects to work with and made the correct selections from the lower part of the window, press the **Process** button. The utility displays a confirmation message describing the nature of the changes it is about to make, and asking if you want to continue. Verify that this is what you want and answer “Yes” to the question.

If any errors occur during processing, an error message is displayed, processing stops, and any processing that had been done is backed out. Whether the processing succeeds or is aborted because of an error, a log file called OTC.log is written to the working directory. It displays the options that were selected and lists the objects that were processed. If an error occurred, the error message is written to the log file. If you run the utility multiple times, information and messages are appended to the file rather than overwriting it. You should also clear the log file from time to time for performance purposes.

2.5.6 Resetting the session cache to test the results

Once processing is complete, exit and restart your Progress Dynamics development session. The Repository Manager caches Object types and their attributes, and therefore this information all needs to be flushed from the session in order for all the changes to be reflected.

Alternately, if you have defined a new subclass or made changes to any existing class to define or alter its attributes or their default values, you can run `destroyClassCache` in the Repository Manager, which will actually delete the temp-tables themselves and force everything to be reconstructed. Since the temp-tables include a field for each attribute, this is necessary if the list of attribute itself has changed for any classes. Including `globals.i` gives you access to the Manager handle:

```
{src/adm2/globals.i}  
RUN destroyClassCache IN gshRepositoryManager.
```

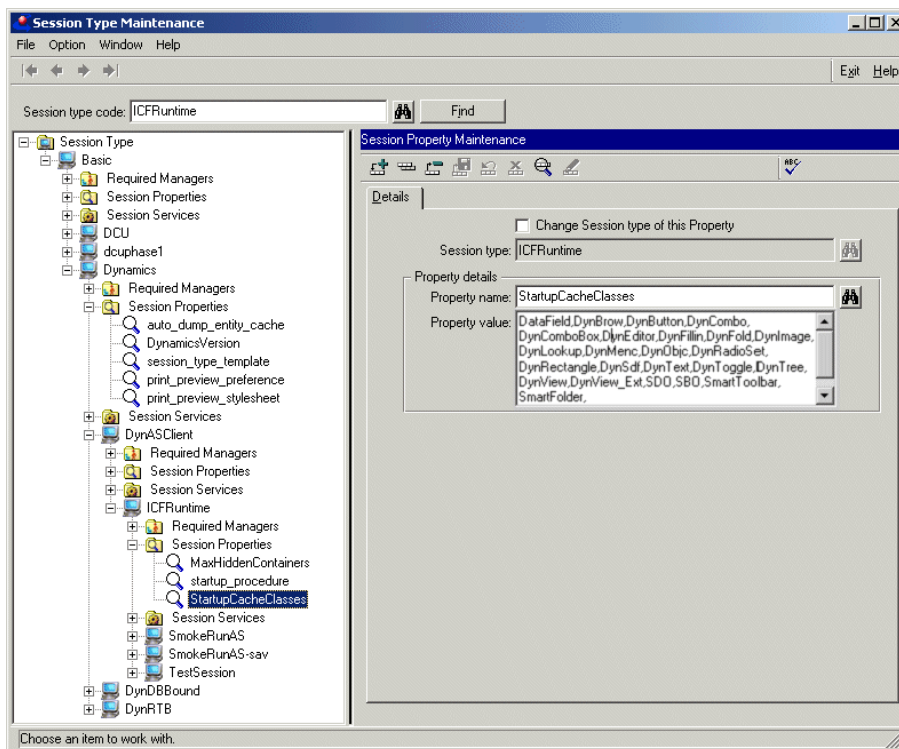
If you have a client side cache for your classes, you will need to regenerate this cache in order to test the results. It is also recommended that all files beginning with “class_” are deleted from the client cache directory (typically the `ry/clc` directory).

2.5.7 Modifying the classes cached at startup

Any class in the class hierarchy can be cached automatically at startup to improve the performance of the application modules when you first run them. You may want to consider adding any new classes you create to this list. If these classes will be part of application components that are typically run, then you may improve the initial performance of your application by adding your classes to the list.

To support this, there is a Progress Dynamics session property called `StartupCacheClasses` where you can define a comma-separated list of any class including your custom classes. If you want to define this attribute for one or more of the session types your users will be running in, select **Session→Session Type Control** from the **Dynamics Administration** menu. Find the session type (that is, **ICFRuntime**) and expand the node to display the **Session Properties** node. Right click on the node and select **Add Session Property**.

Enter `StartupCacheClasses` as the **Session Property** or select it from the lookup, and enter your list of classes as the value. You can also enter the value asterisk (*) to indicate to the system to cache all classes. Note that the number of classes cached at startup may affect the time taken to bootstrap a Progress Dynamics session. If the property value is left blank, then classes will be cached on demand:



Save this new property value. Select the **Session→Generate Configuration File** menu item to recreate the ICFCONFIG.XML file where your session values are stored.

2.6 Using and extending dynamic basic objects

In addition to subclassing SmartObject types, it is also possible to subclass basic field-level objects such as fill-ins or buttons. The behavior described in this section depends on features in Dynamics Version 2.0A02, especially the work to represent the AppBuilder custom files that make up the palette and the **New** dialog in the repository. This new work makes it possible to select these extended basic objects from the AppBuilder **Palette** window and add them to a design window.

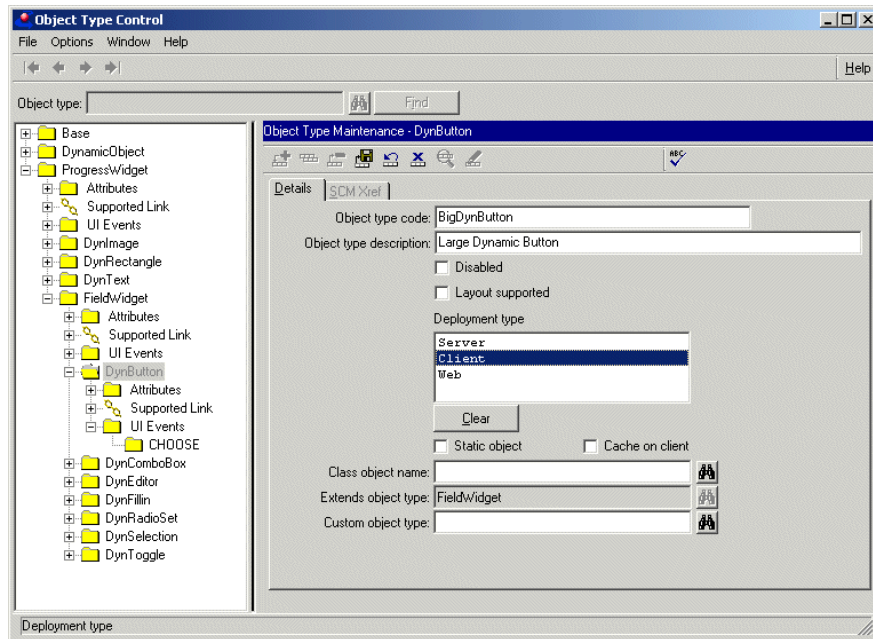
There are two levels at which you can subclass a basic object (or widget). You can actually subclass it in the Object Type Control, as you did with the DynView SmartObject, or you can simply add a new instance of an existing basic object on the **Palette** with its own attributes, much as there are already specialized objects such as **OK** and **Cancel** buttons on the **Palette** with their own attributes. In most cases, simply defining a new object instance will be sufficient, if what you want is an object with some specific attribute values. If you do not need to subclass a palette object, but simply want to add a variation of a button on a palette, you can skip to [“Adding a custom palette instance.”](#)

CAUTION: If you want to add new instances as described below, it is important that you do not add the instance to an object shipped with the product. You must create your own object and add the instance to your object; otherwise your additions may be overridden with the next update of the product.

2.6.1 Subclassing a button object

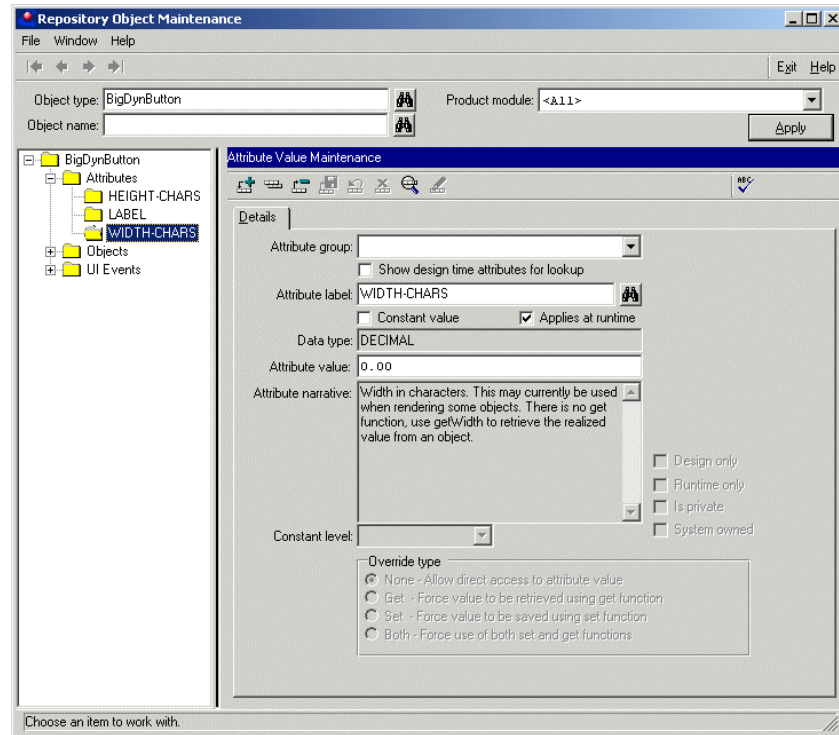
First, the text walks through the steps of subclassing a dynamic object to see what's involved. You will then subclass the dynamic button to create a new button type with some special attributes.

- 1 ♦ Go into the **Object Type Control**. All SmartObjects are under the **Base** node. Basic objects are under the **ProgressWidget** node:



- 2 ♦ Expand the **ProgressWidget** node, then the **FieldWidget** node. Note that buttons are under the **FieldWidget** node, along with true field-level objects even though they are not actually fields.
- 3 ♦ Expand the **DynButton** node. If you want, you can expand the **Attributes** node to see a list of all the attribute values that are defined for the standard dynamic button. Any new subclass of DynButton will inherit all of these attributes and values, unless a value is specifically overridden in the subclass.
- 4 ♦ Right click on the **DynButton** node and select **Add Object Type**.
- 5 ♦ In the **Maintenance** frame, enter an **Object Type Code** of **BigDynButton** and a **Description**. Select **Client** as the **Deployment Type**.
- 6 ♦ Press the **Save** button.

- 7 ♦ Exit the Object Type Control and go into the Repository Maintenance Tool. Here you will first define the special attributes for your new object subclass:

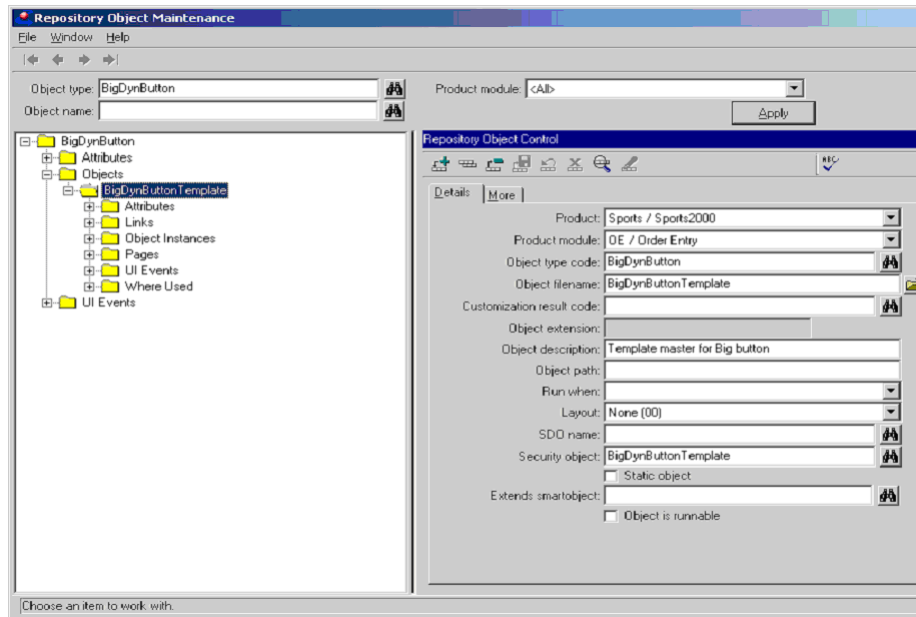


- 8 ♦ Enter **BigDynButton** as the **Object Type** and press **Apply**.
- 9 ♦ Right click on the **Attributes** node and add some new attribute values for the new type. For example, you can define a new LABEL for the button of “Big Button”, a HEIGHT-CHARS of 2.00 and a WIDTH-CHARS of 30.00, to make it a big button.

Next, you will define a template record and an instance for the new subclass much as you did for the new SmartObject class. But instead of doing this under the Template Object Type, you’ll do it under the Palette Object Type. As you have seen, the Template class is the basis for the New dialog. The Palette class is the basis for objects on the Palette.

To define the record:

- 1 ♦ Right click on the **Objects** node to add a new **Object**:



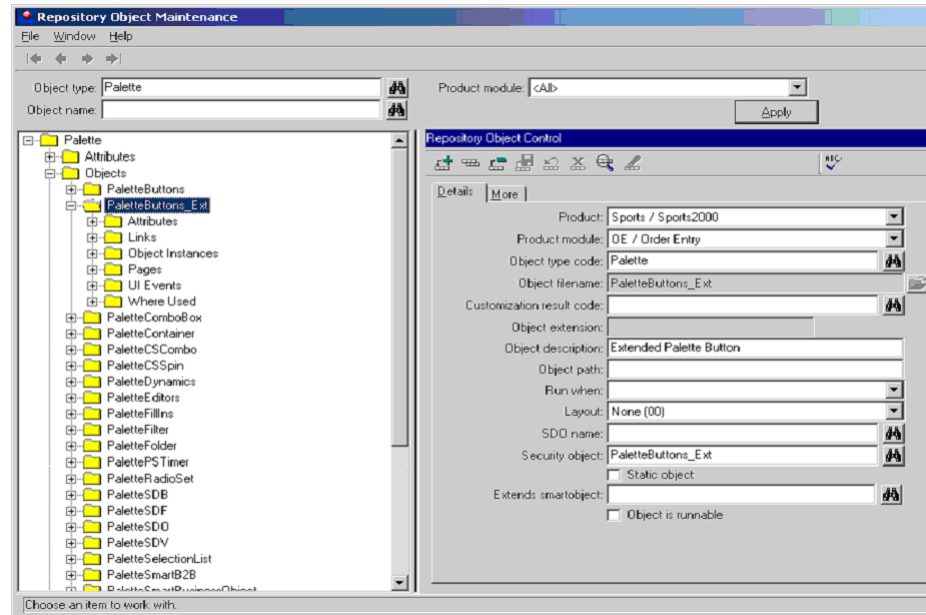
- 2 ♦ Enter a **Product** and **Product Module**.
- 3 ♦ Enter or select the **Object Type Code BigDynButton**.
- 4 ♦ Enter an **Object Filename** of **BigDynButtonTemplate**.
- 5 ♦ Enter an **Object Description** and select the **Layout type None**.
- 6 ♦ Press the **Save** button.

Now you have created an actual subclass of the dynamic button type which could be used as the basis for multiple other objects that inherited the height, width, and label of the BigDynButton object. In order for a new palette button to appear in the appBuilder palette, you still need to define an instance of the palette object with certain attributes.

2.6.2 Adding a custom palette instance

In most cases, as we noted, subclassing a palette object is not really required. If you just want to define some special attributes for a button type (or any other object type), you can just create a Palette instance for it. For example, you can follow these steps to create a new dynamic button that is wider than the normal button:

- 1 ♦ Still in the Repository Maintenance Tool, enter **Palette** as the **Object Type** and press **Apply**:



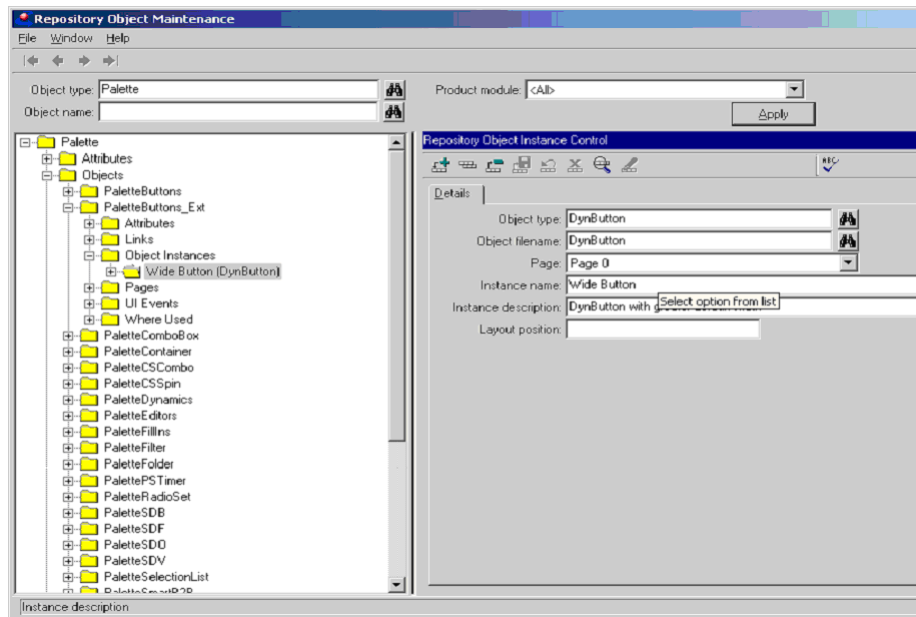
- 2 ♦ Expand the **Palette** node, and the **Objects** node under it. Here you will see a list of all the principal subclasses of dynamic objects on the **Palette**, including **PaletteButtons**, **PaletteContainer**, and so on.
- 3 ♦ Create a new object of the Palette class called **PaletteButton_Ext** by right clicking on the **Objects** node and selecting **Add Smart Object**.
- 4 ♦ Enter a **Product** and **Product Module**.
- 5 ♦ Enter or select the Object type code **Palette**.
- 6 ♦ Enter the **Object Filename** **PaletteButtonsExt**.

- 7 ♦ Enter an **Object Description** and select the **Layout** type **None**.
- 8 ♦ Press the **Save** button.

Next you need to define the Palette instance for the new object type template.

- 1 ♦ Expand the **PaletteButtons_Ext** object and right click on the **Object Instance** to create a new instance and fill it in as in this screen shot to create a new button instance called **WideButton**.

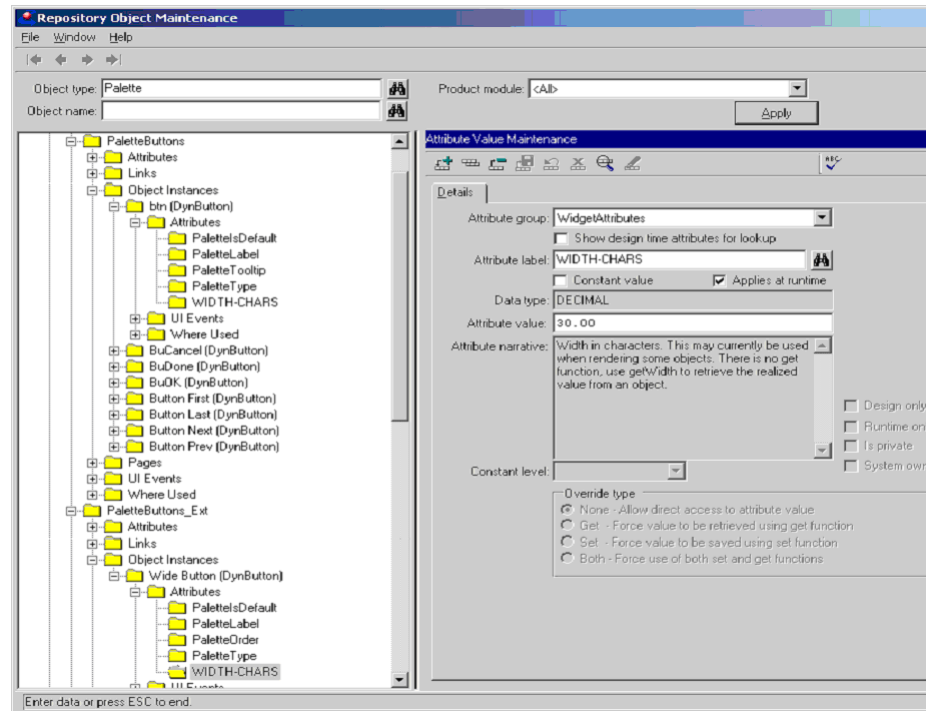
NOTE: Note that we are specifying the object type DynButton. If we wanted to use a subclassed object, as specified in the previous section), we would have specified that object type and object here:



- 2 ♦ In order to view the various required attributes, expand the **PaletteButtons** object and also the **btn (DynButton)** node instance. This is the basic instance for all dynamic buttons. Expand its **Attributes** as well.

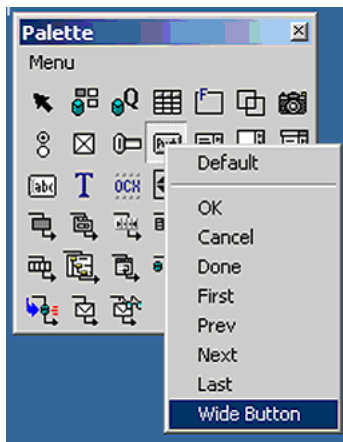
You see that it has an attribute called **PaletteIsDefault**, indicating it is the default object for this object type on the Palette. Because your new object will not be the default, you do not want to duplicate this attribute. However, you do need to create attribute values for the other attributes, much as you did for the basic attributes of your new dynamic Viewer.

- 3 ♦ Right click on the **Attributes** node of the new **WideButton**, and add attribute values for **PaletteLabel (&Wide Button)**; **PaletteType (Button)**; and the new attribute **WIDTH-CHARS (30.0)**. You don't need to define the **PaletteTooltip** attribute unless you want your button to have a distinctive tooltip. Also create a **PaletteOrder** attribute, which determines the objects position in the list of specialized buttons. If you give it a value of 10 in this case, it will appear at the end of the list when you right click on the button in the appBuilder palette:

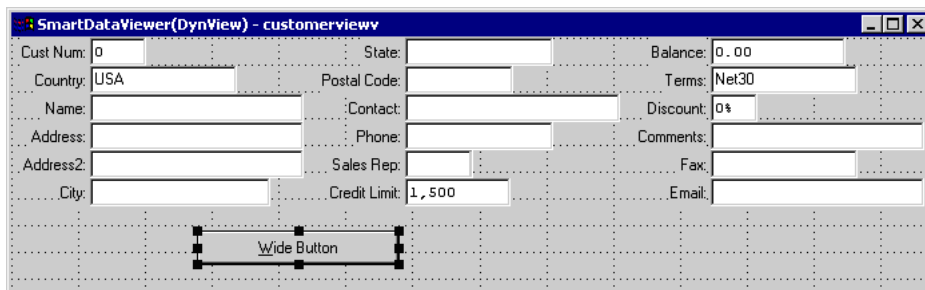


- 4 ♦ (IMPORTANT) Add the new palette object (paletteButtons_Ext) to the list of objects defined in your icfconfig file for the tag IDEPALETTE in a comma delimited format. Alternatively, you can go to the user preferences by selecting the menu option **File→Preferences**, and then enter the new template object as a comma delimited list in the **Custom Palettes** field. (that is, *, paletteButtons_Ext)
- 5 ♦ You can exit the Repository Maintenance tool, and select **Menu→Use Custom** from the **Palette** window and press **OK** to reload the Palette from the new data in the repository.

Now your new button has been added to the **Palette**. If you right click on the button icon in the **Palette** window, you'll see it in the list of button instances:



You can select your new Wide Button and drop it onto a design window and see its new attributes:



2.7 Extending object classes tutorial

The following is a step-by-step guide to help you get started with extending dynamic object classes. This tutorial illustrates extending a class at the bottom of the hierarchy. As an example of extended behavior, you will add a subclass to the DynView class in order to add a new attribute at that level, and define behavior for the extended class that uses that attribute's value. The new behavior is to display all mandatory fields in a viewer highlighted in yellow, so that the user knows which fields must be entered. The extended code builds an initial list of these mandatory fields by looking at fields that are in indexes and assuming that these are all mandatory.

We'll then show the actual subclassing process by defining a new attribute that the extended behavior needs to use. The new attribute we define allows the developer to define additional mandatory fields beyond those identified through the index definitions in the meta-schema. Because of the new attribute, we have to subclass the DynView class to add the new attribute at that level.

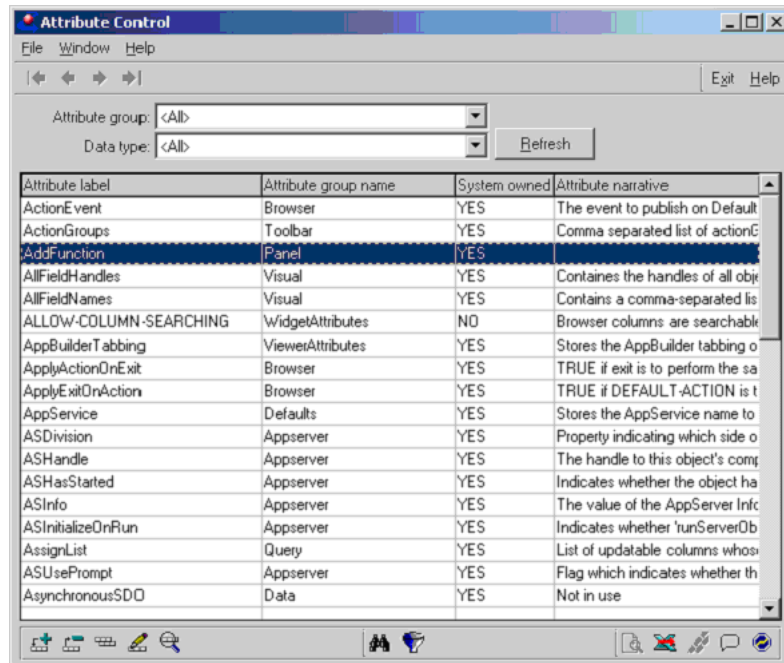
2.7.1 Adding a new attribute

To add a new attribute:

- 1 ♦ Choose the **Attributes→Attribute Control** menu item in the **Dynamics Development** menu:



The **Attribute Control** window appears:



- 2 ♦ Select the **Add Record** button to add a new attribute:

The screenshot shows the 'Attribute Maintenance' dialog box with the following settings:

- Attribute group:** DataVisual
- Attribute label:** MandatoryField_Ext
- Narrative:** Developer-specified list of mandatory fields: to highlight beyond those derived from the schema.
- Data type:** Character
- Object types:** (empty)
- Runtime and Realization Settings:**
 - Constant level:** Instance Level
 - ☐ Private
 - ☐ Runtime only
 - ☐ System owned
 - ☐ Derived value
 - ☐ Design only
 - Override type:**
 - ☒ None - Allow direct access to attribute value
 - ☐ Get - Force value to be retrieved using get function
 - ☐ Set - Force value to be saved using set function
 - ☐ Both - Force use of both set and get functions
- Property List Settings:**
 - Lookup type:** Free Text
 - Lookup value:** (empty)

- 3 ♦ Enter the following information:

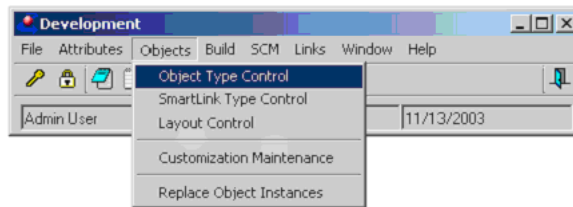
- **Attribute group** — DataVisual. This is just a way of helping to locate and identify the attribute by filtering on this group.
- **Attribute label** — MandatoryFields_Ext. This is the name of the new attribute. (see section Providing Distinctive Names for Attributes).
- **Narrative** — Developer-specified list of mandatory fields to highlight beyond those derived from the schema. (This description is shown in the Dynamic Property Sheet when adding values to the attribute).
- **Data type** — Character.
- **Constant level** — Instance Level. This implies that the attribute value can be set for all classes, objects and instances. 'Class Level' is used if the attribute value can only be specified for a class. 'Master Level' implies that an attribute value could be specified for a master Object (that is, a SmartView).

- 4 ♦ Press **OK** to save the new attribute and exit the window, and press **Exit** to leave the **Attribute Control** window.

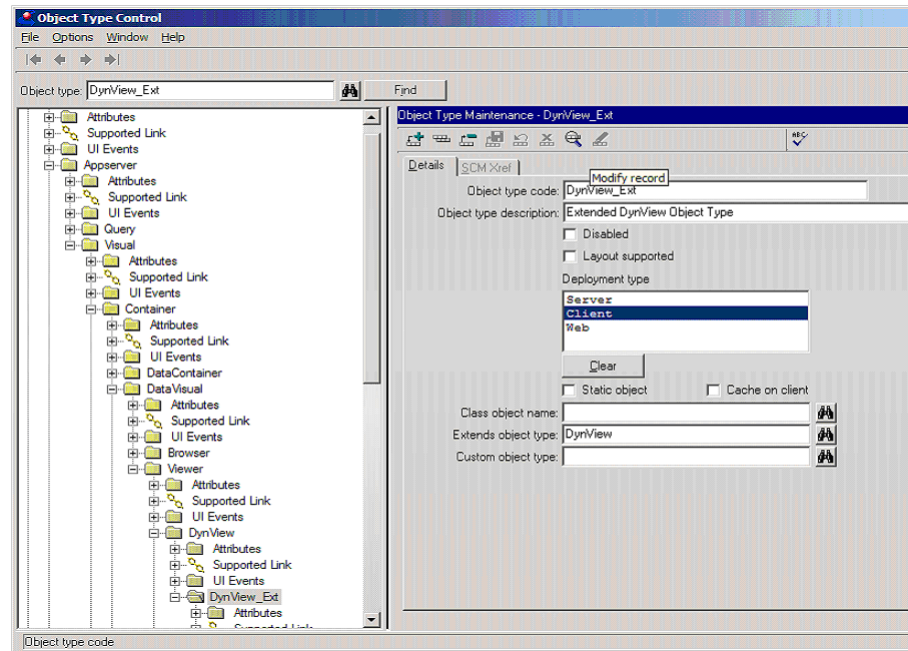
2.7.2 Extending the class at the bottom of the hierarchy

In order to allow your new attribute to apply to new objects that you may create, you have to extend the class hierarchy from the bottom to provide a new level where the attribute is used. As noted before, it is important that you not add new attributes or change attribute initial values in existing classes.

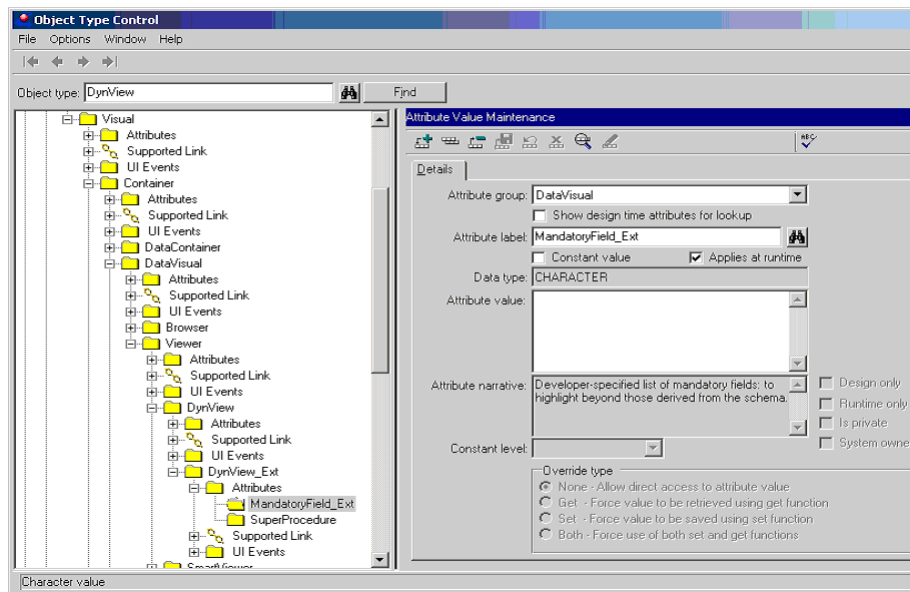
- 1 ♦ Choose the **Objects→Object Type Control** menu item in the **Dynamics Development** menu:



The Object Type Control appears:



- 2 ♦ Enter **DynView** as the **Object Type** and press the **Find** button.
- 3 ♦ Right-click on the **DynView** node and select the **Add Object Type** popup item and enter the following:
 - **Object type code** — DynView_Ext.
 - **Description** — Extended DynView object type.
 - **Deployment Type** — Client. Any code associated with this extended class will need to be deployed to only client machines.
 - **Class object name** — Not used.
 - **Extends Object Type** — DynView, This should be filled in for you to be the Object type you have just subclassed.
- 4 ♦ Press the **Save** button when you have completed entering data for the new class. Next you need to add the attribute MandatoryFields_Ext to this new subclass.
- 5 ♦ Right mouse click on the **Attributes** node below DynView_Ext and select the **Add Attribute Value** popup item:



6 ♦ Enter the following:

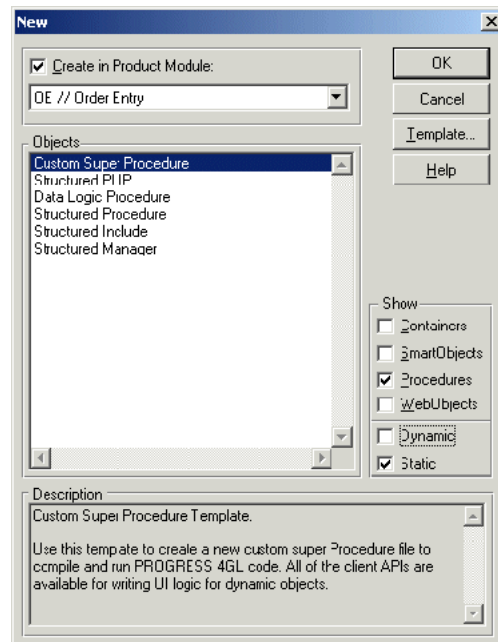
- **Attribute label** — MandatoryFields_Ex. You may use the lookup or simply type in the value The rest of the data for the attribute is then filled in for you.
- **Attribute Value** — Leave blank. If you wanted to enter a default or initial value for the attribute you could do that by filling in the Attribute Value field. But in this case the default value is simply blank.

7 ♦ Press **Save** when you're done.

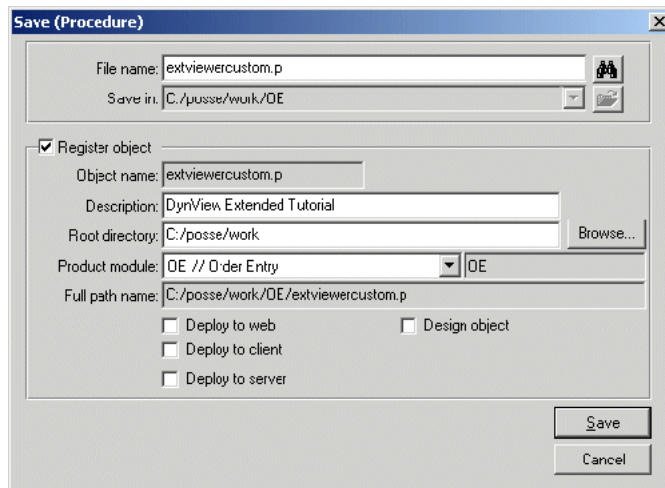
2.7.3 Defining custom behavior for the subclass

If you need to specify custom behavior for your new extended class, you can create a super procedure and associate it to the new class.

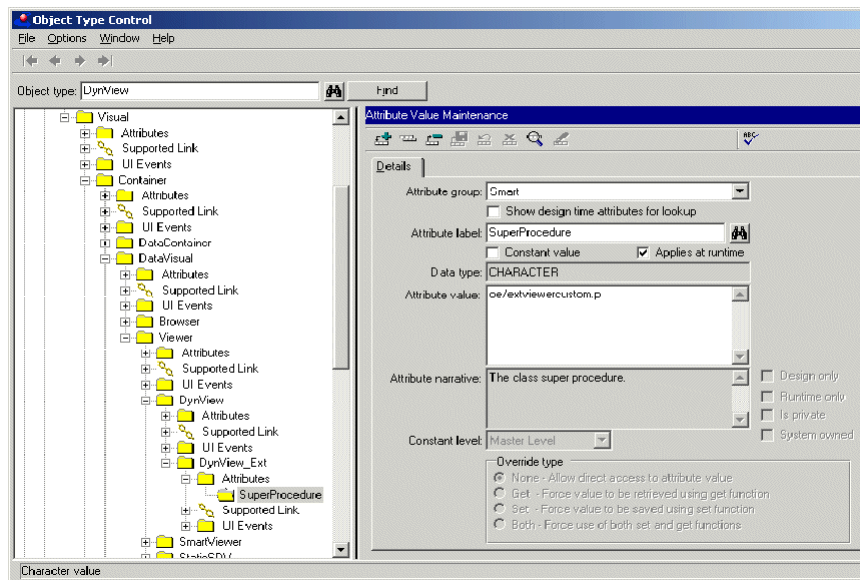
- 1 ♦ Create a super procedure starting with the Custom Super Procedure template in the AppBuilder by selecting the **New** button in the appBuilder:



- 2 ♦ Press the **Save** button in the AppBuilder to bring up the **Save** dialog. Ensure that the **Register object** checkbox is selected and register the procedure in a specified product module:



- 3 ♦ In the **Object Type Control** window, right mouse click on the **Attributes** node below DynView_Ext and select the **Add Attribute Value** popup item:



- 4 ♦ Enter the following:
 - **Attribute label** — SuperProcedure. You may use the lookup or simply type in the value.
 - **Attribute Value** — OE/ extviewercustom.p. Enter the relative path filename of the custom super procedure that you will create to override standard behavior of the viewer class
- 5 ♦ Press **Save** when you're done.
- 6 ♦ If you have generated a client side cache for performance optimization purposes, you will need to regenerate the class cache and redeploy. You may also want to add this new class to the session parameter StartupClassCaches for performance optimization. (See [Progress Dynamics Administration Guide](#) for more information about performance.)

2.7.4 Defining extended behavior for the new attribute

Follow these steps to build the custom super procedure with the new behavior:

- 1 ♦ Open the custom procedure extviewercustom.p that you created earlier and define a new initializeObject procedure.
- 2 ♦ Enter the variable definitions that it uses and the RUN SUPER statement to invoke the standard initialization behavior:

```
PROCEDURE initializeObject:
/*-----
-----
  Purpose:      Extends the standard behavior for Objects by highlighting
                  fields that are marked as mandatory.
  Parameters:   <none>
  Notes:
-----*/
DEFINE VARIABLE cFields      AS CHARACTER NO-UNDO.
DEFINE VARIABLE cHandles    AS CHARACTER NO-UNDO.
DEFINE VARIABLE iField      AS INTEGER NO-UNDO.
DEFINE VARIABLE iEntry      AS INTEGER NO-UNDO.
DEFINE VARIABLE hField      AS HANDLE NO-UNDO.
DEFINE VARIABLE cFieldList  AS CHARACTER NO-UNDO.
DEFINE VARIABLE hSource     AS HANDLE NO-UNDO.
DEFINE VARIABLE cTable      AS CHARACTER NO-UNDO.
DEFINE VARIABLE cMandExt    AS CHARACTER NO-UNDO.

RUN SUPER.
```

The first block of executable code after the RUN SUPER statement locates each _Field record in the Progress Sports2000 meta-schema that participates in an index for the current table. This example is simplified to the extent that it assumes that the current database (DICTDB) is the only database that needs to be queried. It might also be more appropriate to use the _Mandatory field of the meta-schema rather than making a list of fields that participate in indexes. In the Sports2000 database, however, the _Mandatory field is not used, so the example uses the technique of looking at indexed fields.

The example also makes direct access to the database from the client code, which is not good practice. In a completed application, you should always use the Session Manager to invoke a server-side procedure to return any values that require database access:

```
/* Build an initial list of mandatory fields by identifying
   all indexed fields. */
{get DisplayedFields cFields}.
{get FieldHandles cHandles}.
{get DataSource hSource}.
{get Tables cTable hSource}.

/* In case there's more than 1 table just use the first. */
cTable = ENTRY(1, cTable).

FOR EACH _File WHERE _File._File-Name = cTable,
  EACH _Field OF _File WHERE CAN-FIND (_Index-Field OF _Field):
    cFieldList = cFieldList + _Field._Field-Name + ",".
END.
```

Next the code uses the new attribute value if it has been set. It retrieves the list of extra mandatory fields and appends that to the list of fields derived from indexes:

```
/* This extended attribute may hold additional fields to
   be added to the mandatory fields list. */
{get MandatoryFields_Ext cMandExt}.
IF cMandExt NE "" AND cMandExt NE ? THEN
  cFieldList = cFieldList + "," + cMandExt.
ELSE cFieldList = RIGHT-TRIM(cFieldList, ",").
```

Finally the code gets the handle of each field in the list and sets its background color to yellow:

```
/* Highlight the mandatory fields in a Viewer. */
DO iEntry = 1 TO NUM-ENTRIES(cFieldList):
  iField = LOOKUP(ENTRY(iEntry, cFieldList), cFields).
  IF iField NE 0 THEN
    DO:
      ASSIGN hField = WIDGET-HANDLE(ENTRY(iField, cHandles))
      hField:BGCOLOR = 14 NO-ERROR.
    END.
  END.
END PROCEDURE.
```

- 3 ♦ Save the changes.

2.7.5 Defining support functions for new attributes

It is recommended to add support functions for your new attribute (although not required). Here by example is the function to allow your code to retrieve the new attribute value. It defines the necessary “xp” preprocessor temporarily in order to then be able to retrieve the value directly from the attribute temp-table record, retrieves it, returns it, and then undefines the preprocessor. This function and any others you need go into your custom super procedure.

- 1 ♦ Create the get and set support functions in your new custom super procedure:

```
FUNCTION getMandatoryFields_Ext
  RETURNS CHARACTER
  ( ) :
  /*-----
  -----
  Purpose: Retrieves the value of the extended list of
           mandatory fields
  Notes:
  -----*/
  &SCOPED-DEFINE xpMandatoryFields_Ext
  DEFINE VARIABLE cMandExt AS CHARACTER NO-UNDO.
  {get MandatoryFields_Ext cMandExt}.
  RETURN cMandExt. /* Function return value. */
  &UNDEFINE xpMandatoryFields_Ext

END FUNCTION.
```

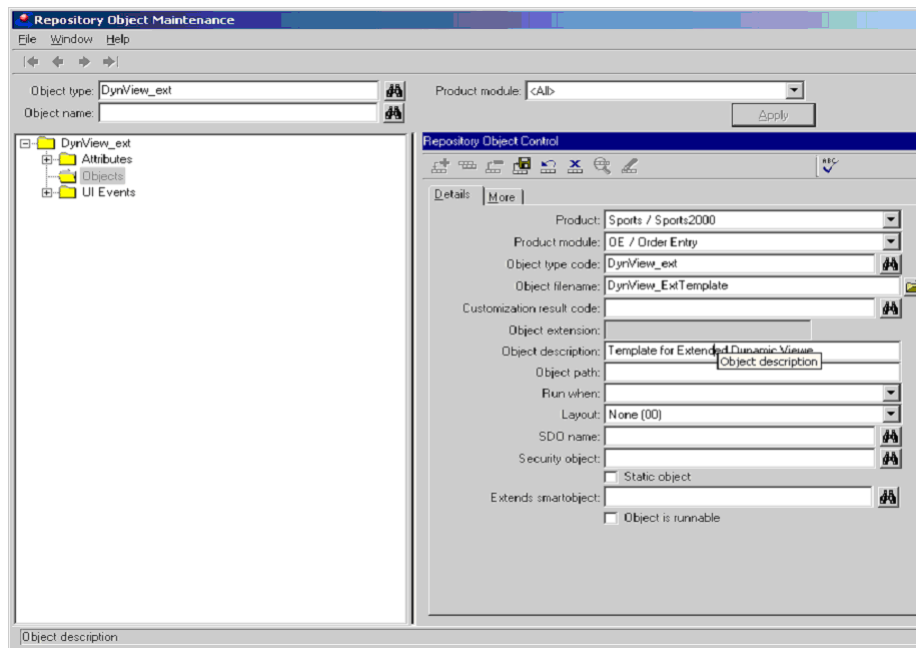
- 2 ♦ As described previously in the section on ADM customization, it is necessary to define ADM2 customization both through the adm2/custom include and in the repository in order to enable ADM2 customized procedures to run interchangeably in non-repository environments (including static containers in Dynamics) and in repository driven dynamic containers.

If your viewers fall into this category, you would need to add the new class using the New ADM Class tool in the AppBuilder which would add all supporting files for the new class . See the section on developing ADM extensions in the [Progress Dynamics Administration Guide](#) for more information.

2.7.6 Adding the new class to the AppBuilder

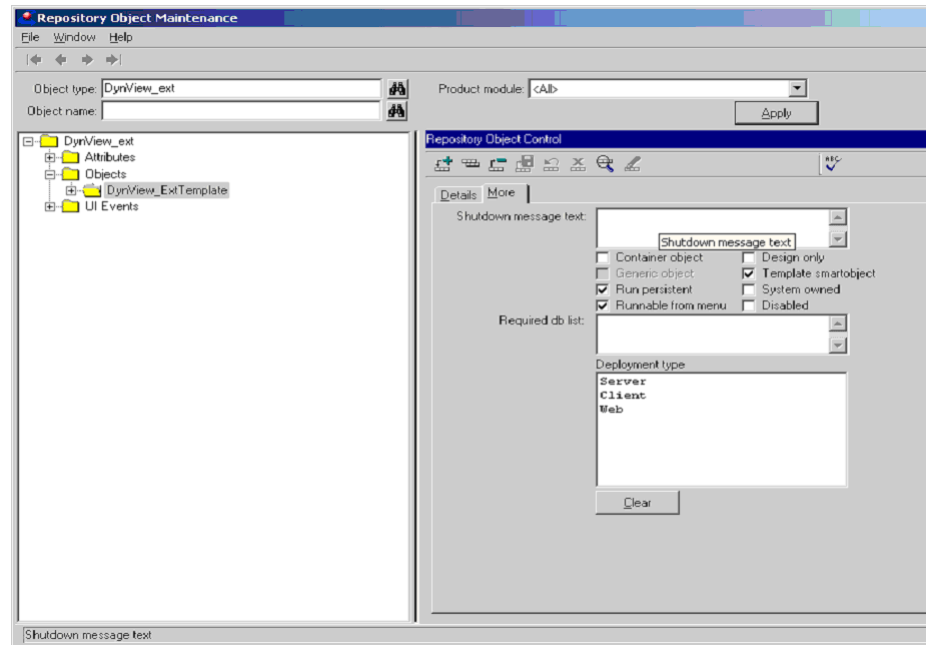
To allow the AppBuilder to recognize your newly added class, follow these steps:

- 1 ♦ From the **Repository Object Maintenance** window, enter **DynView_Ext** as the **Object Type** and press **Apply**:



- 2 ♦ Expand this to see the **Objects** node.
- 3 ♦ Right click on the **Objects** node and select **Add SmartObject**.

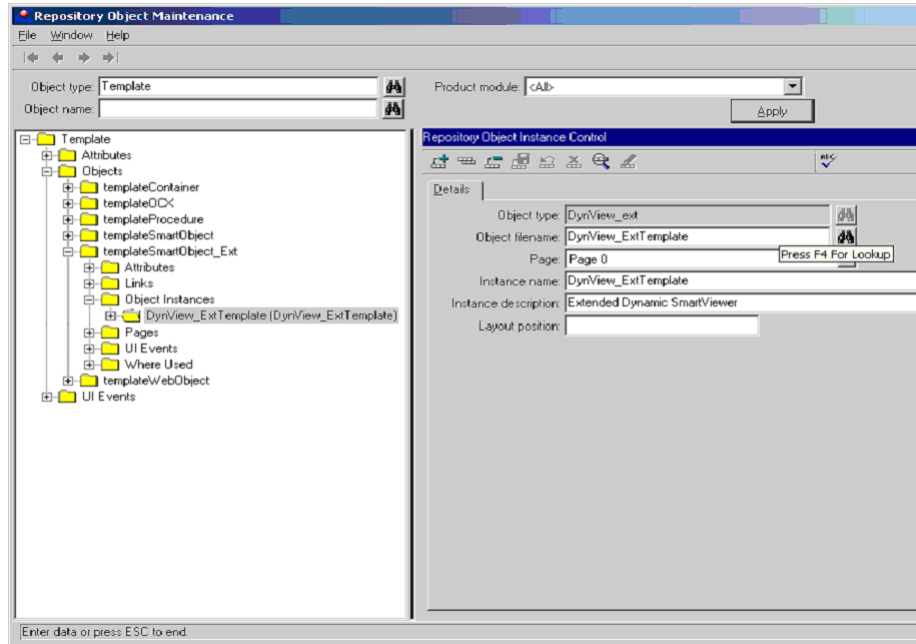
- 4 ♦ In the maintenance frame to the right, enter the following:
 - **Product** — Enter your specific product.
 - **Product Module** — Enter your specific product module.
 - **Object type code** — DynView_Ext.
 - **Object Filename** — DynView_ExtTemplate.
 - **Description** — Template for extended dynamics viewer.
 - **Layout** — None.
- 5 ♦ Select the **More** tab in the maintenance frame and check the **Template SmartObj** toggle on, which marks this object as a template. Also check the **Container Object** toggle off, since this object is not a container window:



- 6 ♦ Press the **Save** button. Now you need to create a new object in the template class and then define an Instance for that object.

CAUTION: If you want to add new instances as described below, it is important that you do not add the instance to an object shipped with the product. You must create your own object and add the instance to your object, otherwise your additions may be overridden with the next update of the product.

- 7 ♦ Enter an **Object Type of Template**, and press **Apply**:



- 8 ♦ Create an object called **TemplateSmartObject_Ext** by right clicking on the **Objects** node and select **Add Smart Object**. Enter the following:

- **Product** — Enter your specific product.
- **Product Module** — Enter your specific product module.
- **Object type code** — Template.
- **Object Filename** — TemplateSmartObject_Ext.
- **Description** — Template for AppBuilder custom SmartObjects.
- **Layout** — None.

- 9 ♦ Press the **Save** button.
- 10 ♦ Select the **Object Instances** node of the newly created object and right click and select the popup option **Add Object Instance**. Enter the following:
 - **Object Type** — DynView_Ext.
 - **Object Filename** — DynView_ExtTemplate.
 - **Instance Name** — DynView_ExtTemplate.
 - **Description** — Extended dynamic smartViewer.

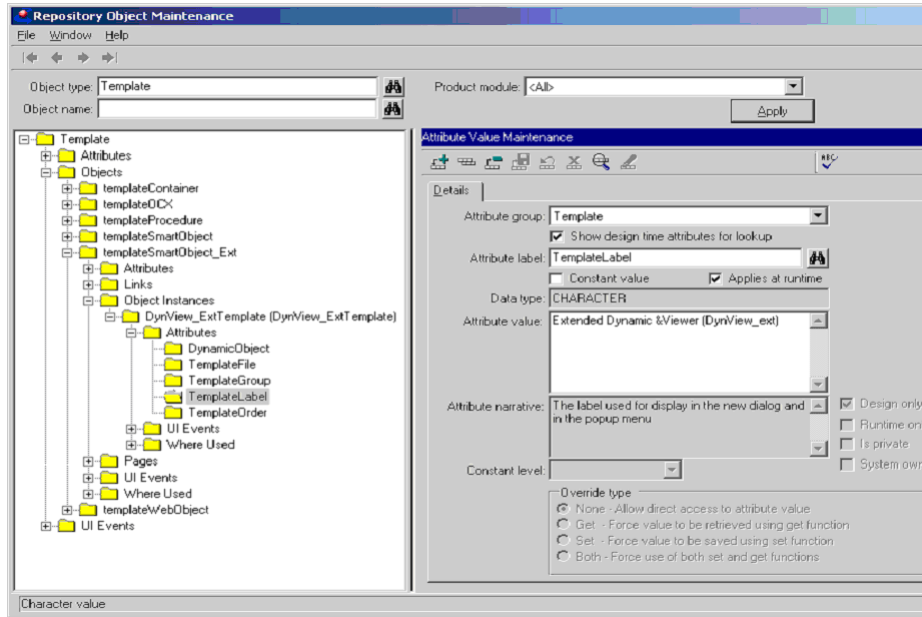
- 11 ♦ Press the **Save** button.

Next you need to define attribute values for the new instance to match the attribute values for the class template that this instance is derived from. At this time there is no way to do this automatically, so you need to look at the attribute values for the closest relative of your new instance, and enter appropriate values for the same attributes for your template instance. Follow these steps to do this:

- 1 ♦ Expand the **templateSmartObject** object
- 2 ♦ Expand the **Object Instances**, then expand the **Dynamic SmartDataViewer** node, and then expand its **Attributes**.
- 3 ♦ Expand the new extended **Viewer** and its **Attributes**. You'll see five attribute values for the Dynamic SmartDataViewer. You need to enter attribute values for your new instance to match these values:
- 4 ♦ Right click on the **Attributes** node for the new instance and add an attribute **DynamicObject** and set it to **True**. This registers your new class as a dynamic rather than a static Object.
- 5 ♦ Add an attribute value for **templateFile** of ry/obj/rydynvwizw.w. This is the procedure that is used at design time to build a new Object of the type. For a dynamic Viewer, for instance, it includes the pointers to the code that walks you through the wizard for defining a new Viewer (hence the name). You can copy this name from the attribute value for the standard dynamic Viewer.
- 6 ♦ Add an attribute value for **templateGroup** of SmartObject. There are four templateGroups (SmartObject, Container, WebObject, and Procedure), which divide the Objects into basic groups. These are used to filter the Objects in the **New** dialog based on which toggle boxes you check to see various types of Objects.

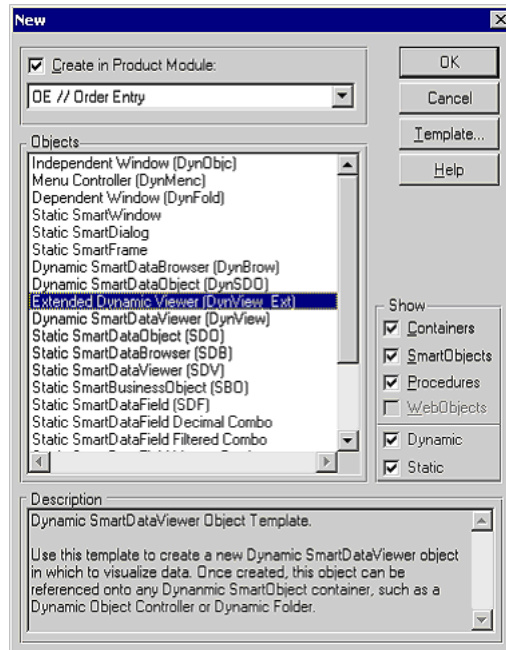
- 7 ♦ Add an attribute value for **templateLabel** of Extended Dynamic &Viewer (DynView_Ext). This matches the pattern for the label of other Object, and is what will appear for the Object in the AppBuilder's **New** dialog.
- 8 ♦ Add an attribute value for **templateOrder**. This determines the order in which the Objects are displayed in the **New** dialog. There are gaps in the numbering of existing Objects so that you can specify a **templateOrder** in between two existing Objects.

This is roughly what you should see when you have added all the new attribute values:



- 9 ♦ Exit the Repository Maintenance tool.
- 10 ♦ (IMPORTANT) Add the new object (templateSmartObject_Ext) to the list of objects defined in your icfconfig file for the tag IDETEMPLATES in a comma delimited format. Alternatively, you can go to the user preferences by selecting the menu option **File→Preferences**, and then enter the new template object as a comma delimited list in the **Custom Templates** field. (that is, *, templateSmartObject_Ext)
- 11 ♦ To rebuild the AppBuilder Palette and **New** dialog, select **Menu→Use Custom** on the **Palette** window, and press **OK**. After a few seconds the **Palette** will reappear. The contents of the **New** dialog will also be rebuilt, and you will be able to see this when you press the **New** button.

- 12 ♦ Press the **New** button to create a new Object of your new Type. You should now see your custom object in the list of templates:

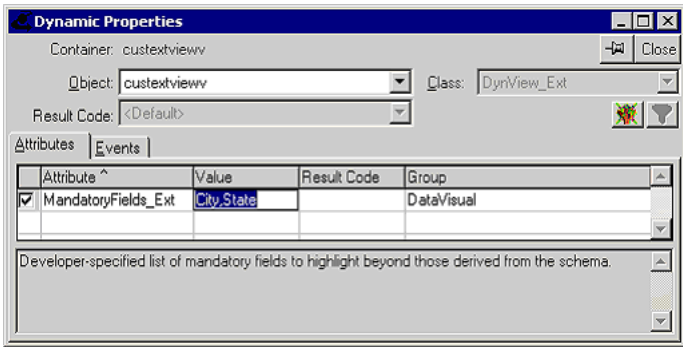


2.7.7 Building an object of your extended class

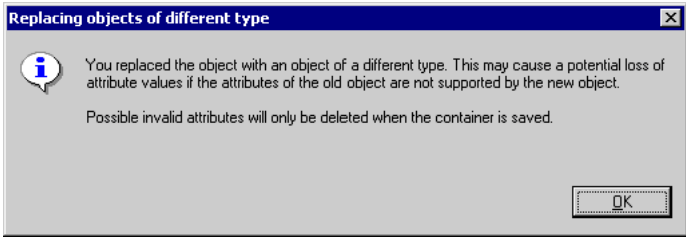
Create a Viewer based, for example, on the Customer table:

Cust Num:	0	Name:	
		Sales Rep:	
Address:			
City:			
State:		Country:	USA
Postal Code:			
Credit Limit:	1,500	Balance:	0.00

From the AppBuilder design window, select the **Dynamic Properties Sheet**. Here you see your new attribute. You can give it a value to add additional mandatory fields to be highlighted in your Viewer:



Use the Viewer to build a new window. Alternatively, you can open an existing window that uses a standard dynamic Viewer for the same table and replace the other Viewer with your new one. If you do this, you'll see a warning message indicating that you have changed the Object type:



In your case your new Object has all the attributes of the old one, and more, so replacing it will not cause problems.

Now you can run your new window to see the effects of your custom code:

The screenshot shows a window titled 'SmartDataViewer(DynView) - customerviewv'. It contains a table with the following data:

Cust Num	Country	Name	Address
1	USA	Lift Tours Inc.	276 North Drive
2	Finland	Urpon Frisbee	Rattipolku 3
3	USA	Hoops	Suite 415
4	United Kingdom	Go Fishing Ltd	Unit 2
5	USA	Match Point Tennis	66 Homer Pl
6	United Kingdom	Fanatical Athletes	20 Bicep Bridge
7	Finland	Aerobics valine Ky	Peltolantie 3

Below the table is a form with the following fields:

- Cust Num: 1
- Name: Lift Tours Inc.
- Sales Rep: HXM
- Address: 276 North Drive
- City: Burlington
- State: MA
- Country: USA
- Postal Code: 01730
- Credit Limit: 66,700
- Balance: 903.64

2.7.8 Setting your extended attributes in an object

After you've restarted your session, you can test the effects of your new attribute and its supporting code.

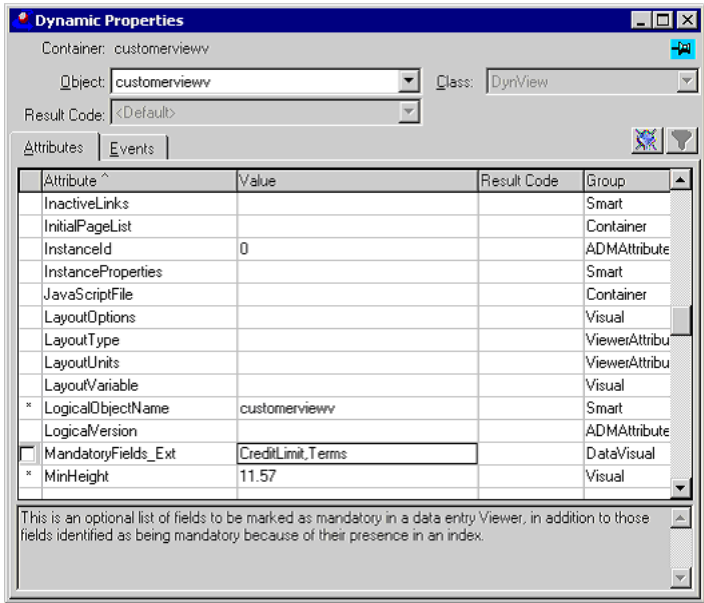
- Open a dynamic Viewer in the AppBuilder such as this one for the Customer table:

The screenshot shows a window titled 'SmartDataViewer(DynView) - customerviewv'. It contains a form with the following fields:

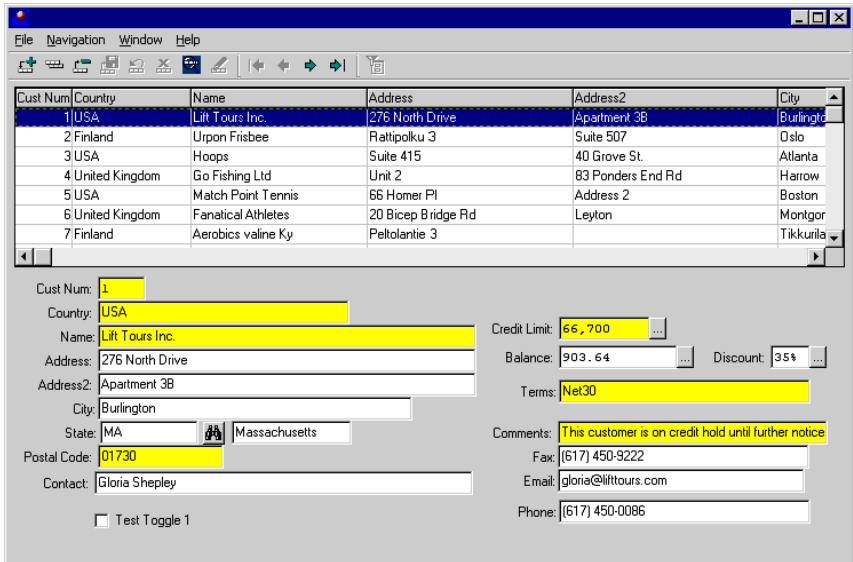
- Cust Num: 0
- Country: USA
- Name:
- Address:
- Address2:
- City:
- State:
- Postal Code:
- Contact:
- Credit Limit: 1,500
- Balance: 0.00
- Discount: 0%
- Terms: Net30
- Comments:
- Fax:
- Email:
- Phone:
- Test Toggle 1: ☐

- Select **Window→Dynamic Properties** from the AppBuilder menu. Because the attribute list this window uses is built dynamically from the contents of the repository, you should find your new attribute in the list of attributes the window displays.

- Define a value for the attribute to add one or more fields to the mandatory fields list, so that you can see the effect when you run a window containing this Viewer:



- Save the Viewer and run a window that contains it to see both types of mandatory fields, those that come from indexes and those that you added through the attribute value:



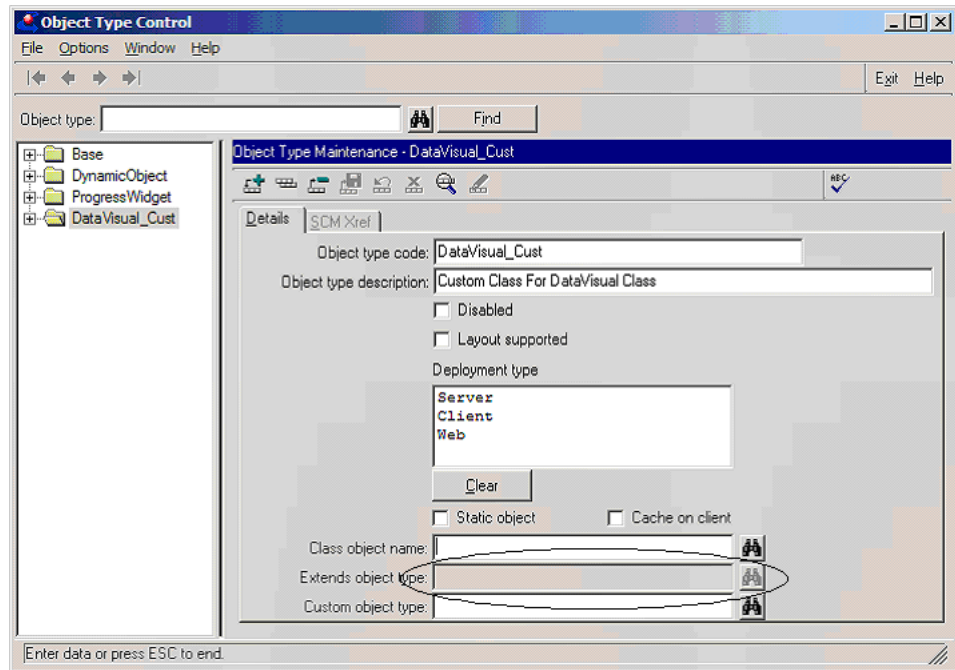
2.8 More complex customization

The following sections introduce you to techniques for advanced customization.

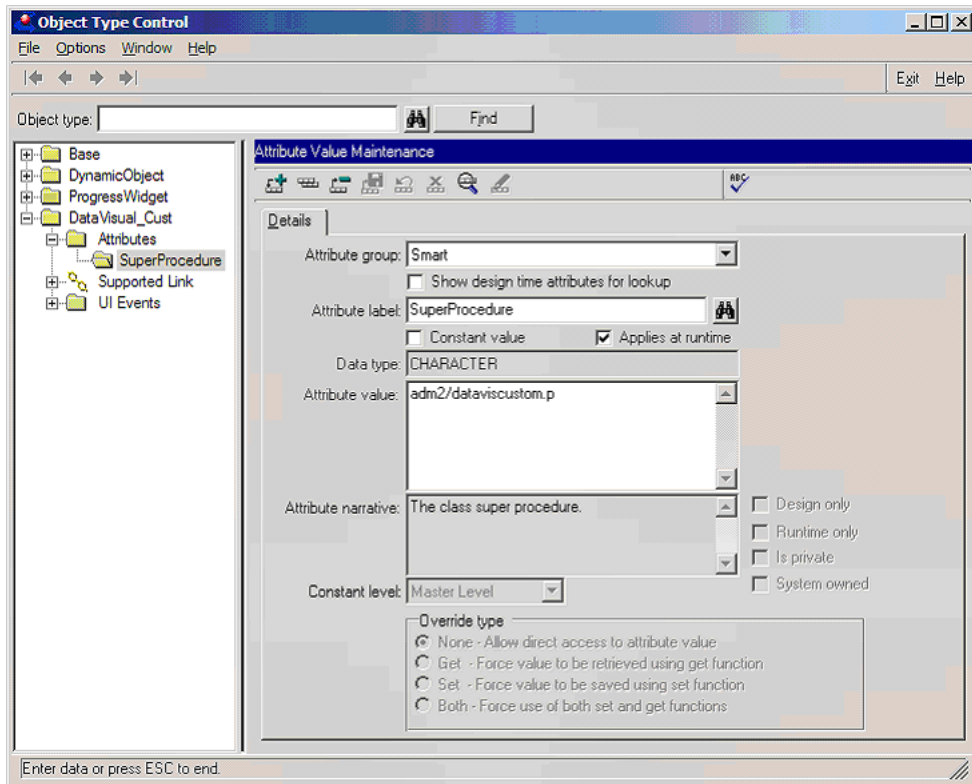
2.8.1 Customizing class behavior in the middle

To change the behavior of all the objects in a class the class needs to be extended via the Custom Object Type. The method used applies equally when extending any class—whether at the bottom or in the middle, excepting subclassing.

The first step in extending a class is to create the new customizing class. In this example, we are going to customize the DataVisual class with the DataVisual_Cust class. This figure illustrates the newly created DataVisual_Cust class. Note that this class does not extend any other class:

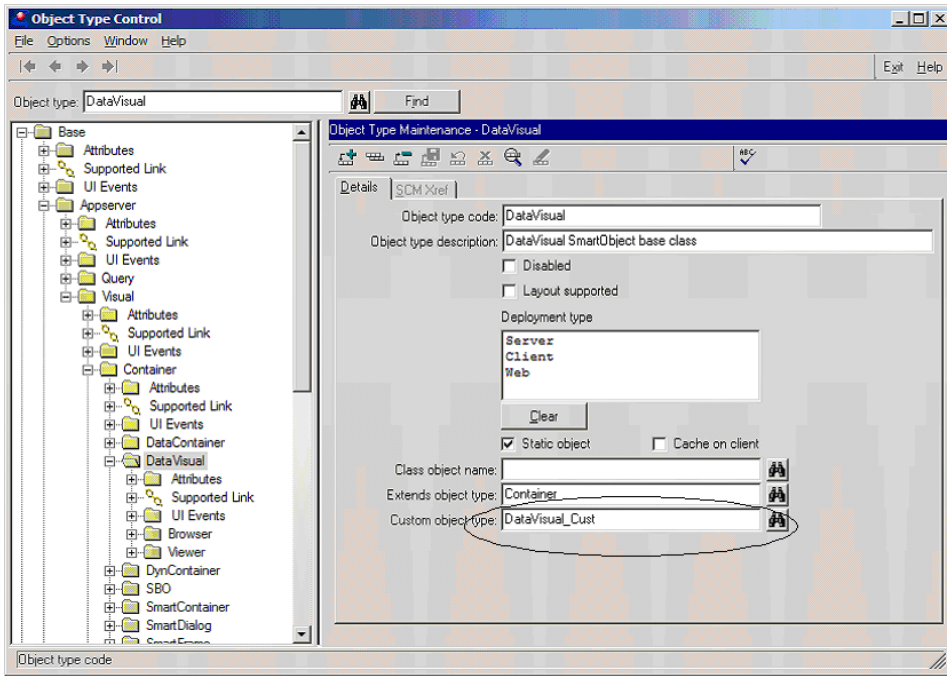


After the class has been created, we can now add attributes that will modify the behavior of the class. For example, in the figure below, we add a SuperProcedure attribute which has a value of `adm2/dataviscustom.p`. We can further add other attributes to modify the behavior of the class, as necessary. The creation of other attributes has been covered elsewhere in this tutorial and so will not be repeated:



Now we need to tell the DataVisual class that the DataVisual_Cust class customizes it. We do this by setting the value of the Custom Object Type field of the DataVisual class to DataVisual_Cust.

There is a restriction on which classes can be used as custom classes, aimed at preventing duplicate inheritance. The class being customized (DataVisual) and the class doing the customizing (DataVisual_Cust) cannot share the same ancestor classes. The Object Type Control will prevent this from happening:



Once this is done, the session should be restarted, so that the relevant caches can be flushed and repopulated. Any static caches that exist need to be re-built.

For a standard DynView-class object, the runtime class hierarchy before customization looks like:

DynView	ry/prc/rydynview.p
Viewer	adm2/viewer.p
DataVisual	adm2/datavis.p
Container	adm2/containr.p
Visual	adm2/visual.p
Base	adm2/smart.p

After customizing, the runtime class hierarchy looks like:

DynView	ry/prc/rydynviewp.p
Viewer	adm2/viewer.p
DataVisual_Cust	adm2/dataviscustom.p
DataVisual	adm2/datavis.p
Container	adm2/containr.p
Visual	adm2/visual.p
Base	adm2/smart.p

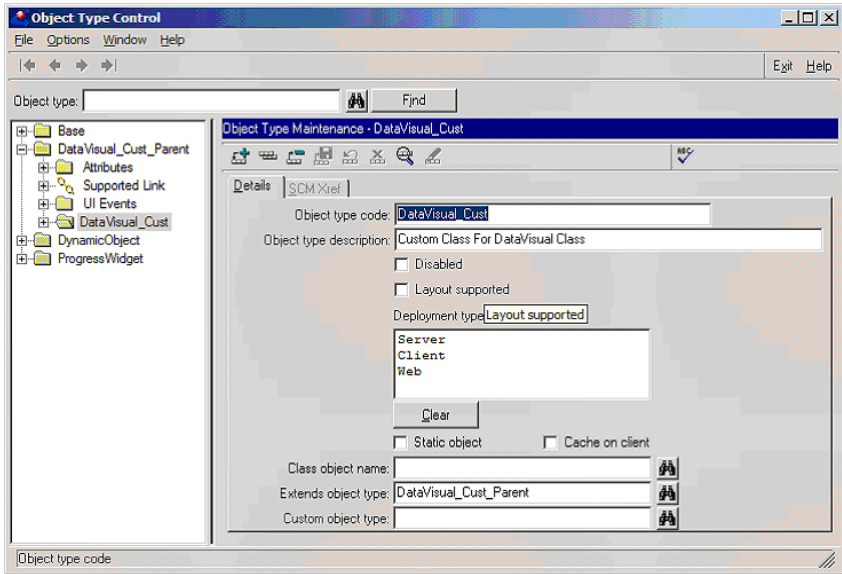
The custom class has been highlighted. Note that even though the class extension is “sideways” (that is, the DataVisual class has not been extended “downwards”) at design time, the runtime Classes that are used as custom classes should not act as object types to any objects. This is because these objects will only inherit from the custom class and not from the class being customized.

For instance, if an object were created with a class of DataVisual_Cust then it would only inherit from the DataVisual_Cust class and not from the DataVisual class that is, none of the behavior from DataVisual, Container, Visual and Base would be inherited.

If the intention is to have a group of objects have a specific behavior that differs from a particular class, then the class should be extended off the bottom, as in the next section of this tutorial.

2.8.2 Customizing a class with multiple classes

The example in the section above uses a single class to extend the DataVisual class—the DataVisual_Cust class. It is possible that the DataVisual_Cust class itself inherits from another class, and so has its own set of ancestor classes. An example of this is shown:



After customizing as shown above, the runtime class hierarchy looks like:

DynView	ry/prc/rydynview.p
Viewer	adm2/viewer.p
DataVisual_Cust	adm2/dataviscustom.p
DataVisual_Cust_Parent	-
DataVisual	adm2/datavis.p
Container	adm2/container.p
Visual	adm2/visual.p
Base	adm2/smart.p

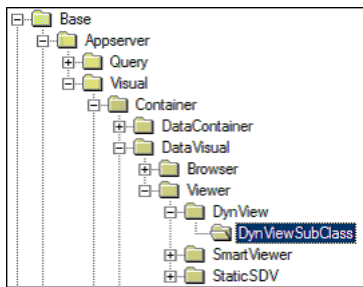
Note that the DataVisual_Cust_Parent class has no SuperProcedure property for this example. In this case there would be other attributes that provide the behavioral changes.

2.8.3 Migrating existing customizations

The steps below provide aid with the migration of extended classes from using approach of extending the existing class into using the new custom class field. In all cases examples are used.

Simple subclassing

If the DynView class is subclassed by the DynViewSubClass and not all objects belonging to the DynView class have been moved to the subclass, then nothing needs to be done. subclassing basically means that a new set of behavior is applied for certain objects belonging to a class only—not all objects in that class. The next section details the treatment for a subclass that is used to extend the behavior of all objects in the DynView class:

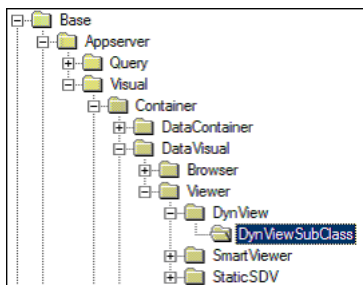


Customizing off the bottom of the class hierarchy

If the DynView class has been extended so that the behavior of the DynViewSubClass is meant to apply to all the objects that belong to the DynView class, then the following steps need to be taken.

The fundamental distinction between this style of extending and the subclassing mentioned above is that the behavior of all objects belonging to that class changes when extending the class, and not simply a subset of objects.

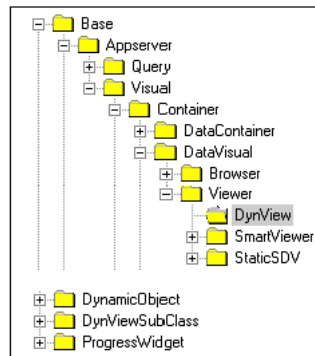
Before upgrade, the class structure looks like this:



To migrate:

- 1 ♦ Upgrade icfdb with the DCU and load ADOs.
- 2 ♦ From the **Object Type Control Options** menu, run the **Object Type Change Utility**.
- 3 ♦ Select all objects for the DynViewSubClass and change these to **DynView**.
- 4 ♦ Select the **DynViewSubClass** class.
- 5 ♦ Clear (blank) the **Extends Object Type** field and save.
- 6 ♦ Either restart the session or run destroyClassCache in gshRepositoryManager to clear all cached classes.
- 7 ♦ Run the **Object Type Control** tool.
- 8 ♦ Drill down to the **DynView** class.
- 9 ♦ Set the value of the **Custom object type** field to **DynViewSubClass**.
- 10 ♦ Repeat step 5.

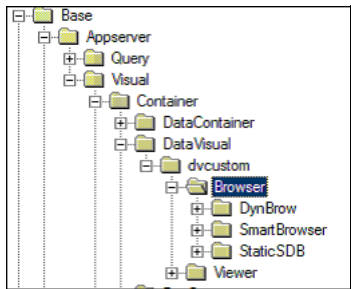
The resulting class structure looks like this:



Customizing in middle with a single custom class

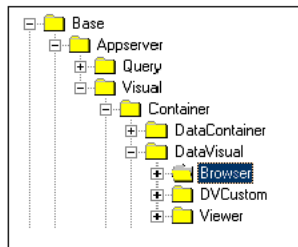
For a class hierarchy where the DataVis class is extended by the DVCustom class, and there are no objects associated with the DVCustom class.

Before upgrade, the class structure is:



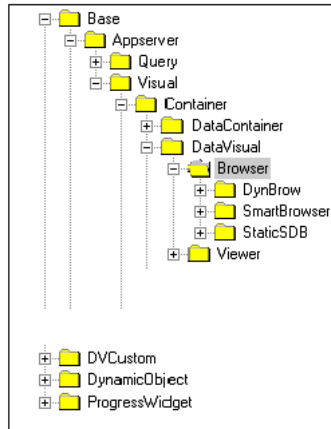
To migrate:

- 1 ♦ Upgrade icfdb with the DCU and load ADOs, this will override customizations making DVCustom, Browser and Viewer all inherit from DataVisual:



- 2 ♦ Follow these steps for any objects that may exist for the custom class:
 - a) From the **Object Type Control Options** menu, run the **Object Type Change Utility**.
 - b) Select all objects for the **DVCustom** class and change these to **DataVisual**. While it is unlikely that DVCustom objects exist, there could be objects for custom classes extended in the middle that would need to change.
- 3 ♦ In the **Object Type Control** tool, select **DVCustom** class.
- 4 ♦ Clear the **Extends Object Type** field and save.
- 5 ♦ Drill down to the **DataVisual** class.
- 6 ♦ Set the value of the **Custom object type** field to **DVCustom**.
- 7 ♦ Either restart the session or run `destroyClassCache` in `gshRepositoryManager` to clear all cached classes.

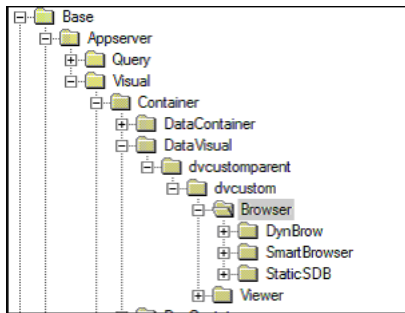
The resulting class structure is:



Customizing in middle with multiple custom classes

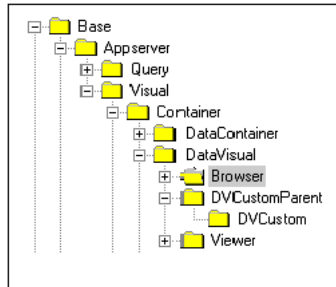
For a class hierarchy where the DataVis class is extended by the DVCustomParent and DVCustom, classes, the following steps apply.

Before upgrade, the class structure looks like this:



To migrate:

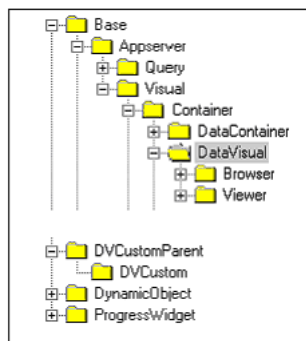
- 1 ♦ Upgrade icfdb with the DCU and load ADOs, this will override customizations making DVCustomParent, Browser and Viewer all inherit from DataVisual:



- 2 ♦ Follow these steps for any objects that may exist for the custom class:
 - a) From the **Object Type Control Options** menu, run the **Object Type Change Utility**.
 - b) Select all objects for the **DVCustom** class and change these to **DataVisual**.
 - c) Select all objects for the **DVCustomParent** class and change these to **DataVisual**. While it is unlikely that DVCustom and DVCustomParent objects exist, there could be objects for custom classes customized in the middle that would need to change.

- 3 ♦ In the **Object Type Control** tool, select the **DVCustomParent** class.
- 4 ♦ Clear the **Extends Object Type** field and save.
- 5 ♦ Drill down to the **DataVisual** class.
- 6 ♦ Set the value of the **Custom object type** field to **DVCustom**.
- 7 ♦ Either restart the session or run `destroyClassCache` in `gshRepositoryManager` to clear cached classes.

The resulting class structure is:



Advanced User Interface Design In Progress Dynamics

A typical Progress Dynamics application has windows with many objects on them, including Toolbars with a variety of buttons and menus, and multi-page tab folders with interrelationships between the objects on different pages. The relationships between the objects are determined by SmartLinks™, by properties, and by various functions that control the interactions. Progress Dynamics Version 2 has many features that support building complex windows and customizing behavior to suit the needs of your application. This chapter describes some of those features and the techniques you can use to take advantage of them.

This chapter describes a single folder window that illustrates many of these features. It is a multi-page window called **oemaintwin** (for Order Entry Maintenance Window), designed to display and update Customers, Orders, and OrderLines in the Sports2000 database. You can assemble the window yourself to follow along with the various examples in the chapter. In describing the window, we presume that you have created a Sports product with an OE product module for Order Entry, done an Entity Import of at least the Customer, Order, OrderLine, Item, and SalesRep tables, and used the Object Generator to create a suite of dynamic objects (SDOs, DataFields, Viewers, and Browsers) for those tables in the OE product module. Refer to the [*Progress Dynamics Developer's Guide*](#) for guidance on any of these preparatory steps, and also for detailed information on using the Container Builder. To create the window, start with a dynamic Independent Window (DynObjc), referred to as independent because the window does not expect to receive a key value from another window when it starts up. In general, **all** the objects you use to create the window can be dynamic objects unless otherwise noted.

The following sections describe how to assemble the pieces of the window and how the window illustrates some of the chapter's material on User Interface design:

- [Advanced support for object links](#)
- [Modifying visual properties at the master and instance level](#)
- [Disabling and hiding buttons and menu items](#)
- [Defining action rules for menu and toolbar items](#)

3.1 Advanced support for object links

Communication between the SmartObjects that make up a Progress Dynamics application window is controlled by SmartLinks, or Links for short. For more information on the basic ADM2 Links and the named events that they handle, see the standard Progress Version 9 documentation. To learn more about the new Links and events and how they are used by Progress Dynamics see the *Progress Dynamics Developer's Guide*. This section describes some of the new features involving Links and shows ways to use them in building your application, including:

- [Using a single toolbar in a multi-page folder](#)
- [Setting up your test window](#)
- [Modifying the default link activation](#)
- [Disabling data links to dependent SDOs](#)
- [Using a GroupAssign link to group pages](#)

3.1.1 Using a single toolbar in a multi-page folder

One significant improvement in the area of Links is that you can use a single Toolbar to control objects on any number of pages in a tab folder. For example, a Toolbar with one of the Update bands of buttons for Add, Modify, and Delete can be the TableIO-Source for any number of Viewers or other updatable objects in a tab folder. The state of the buttons and the equivalent menu items switches automatically as the user selects one page and then another, and the actions on one page properly affect the state of the buttons and menu items on other pages. In addition, a Toolbar with a Navigation band can be the Navigation-Source for SDOs on any number of pages in a folder. The state of the buttons changes to reflect the query position of the SDO on each page.

3.1.2 Setting up your test window

To illustrate the toolbar's ability to control objects on multiple pages, you need to create several pages in the window for the Toolbar to control, including:

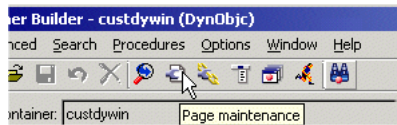
- [Toolbar and folder on page 0](#)
- [Customer objects on page 1](#)
- [Customer maintenance viewer on page 2](#)
- [Order maintenance objects on page 3](#)

- [Defining the Foreign Fields property for a child SDO](#)
- [OrderLine maintenance objects On Page 4](#)
- [Modifying the resize attributes of a browser](#)
- [Summary of all the links for the window](#)

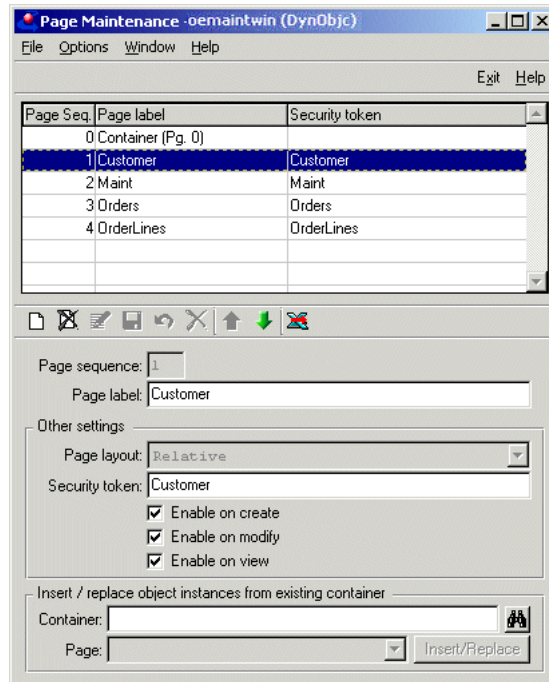
Toolbar and folder on page 0

Start on Page 0 of the window and follow these steps:

- 1 ♦ Use the FolderPageTop Toolbar, which has Update and Navigation buttons and also a standard File menu. There are in fact a variety of different built-in Toolbars that you can investigate using the Toolbar and Menu Designer, or you can build a new one of your own.
- 2 ♦ Add an instance of the dynamic Folder object, called afspfoldw.w. When you add the Folder to the window in the Container Builder, the tool creates a Page link automatically from the Folder to the window itself, since this is required for the Folder to function properly.
- 3 ♦ In the Container Builder, choose the **Page Maintenance tool**:



- 4 ♦ In the Page Maintenance window add four folder tabs, with the labels **Customer**, **Maint**, **Orders**, and **OrderLines**:



Customer objects on page 1

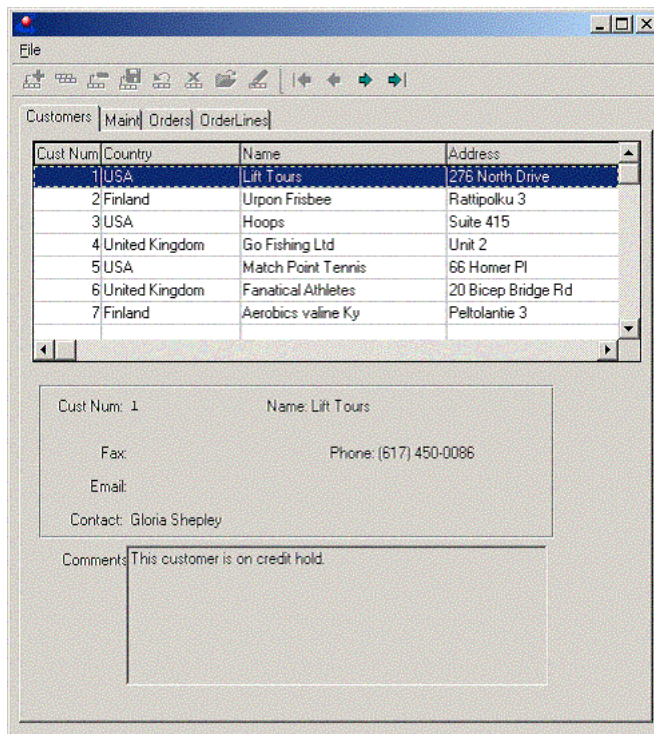
To add Customer Objects on Page 1 of the window, follow these steps:

- 1 ♦ Add a Customer SDO, such as the customerfull0 dynamic SDO that the Object Generator created for you.
- 2 ♦ Add a dynamic Customer Browser such as the customerfullb that the Object Generator created, where the user can select a Customer to work with. There is also a Viewer under the Browser to display a selection of fields that may be helpful. You can build this dynamic Viewer in the AppBuilder, selecting just the fields you want to see. In the example we call this Viewer custcommentsv, since it features the Customer Comments field.

3 ♦ Add the following links:

- A Navigation link from the Toolbar to the Customer SDO on Page 1.
- A Data link from the SDO to the Browser on Page 1.
- A Data link from the SDO to the Viewer on Page 1. The Viewer is just for data display purposes, so there is no TableIO link to it and no Update link back to the SDO.

At this point, the window, which comes up with Page 0 and Page 1 displayed, should look like this:



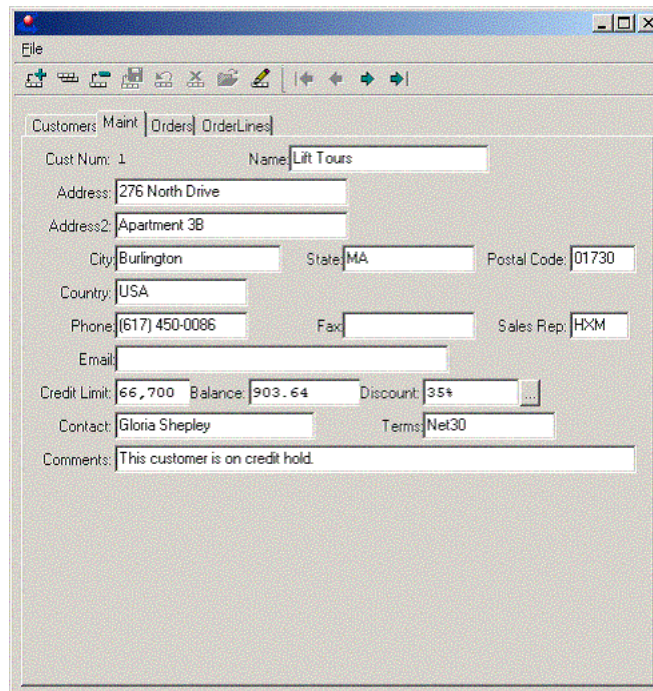
Note the state of the Toolbar buttons. The Next and Last Navigation buttons are enabled to show that the query is now positioned on the first record of the dataset. The Update buttons are all disabled because there is no Object on the page that can do an Update (and no TableIO Link to communicate those Update events through).

Customer maintenance viewer on page 2

To add a Customer Maintenance Viewer, follow these steps:

- 1 ♦ Add another Customer Viewer to Page 2 of the window. Since this is for Customer Maintenance, you can start with the dynamic `customerviewv` that the Object Generator created, open it in the AppBuilder, and rearrange the fields to suit your needs.
- 2 ♦ Add the following links:
 - A Data link from the Customer SDO to this Viewer.
 - A TableIO link from the Toolbar to the Viewer.
 - An Update link from the Viewer to the SDO, so that users can add and maintain Customers using the Viewer.

Page 2 of the window looks like this:



The screenshot shows a window titled "File" with a menu bar and a toolbar. The main area contains a form for editing customer information. The form has tabs for "Customers", "Maint", "Orders", and "OrderLines". The "Customers" tab is selected. The form fields are as follows:

Cust Num:	1	Name:	Lift Tours
Address:	276 North Drive		
Address2:	Apartment 3B		
City:	Burlington	State:	MA
Postal Code:	01730		
Country:	USA		
Phone:	(617) 450-0086	Fax:	
Sales Rep:	HXM		
Email:			
Credit Limit:	66,700	Balance:	903.64
Discount:	35%	...	
Contact:	Gloria Shepley	Terms:	Net30
Comments:	This customer is on credit hold.		

Note that because the Viewer is a TableIO-Target for the Toolbar, and an Update-Source for the SDO, the appropriate Update buttons in the Toolbar are enabled.

Order maintenance objects on page 3

Page 3 of the window is for viewing and editing Orders. To add Order Maintenance Objects, follow these steps:

- 1 ♦ In the Container Builder, place the Order SDO `orderfullo`, the dynamic Browser `orderfullb`, and the dynamic Viewer `orderviewv` onto Page 3.
- 2 ♦ Edit the Viewer in the AppBuilder to arrange the fields more appealingly.
- 3 ♦ Add the following links:
 - A Data link from the Customer SDO on Page 1 to the Order SDO on Page 3.
 - A Data link from the Order SDO to the Order Browser.
 - A Data link from the Order SDO to the Order Viewer.
- 4 ♦ To enable the Toolbar support for this Page, define the following links:
 - A TableIO link from the Toolbar to the Order Viewer.
 - An Update link from the Order Viewer to the Order SDO.
 - A Navigation link from the Toolbar to the Order SDO.

Defining the Foreign Fields property for a child SDO

When you create a Data link from Customer to Order, you must set the `ForeignFields` property of the Order SDO to identify which key gets passed into the Order SDO to qualify its query. To define the foreign fields property, follow these steps:

- 1 ♦ Select the **Order SDO** in the Container Builder.
- 2 ♦ To the immediate right of the Foreign Fields editor, click the button to bring up the Foreign Fields Mapping dialog box. This dialog allows you to quickly map sources and targets,
- 3 ♦ Map `Order.Custnum` to `Custnum`. This tells the framework that the Order query must be modified at runtime to insert the phrase `Order.CustNum = <CustNum>`, where `<CustNum>` represents the current value of the `CustNum` field in the parent Customer SDO.

4 ♦ Page 3 of your window should look like this:

The screenshot shows a window titled "OrderLines" with a menu bar (File) and a toolbar. Below the toolbar are tabs: Customers, Main, Orders, and OrderLines. The OrderLines tab is active, displaying a table with the following data:

Order Num	Cust Num	Ordered	Shipped	Promised	Carrier
6	1	02/11/98	02/09/1999	02/16/98	Standard Mail
36	1	09/27/97	10/02/1997	10/02/97	FlyByNight Courier
79	1	11/21/97	11/26/1997	11/26/97	Standard Mail
177	1	03/02/98	03/07/1998	03/07/98	Standard Mail
185	1	11/26/97		12/01/97	Walkers Delivery
1335	1	02/04/98	02/09/1998	02/09/98	UPS Ground
1340	1	09/19/97		09/24/97	UPS Ground
1350	1	01/01/98		01/06/98	UPS Ground
1390	1	10/16/97		10/21/97	UPS Ground
6050	1	01/29/98		02/03/98	

Below the table is a form with the following fields:

Order Num: Cust Num: Sales Rep:

Ordered: Promised: Shipped:

Bill To ID: Ship To ID: Carrier:

Order Status: PO: Terms:

Instructions:

Credit Card: Warehouse Num:

OrderLine maintenance objects On Page 4

Page 4 is the OrderLine page. To define OrderLine maintenance objects, follow these steps:

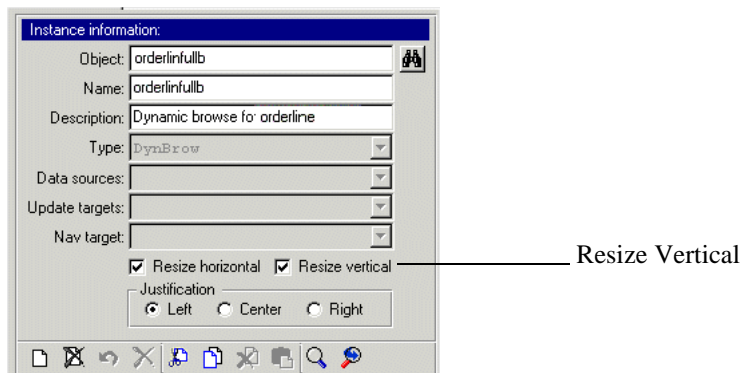
- 1 ♦ Place the OrderLine SDO **orderlinfullo**, the OrderLine Browser **orderlinfullb**, and the OrderLine Viewer **orderlinviewv** on this page.

- 2 ♦ Edit the Viewer in the AppBuilder to rearrange the fields.
- 3 ♦ Define these links:
 - A Data link from the Order SDO to the OrderLine SDO.
 - A Data link from the OrderLine SDO to the OrderLine Browser.
 - A Data link from the OrderLine SDO to the OrderLine Viewer.
 - A TableIO link from the Toolbar to the Viewer.
 - An Update link from the Viewer to the SDO.
 - A Navigation link from the Toolbar to the OrderLine SDO.
- 4 ♦ Set the OrderLine SDO's ForeignFields to **OrderLine.OrderNum,OrderNum.**

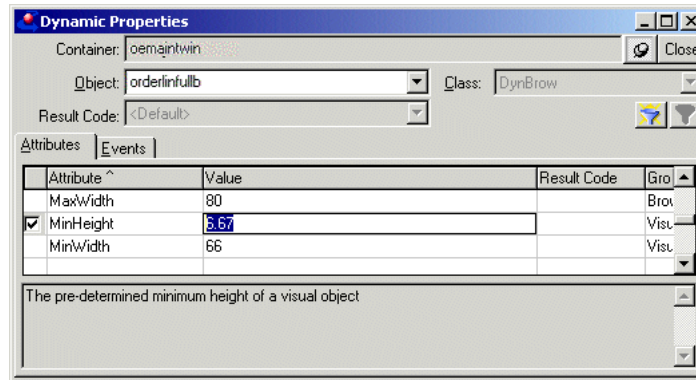
Modifying the resize attributes of a browser

Page 4 is sized to fit the overall size of the largest page when you run the window. Because there are typically only a few OrderLines per Order, the Browser might be taller than it needs to be, since by default it is sized to take up all available space on the page after the Viewer, which is a fixed size, has been placed at the bottom. To change this layout behavior, follow these steps:

- 1 ♦ Make sure that the Resize Vertical toggle is **off** in the Container Builder for the OrderLine Browser. This leaves it at its initial size regardless of the overall size of the Folder:

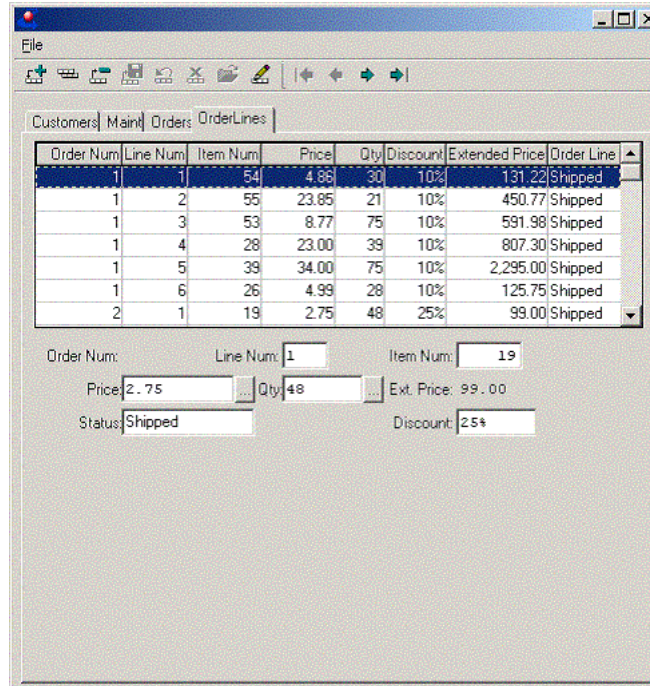


- 2 ♦ Set the **MinHeight** attribute for the Browser in the dynamic property sheet to change what that initial height is:



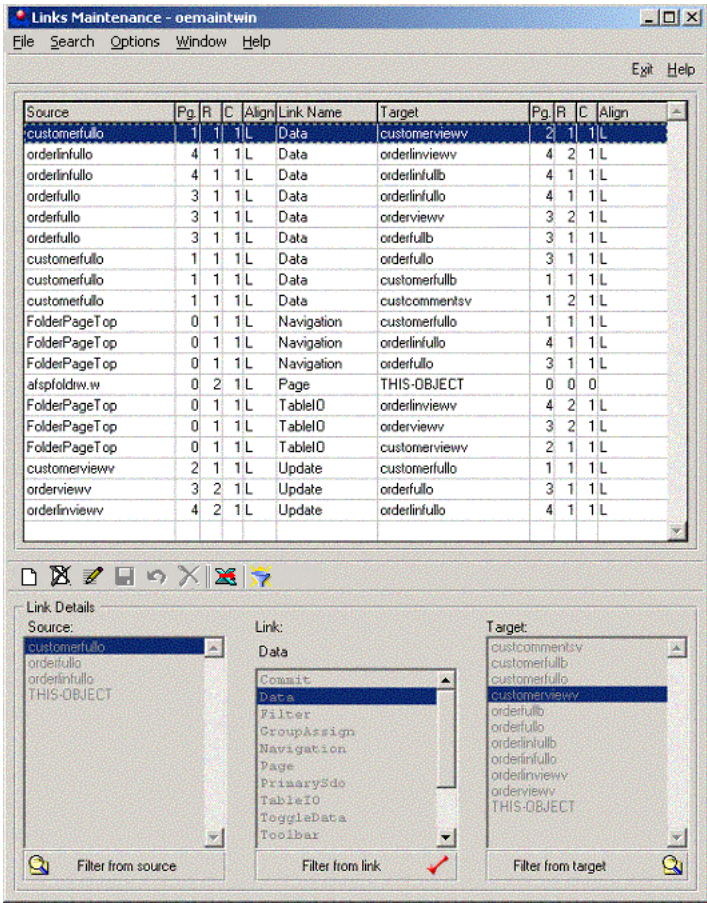
NOTE: You can also modify the MinWidth and ResizeHorizontal attributes. You can set the ResizeHorizontal attribute to False and set the MinWidth of the Browser in order to fix its width in the window, so that it is always just the right size to display its fields.

With the changes to MinHeight and ResizeVertical, Page 4 looks like this:



Summary of all the links for the window

The Container Builder’s Link Editor displays a summary of all the links you should have when you have completed these tasks:



3.1.3 Modifying the default link activation

As noted, the single Toolbar correctly changes its state to reflect which page is the active page. When the window first comes up, the Update buttons are disabled because there is no updatable object on that page. When you select Page 2, 3, or 4, the buttons are enabled.

However, if you select Page 1 again, the buttons remain enabled, which is not correct. This happens because of the setting of the Toolbar property `DeactivateTargetOnHide`. Navigation and TableIO links need to be adjusted when the page changes and there is more than one target for the link. These links cannot support more than one active target at a time, since the Toolbar needs to reflect the state of a single Object. The `DeactivateTargetOnHide` property determines how the Toolbar handles that adjustment. By default, the property value is `False`, so when a page containing a TableIO or Navigation target for a Toolbar is hidden by a page change operation, the Toolbar waits until another page with an Object that uses the same link is viewed before re-evaluating the state of the buttons. If every page in the folder has a TableIO link, then as each new page is viewed, the Toolbar resets its button state for each new target. In this case the default behavior works fine.

But if there is a page with **no** TableIO-Target for the Toolbar, then nothing happens when the user selects that page, and the state remains set to what it was for the previously selected page. This is normally not appropriate.

In this case you need to set the `DeactivateTargetOnHide` property to `True`, so that the Toolbar resets its buttons immediately when a page is hidden. In this way, they remain disabled when the user selects Page 1 after Page 2 because there is no TableIO-Target on Page 1. The downside of this, and the reason why this is not the default behavior, is that this may result in some flashing as the buttons are disabled when one page is hidden and then (perhaps) immediately re-enabled when the next page is viewed.

When you add a Toolbar to a static Window, there is a custom property sheet, shown in [Figure 3-1](#), that you can access through the AppBuilder and where you can set the property.

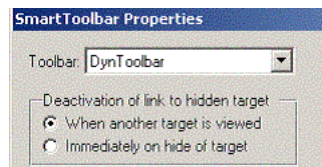


Figure 3-1: Smart Toolbar Properties window

For dynamic Windows, you can bring up the dynamic property sheet, shown in [Figure 3–2](#), for the Toolbar in the Container Builder and set the property there.

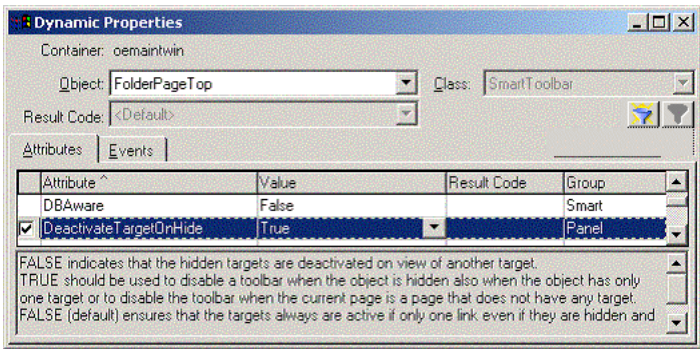


Figure 3–2: Dynamic Properties window

After you do this, the Update buttons are always enabled when you select Page 1.

To clarify something about this behavior, bring up your test window, and select Page 2. The Customer Maintenance Viewer is displayed, and the Update buttons are enabled, as they should be. Notice that the Navigation buttons are also enabled, which might not seem correct. Given that you just set the DeactivateTargetOnHide property to True, and there is no Navigation-Target on Page 2, you might expect that the Navigation buttons would be disabled when you select Page 2. What happens, however, is that the Toolbar code detects that there is an object on Page 2 that is a Data-Target of the SDO (on Page 1) that is the Navigation-Target of the Toolbar. For this reason the standard behavior is to leave the Navigation buttons enabled and allow them to navigate through the Customer records from Page 2 exactly as you could from Page 1. If you want to change this behavior, you can use the DisabledActions Toolbar property that is described in the [“Disabling and hiding buttons and menu items”](#) section.

3.1.4 Disabling data links to dependent SDOs

Another situation that often arises with multi-page windows is that you have a series of SDOs (parent-child-grandchild) on a sequence of pages. When the user views Page 1 and the parent data, by default the Data links between the SDOs send the dataAvailable event on to the next SDO, causing it to open its query using the ForeignFields keys for the currently selected record in the master. You may or may not want this to happen. There is obviously an overhead associated with it, and as the user selects one record or another on the page for the master SDO, it may be useless to retrieve dependent data on another page until it is actually viewed.

You can control this behavior by using the SDO's `ToggleDataTargets` attribute. The attribute is `True` by default, so that behavior is optimized by effectively making the link inactive. What actually happens is that the code unsubscribes the Data-Target to the events that activate the Target to open its query, including `dataAvailable`. The link itself is still there and can be used for other purposes.

To force dependent SDOs to retrieve dependent data immediately, set the `ToggleDataTargets` attribute to `False` for the master SDO (and any other SDOs that have dependent SDOs under them) in the Container Builder property sheet. You can also set this value in the Dynamic property sheet, or you can bring up the object properties for the SDO and unselect the **Activate/deactivate Data Triggers on view/hide**.

NOTE: The `ToggleData SmartLink` used in earlier versions of the framework for much the same purpose as the `ToggleDataTargets` property is no longer actively supported. Use the `ToggleDataTargets` property instead.

3.1.5 Using a GroupAssign link to group pages

The Toolbar also keeps track of whether a GroupAssign link relates Viewers on different pages of a Folder. To allow an update of many fields to be divided between two or more pages, you can place a Viewer on each page for different groups of fields in the same SDO. You then need to create the following Links:

- A Data link from the SDO to each of the Viewers.
- A GroupAssign link from the master Viewer, the first in sequence, to each of the other Viewers.
- A TableIO link only to the master Viewer.
- An Update link from only the master Viewer back to the SDO.

All the supporting code treats the multiple Viewers as part of a single display and update mechanism, and the Toolbar does the same.

As an example, you can add a copy of the `custcommentsv` Viewer that appears on Page 1, or any other Customer Viewer with a subset of the fields, onto a new Page after the Maint page. In the Container Builder, go into Page Maintenance and add a new Page 3 for Comments, pushing the Order and OrderLines to pages 4 and 5. Drop another instance of the `custcommentsv` viewer on page 3, and define these Links:

- A Data link from the Customer SDO to the newly added Viewer.
- A GroupAssign link from the master Customer Viewer `customerviewv` to the new Viewer.

Now when you save and launch the window, you can start an Add or Update on Page 2 and continue it on Page 3, or vice-versa. When you select Page 3, as shown in [Figure 3–3](#), the Update buttons are enabled and have the same state as for Page 2, even though there is no TableIO link to that page at all.

The screenshot shows a software application window with a menu bar containing 'File'. Below the menu bar is a toolbar with various icons. The main area of the window has a tabbed interface with tabs labeled 'Customers', 'Main', 'Comments', 'Orders', and 'OrderLines'. The 'Main' tab is currently selected. Inside the 'Main' tab, there is a form with several input fields: 'Cust Num' with the value '1', 'Name' with 'Lift Tours', 'Fax' with '(617) 450-9876', 'Phone' with '(617) 450-0086', 'Email' with 'gloria@littours.com', and 'Contact' with 'Gloria Shepley'. Below these fields is a 'Comments' section with a text area containing the text 'This customer is on credit hold until further notice.'

Figure 3–3: GroupAssign link example

This is because the Toolbar recognizes the Viewer on Page 3 as being part of a grouped update controlled by the Viewer on Page 2.

When you are through experimenting with this, you can delete this Page 3, because it is not used in the remaining examples.

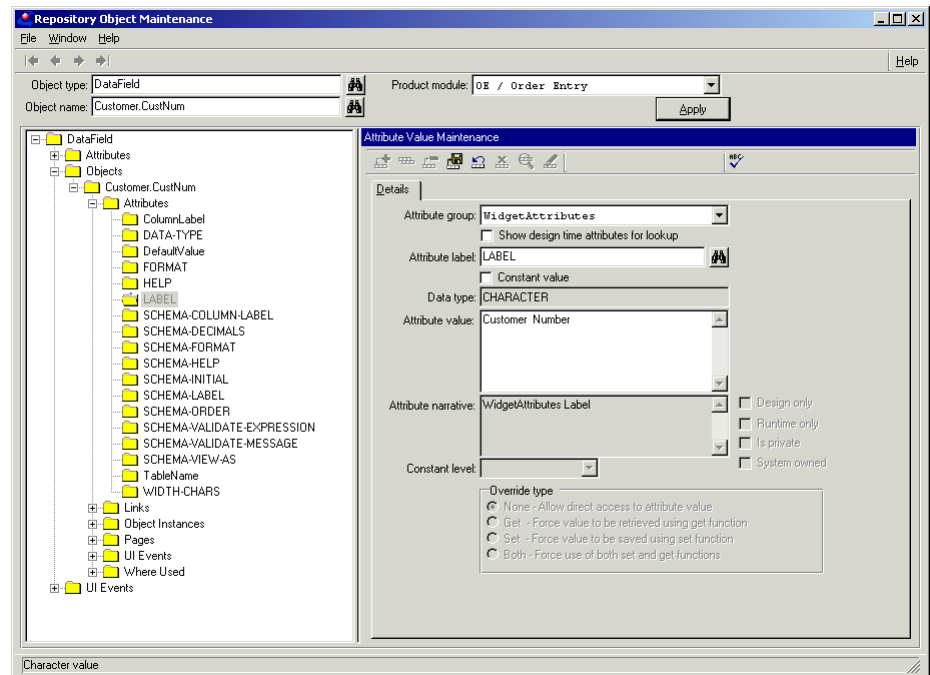
3.2 Modifying visual properties at the master and instance level

When you first edit the `customerviewv` Viewer and test it in your window, note that some of the fields have a button on the right containing an ellipsis (...). This button brings up a popup object to calculate a value for the field. Generally, by default all numeric fields have a Calculator popup, and all date fields a Calendar popup. If you want to turn this off for a field like the `CustNum`, because it is not appropriate for that field, you can do this globally by changing that property in the `DataField` Object. Because of the inheritance model in Progress Dynamics, that property setting for the Master field-level Object is inherited by all uses of that field in Viewers and other Objects unless it is specifically set otherwise.

3.2.1 Setting a property at the master level

Follow these steps to use the Repository Maintenance tool to set a property for a `DataField` to apply to all uses of that field:

- 1 ♦ Open the **Repository Object Maintenance** tool from the AppBuilder's Build menu and enter the name of the field, which is always qualified by the database table name it is derived from:



- 2 ♦ Expand the Objects node for the field in the tree view, then the field node itself, and then the Attributes node. Here you see all the properties for the `DataField` that are predefined.

- 3 ♦ Select the **Label** attribute.
- 4 ♦ Press the **Modify** button in the Maintenance frame, and set its value to Customer Number by typing in the Attribute Value field.

When you run any container with the CustNum field in it, such as the window you are building now, it reflects this change, unless the CustNum value has explicitly been set in the Viewer or the container window.

3.2.2 Setting properties at the instance level

You can also set field properties for a particular Viewer when you edit the Viewer in the AppBuilder. When you do this, you are editing the properties at the Instance level, because you are setting the property only for this particular instance of the field, when it is displayed in this particular Viewer.

The AppBuilder provides two ways of setting field (or other widget) attributes. If you double-click on a field, you see the same property sheet that you have always used in the AppBuilder, customized to show the appropriate attributes for the field or widget type (Fill-In, Editor, Button, etc.). When you change attributes in these property sheets, the AppBuilder saves them to the repository if you are editing a dynamic Object. For instance, [Figure 3–4](#) shows the property sheet for the CustNum field in the customerviewv Viewer.

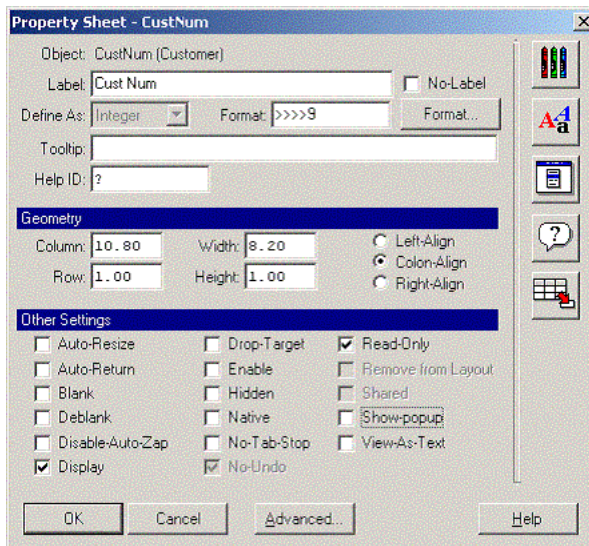


Figure 3–4: Property sheet for the CustNum field

Here you can disable the field and set other properties just as you always could. In this case we have disabled the field by setting the Enable property to False and the Read-Only property to True. A few of these properties, such as Show-Popup, are new with Progress Dynamics, but otherwise the property sheets are as they have always been.

3.2.3 Using the Dynamic property sheet to set properties

You can also set properties in the new dynamic properties sheet by selecting it from the AppBuilder's Window menu. This shows all properties for the selected object, and lets you set properties for it or select other objects in the same container. [Figure 3-5](#) shows some of the properties for the CustNum field in the customerviewv Viewer.

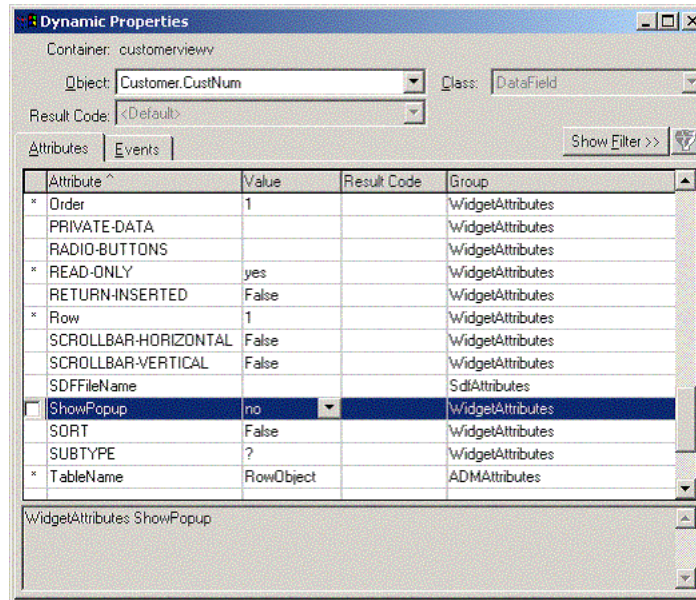



Figure 3-5: Dynamic properties sheet

This property sheet is described in detail in the *Progress Dynamics Developer's Guide*, but it is useful to point out a few things in this example:

- If you look at the ShowPopup attribute, you see that its value is No. The row is not marked by an asterisk (*) or checkmark in the first column, which means that the value was not modified **for this instance**. This is a value inherited from either the Class for the Object type, or (as in this case) from the Master Object. Because you changed the value for the CustNum field in the Repository Maintenance tool, that modified master value is inherited by this instance.

- If you look at other attributes, you can see that some of them **have** been overridden or set specifically for this instance, including the Order (indicating that the CustNum field is the first field in the tab order for the Viewer), the Row it is displayed in, and the RowObject TableName used because the Viewer is derived from an SDO. These are all attribute values that were set automatically when the instance of the Viewer was created in the window, but they are still different from the Master values, and so they show as overrides. In addition, the Read-Only attribute is Yes, and this is also shown as an override. This is because we set the attribute value in the standard AppBuilder property sheet for the field.
- If you double-click on the Viewer itself, outside any field, or press the Object Properties button  in the AppBuilder Toolbar, then the Viewer's property sheet shown in [Figure 3-6](#) appears.

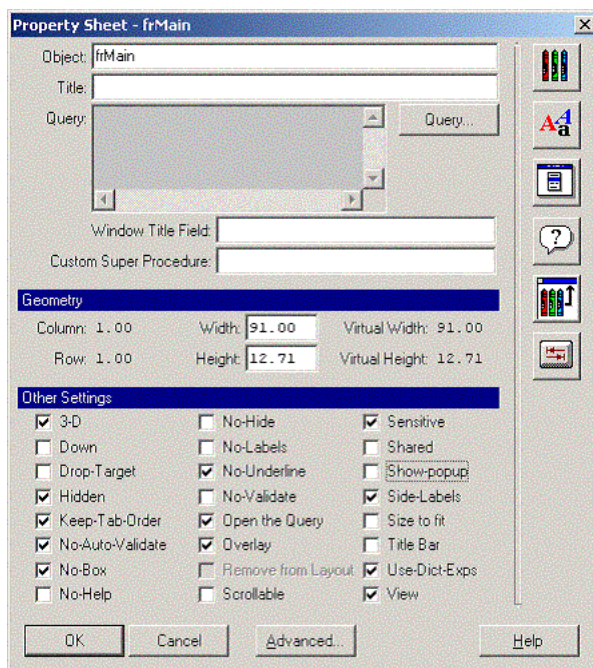


Figure 3-6: Viewer property sheet

Here you can set properties that apply to the Viewer itself. Because you are doing this in the Viewer's design window, you are setting these attributes at the Master level (there's no higher level for this customerviewv Viewer). Here, for instance, you can turn the Show-Popup attribute off **for the Viewer**, because this attribute is supported both for individual Fill-Ins and for Viewers. If you set it to False for the Viewer, this removes all popup buttons for all fields in the Viewer.

- You can also set this property in the dynamic property sheet in the same way you did for the CustNum field. By contrast, you can set the attribute to False for the Viewer just when used in the oemaintwin window by bringing up the dynamic property sheet in the Container Builder for oemaintwin. When you do this you are editing a specific container that uses the Viewer, so any attributes you set for the Viewer are set only for this instance of its use.

So to summarize, the level at which you set attributes depends not on the tool you are using, but on the context of where you are using it.

If you create or edit a Viewer in the AppBuilder, and set one or more Viewer attributes, either in the Viewer property sheet provided by the AppBuilder or the dynamic property sheet you can run from it, you are setting attributes at the master level, for the object you are creating.

If you define attributes for a DataField object in the Repository Maintenance tool, you are also doing it at the master level, for all uses of the field.

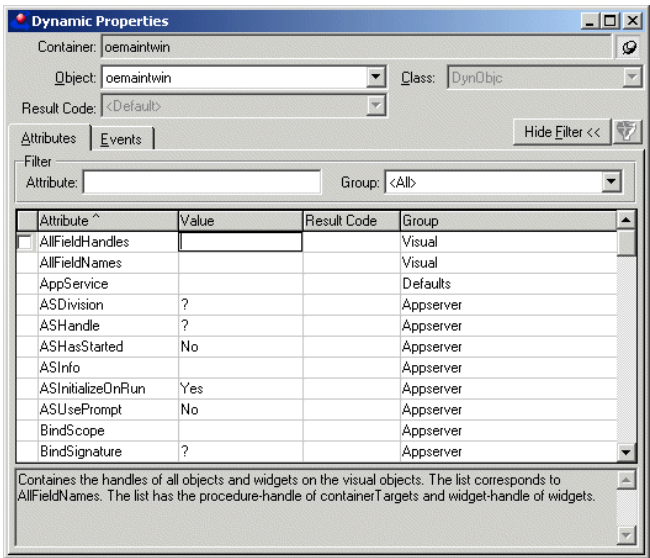
On the other hand, if you bring up the custom property sheet in the AppBuilder for a field or widget of a specific type, such as the one you saw for the CustNum field, you are defining attributes of that field **as used in that particular Viewer**. So you are defining attribute values at the Instance level. Any such settings will be seen only in the context of the Viewer you are editing.

Likewise, if you use the dynamic property sheet to define attribute values for fields in a Viewer, the same thing is true: you are defining Instance attribute values.

To move the discussion up a level in the Object hierarchy, if you are creating a container window in the Container Builder, you can click the Container Properties button to set property values for the window itself:



These are values at the Master level, because you are defining property values for the window you are currently creating:



On the other hand, if you bring up the property sheet for one of the Viewers or other Objects you place into the container, then you are defining values at the Instance level for the object.

3.2.4 Attribute value level summary

It is important to keep these distinctions in mind so that you do not define behavior changes whose scope is larger or smaller than you want. As with any programming exercise, it is always best to do the job once and only once. So if you want to remove the Calculator popup from the CustNum field everywhere it might ever appear, you should do this at the Master level, using the tool that gives you access to the attributes at that level (the Repository Maintenance tool in this case). That way you never have to change the property when you use the field. It also means that you can change your mind about this new default value and only have to change data in one place. Because of the Progress Dynamics inheritance model for attribute values, the change in value at the **Master** level will be inherited by any Instance (that is, any use) of the object where the attribute has not specifically been set to some value.

On the other hand, when you want a change in default behavior to be seen only in the current context, then set the property value at the **Instance** level using the available tools.

To review, for **Master objects**:

- You can set Master attribute values for DataField Objects (individual database fields as used in the framework) in the Repository Maintenance Tool. Because the Repository Tool shows you the actual records in the repository database, it will not show **any** value for an attribute whose value is inherited from its class, so, as you saw for the ShowPopup example, you might need to add a record to define the attribute value.
- You can set Master attribute values for SDOs, Viewers, and Browsers in the AppBuilder. You can either use the AppBuilder's own custom property sheets for these objects (by double-clicking on the object in its design window, for example), or you can bring up the dynamic property sheet from the Windows menu.
- You can set Master attribute values for windows in the Container Builder by clicking on the Container properties button.

And for **Instances**:

- You can set Instance attribute values for DataFields in the Viewer where you use the field, when you edit the Viewer in the AppBuilder, using either the AppBuilder's custom property sheet for the field type or the dynamic property sheet.
- You can set Instance attribute values for SmartObjects such as SDOs, Browsers, Viewers, Toolbars, and Folders, by editing the window where the object instance occurs in the Container Builder and bringing up the dynamic property sheet for the object.

3.2.5 The attribute control and the object type control

Although this chapter does not go into a detailed discussion of how attributes are created, there are a few basic pieces of information concerning attributes that are important to mention here.

First of all, every attribute used in Progress Dynamics is defined in the repository database. Attributes can be added and edited using the Attribute Control on the Progress Dynamics Development window's Attribute menu. [Figure 3-7](#) shows the definition of the ShowPopup attribute.

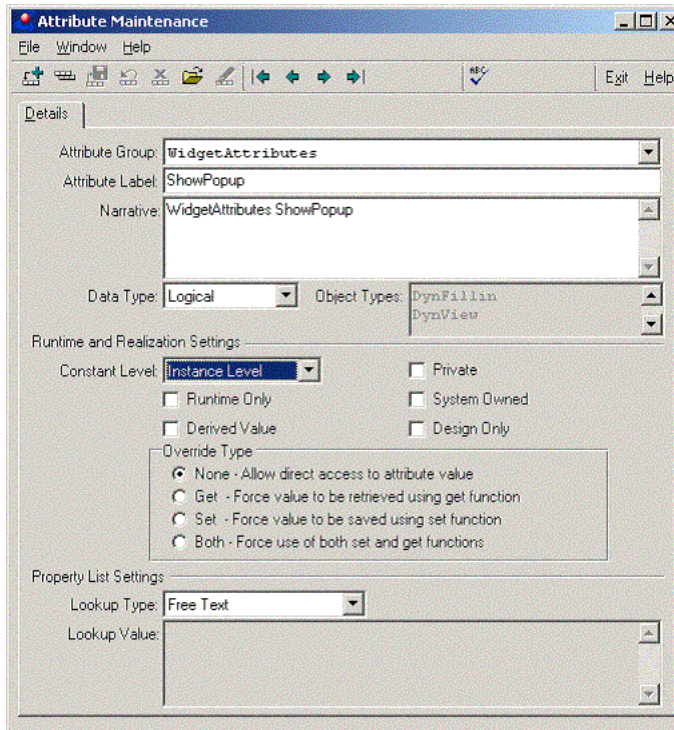


Figure 3-7: Attribute Maintenance window

There are a few key things to note about this definition that will help you understand the ways in which you can modify attribute values:

The **Constant Level** defines the lowest level of the attribute hierarchy at which a value can be defined for the attribute. This can be:

- **Class** — The attribute is given a value for a class, and that value can never be changed for any object in the class. An example of this is the SupportedLinks attribute, which is defined once for each SmartObject class. If you try to edit an attribute whose Constant Level is Class, using the dynamic property sheet, you see that the initial value of the attribute is displayed, but you cannot change it.

- **Master** — The attribute is given a value for each Master object created in the class, and that value cannot be changed in an Instance. If you try to edit an Instance attribute whose Constant Level is Master, using the dynamic property sheet, you see that the initial value of the attribute is displayed, but you cannot change it. An example of this is the `ObjectName` attribute.
- **Instance** — The value can be changed for any Instance of the Object.

There are five **toggles** on the window that also help identify where and how you can change the attribute value:

- **Private** — If this toggle is checked on, then the attribute is intended for internal use only. Private attributes are not displayed in the dynamic property sheet and should never be modified by application code. Note that the functions that set and get the attribute value may not themselves actually be declared as `PRIVATE`, for programming reasons, but you should not use them.
- **Runtime Only** — If this toggle is checked on, then the attribute's value is only set to a meaningful value at runtime. It makes no sense to set the value at design time in one of the application design tools. Examples of this include attributes whose values are handles, such as `DataSource`, and attributes that record the current state of an object at runtime, such as `ObjectInitialized`. These attributes also do not appear in the property sheet.
- **Design Only** — If this toggle is checked on, then the attribute's value is only meaningful at design time. These attributes are also not displayed in the property sheet, because their values are not used at runtime.
- **Derived Value** — If this toggle is checked on, then the attribute value is derived from a calculation that is done by the **get** and **set** functions for the attribute, and is not actually stored with other attribute values. An example of this is the `UpdatableColumns` property of an SDO, which is derived from another property called `UpdatableColumnsByTable`, which itself is used only internally. Because such values are not stored within the `SmartObjects`, or in the repository, they are also not included in the dynamic property sheet.
- **System Owned** — If this toggle is checked on, then the attribute is essential to the workings of the framework and the basic behavior of applications. These attributes and their initial values should not be changed. Developers need special privileges to change these. Many of these attributes can, however, be set and used at runtime.

Thus you see that many attributes used internally are not displayed in the property sheet, for a variety of reasons.

The **Lookup Type** drop-down list lets you define the way in which the attribute is edited in the Dynamic property sheet and includes the following types:

- **Free Text** — Fill in the attribute value in the property sheet.
- **List Item Pairs** — You select a value from a list of possible values. The items are in pairs because the value stored in the repository may be a coded value that is not displayed to you.
- **Dialog** — You bring up a custom dialog box to edit the property value. You can also type directly the attribute value. The dialog is specified in the lookup field value.
- **Dialog (Read Only)** — You bring up a dialog box to edit the property, and you cannot choose any value not presented to you in the dialog. The Color properties, for example, use a dialog box to choose from a color editor.
- **Procedure** — The property sheet runs a user-defined procedure to edit the attribute value.

There are other limits in defining and using attributes. One important characteristic of an attribute that is not defined in the Attribute Maintenance window is its initial value. This is because attributes are not given values until they are associated with a class. You do this in the Object Type Control window under the Dynamic Development window's Object menu. This treeview-based window shows all the Object classes in a hierarchy starting with Base, which represents the **smart** class common to all SmartObjects. Attributes are defined for each class, so an individual object inherits attributes and their initial values from all the classes in its branch of the hierarchy. [Figure 3–8](#) shows an example of drilling down through the DynView class to locate the FrameMinHeightChars attribute.

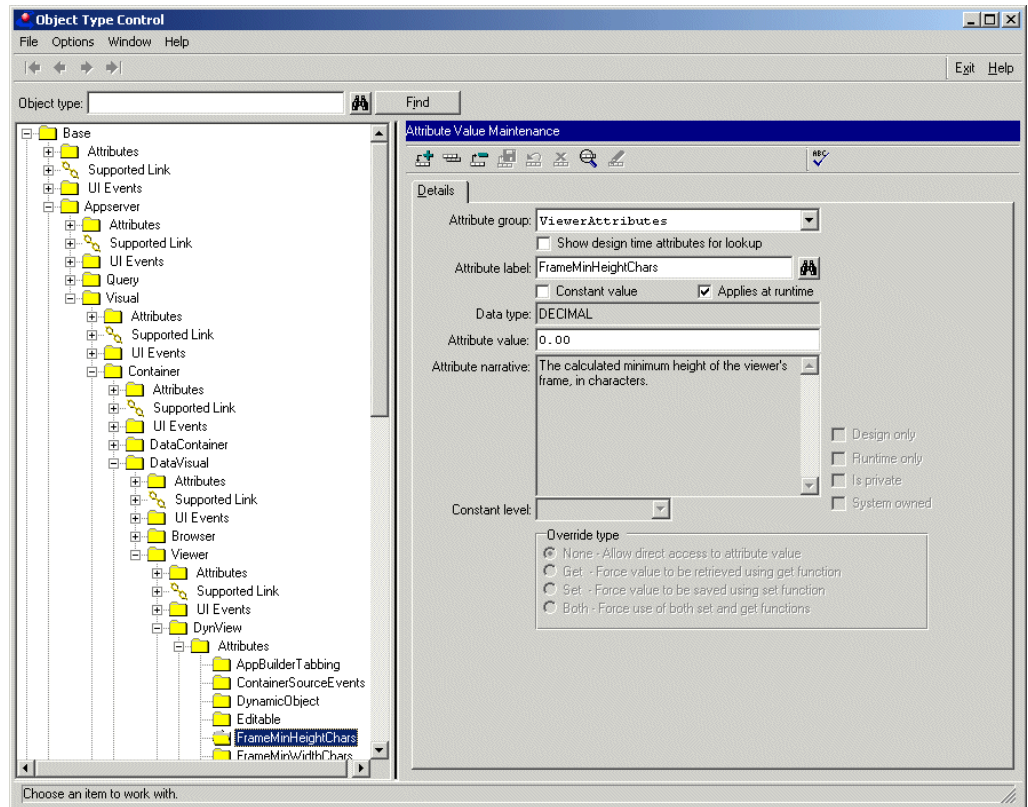


Figure 3–8: Object Type Control window

The organization of the hierarchy is not entirely accurate for all objects. Some classes such as AppServer are optional parts of other classes, but the treeview does not represent this option. This is why, to locate the DynView class, you expand the Base class, then AppServer, then Visual, then Container (again optional, as a Viewer can be a container if it has SmartDataFields such as dynamic Lookups and Combos in it), then DataVisual, then Viewer. This is where you define the default value if any for an attribute as used in that class.

It is important to understand that every attribute you use in an Object **must** be defined for the Object Type at this class level. You cannot simply add an attribute to a single Object or a single Instance of an Object. So when, for example, we described adding an attribute value record for a DataField in the Repository Maintenance tool, this was because attribute values are stored at that level only if they override the default values for the class. So when you add an attribute value record, it must be for an attribute that is defined for one of the classes the Object inherits from.

Obviously all the attributes that are needed to provide all the standard behavior for all the Objects in the framework are predefined in the repository, and in principle you are likely never to need to use these tools. However, if you define your own Objects or create a subclass of an existing Object, you can add new attributes that your application needs the Objects to have.

3.3 Disabling and hiding buttons and menu items

The Progress Dynamics Toolbars are made up of buttons and menu items whose state changes as the application runs. You see this whenever you navigate through records or make changes to a record and press the Save button. Later we discuss how to define the specific logic for when items are enabled and disabled, hidden and viewed, or the button image is changed. This section discusses how to define the list of items that should be removed from a Toolbar at runtime, either by hiding them or disabling them.

Obviously, if your application needs a Toolbar that does not match any of the standard Toolbars already defined in the framework, you should define a Toolbar that has exactly the items you need on it. But sometimes you may want to disable or hide items programmatically under certain circumstances, so that they are sometimes available and sometimes not. There are two Toolbar properties and several functions that assist you in doing this. The properties are **DisabledActions** and **HiddenActions**.

3.3.1 Disabling actions based on a data value

To construct an example, suppose that you want users of the oemaintwin window you built to be able to edit or delete only Customers and Orders for Customers in the USA. To do this, you must disable the buttons that allow those operations, depending on the value of the currently selected Customer, and also enable or disable the fields in the Viewers on the different folder pages accordingly.

Using the TableIOType property

The first thing you need to do is to change the basic display mode of the Toolbar to leave Viewers disabled until someone specifically wants to do an update. That way, the Viewers are initially disabled, and the user has to have access to the Modify button in order to make changes. In earlier versions of the ADM and its SmartPanels, this was done with the PanelType property, which you could set to one of two states:

- **Save** — Puts the buttons and their TableIO-Targets into a state where everything is enabled and the user just has to make changes and press the Save button to record them.
- **Update** — Disables TableIO-Targets and places the buttons into a state where the user must press the Update button to enable the Viewers and begin to make changes, and then press Save when done.

The Toolbar equivalent of this property is TableIOType, and it has the same two possible values: Save and Update. The default setting is Save, so things start out enabled. You must change the setting to Update for this example to work. Change the value in the Container Builder, by bringing up the Toolbar's dynamic property sheet shown in [Figure 3-9](#), and changing the property value there.

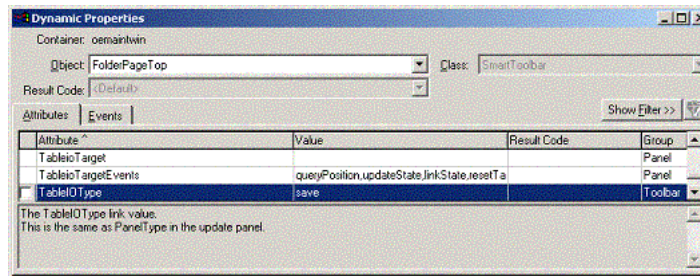


Figure 3-9: Dynamic Properties window

There is another way to make runtime changes to the object in a custom super procedure that executes custom code when the window is initialized. You can define a custom super procedure for the Toolbar itself, but because this must be done as part of the overall window initialization, this example shows the code in a custom super procedure for the container window. Where you put your code is a matter of overall organization and preference; there is always more than one solution to a programming problem.

Defining a custom super procedure for the window

Define a New Structured Procedure in the AppBuilder, and create an `initializeObject` internal procedure for it. Because this code is executed on behalf of the window, it first needs to get the handle of the Toolbar object. To do this, add a `ContainerToolbar` link from the Toolbar to the window (THIS-OBJECT), which may be useful for other purposes as well, as is described in a later section of this chapter. Alternatively, the code could simply check among the window's `Container-Targets` for the handle of an object with the `ObjectName` of `FolderPageTop`. Again, there's almost always more than one way to solve a problem.

Given the Toolbar handle, you then need to set the `TableIOType` property to `Update`. This code sample uses the `{get}` and `{set}` include files to do the job. You could also use the `DYNAMIC-FUNCTION` syntax and the equivalent `get` and `set` functions for the same effect. This is also mostly a matter of personal preference and coding style. This all needs to happen before the `RUN SUPER` statement, so that the property is set before all the Objects are initialized:

Procedure `initializeObject`:

```
/*-----  
Purpose:      Override of initializeObject to reset the TableIOType  
               property value to 'update' so that Viewers are initially  
               disabled.  
Parameters:   <none>  
-----*/  
DEFINE VARIABLE hToolbar AS HANDLE      NO-UNDO.  
  
  {get ContainerToolbarSource hToolbar}.  
  {set TableIoType 'update' hToolbar}.  
  RUN SUPER.  
END PROCEDURE.
```

Registering the custom super procedure

As always, remember to register your super procedure in the repository, by selecting that option in the AppBuilder's File menu. Save it as a Procedure in the appropriate product module. We have called the procedure `oemaintwinsuper.p`.

Once you have done this, attach it as the custom super procedure for the container window. You can do this in the Container Builder shown in [Figure 3–10](#).

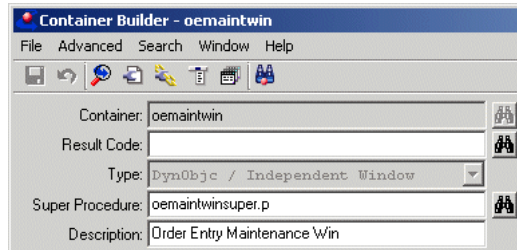


Figure 3–10: Container Builder for oemaintwin

Note that you can set the custom super procedure for objects that are edited in the AppBuilder, such as Browsers and Viewers, in the AppBuilder’s property sheets for those Objects.

As we have mentioned elsewhere, you do not specify the relative pathname where the procedure is actually stored, because you are just providing its LogicalObjectName as stored in the repository. This name includes the .p filename extension, but not the pathname. The pathname is attached automatically at runtime by looking up the product module record in the repository.

Defining a custom super procedure for a viewer

Now that you have set the TableIOType property, you need to create code to selectively enable and disable both the Toolbar buttons and the Viewer fields depending on the value of the Customer Country field. Put this code into a custom super procedure for the custcommentstv Viewer that we have named customersuper.p. See [“Defining a custom super procedure for the window”](#) for more information on how to create this procedure.

Using the modifyDisabledActions procedure In the toolbar

To control the Toolbar buttons and associated menu items, change the DisabledActions or HiddenActions attributes. In this example, you should disable the buttons and menu items rather than hiding them so that they do not appear and disappear as the user changes records, which would not be an appropriate user interface. In other situations you can remove the button or menu item from the Toolbar altogether by adding it to the HiddenActions property.

There is a special support procedure to help you add and remove entries from the DisabledActions property, whose value is a comma-separated list of actions. Note that at present there is no equivalent procedure for HiddenActions; you will have to parse the list yourself, or create your own modifyHiddenActions procedure based on modifyDisabledActions.

First you need to identify the names of the actions that you are going to disable. To identify these names, open the Toolbar in the Toolbar and Menu Designer. Expand the SmartToolbars node and locate the Toolbar you are using, which is FolderPageTop in this case. Expand that node to see the bands it uses, and then expand the TableIoMod band, which holds all the update-related actions. Here you can see the description of each action. Following the description is the actual action name in parentheses, which also appears as the Item Reference in the maintenance frame to the right, as shown in [Figure 3–11](#). This is the name to use in setting DisabledActions.

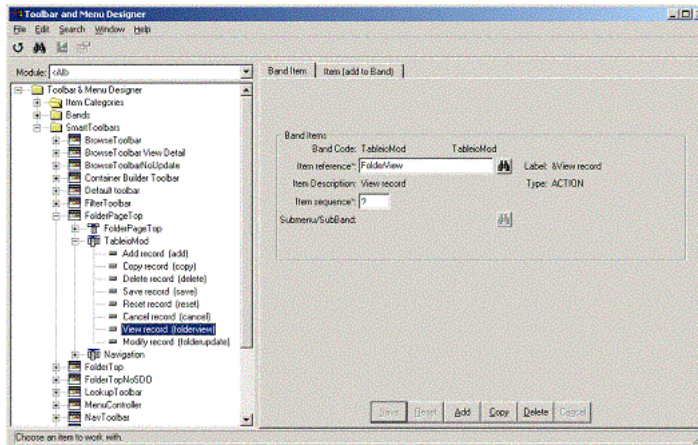


Figure 3–11: Toolbar and Menu designer

To disable the Delete button in the FolderPageTop Toolbar, and also the two buttons that allow you to toggle the state of the displayed record from View mode to Update mode, you need to add **Delete**, **FolderView**, and **FolderUpdate** to the list.

Since you need to do this each time a Customer is selected, the appropriate place for the code is in a local version of the dataAvailable event procedure. This is published by the SDO each time the record position changes, and the Viewer subscribes to the event so that it can display the new values. Unlike most event procedures, which have no parameters, this one has a single INPUT parameter that you need to define and pass on to the super procedure. See [Chapter 5, “Using ADM2 Properties and Methods In Progress Dynamics,”](#) for more information on this and many other useful properties and entry points in the code that you may want to use.

Important note about parameters on overrides and publish

Remember to double-check for parameters in any procedure that you are going to override and in any event that you publish. If you get the parameter list wrong for a local version of a procedure that you RUN, you will get an error at runtime to remind you of your mistake. But if you get the calling sequence wrong for a PUBLISH statement, the event simply will not occur, because there will not be any matching subscriber with the same parameter list. Unfortunately, when you create a custom super procedure and add code to it in the AppBuilder, the Section Editor is not aware of what Object it will be associated with, so it cannot supply you with a list of all the valid procedures and functions you can override, along with their parameters, as it does when you add procedures to a static SmartObject. Therefore, make sure you get this right. Also, do not forget to include the RUN SUPER statement in the right place in your custom code. If you leave it out, the standard code for the event will not execute at all, and the results will likely be very strange.

Getting values from different linked objects

Your custom code must first get the handle of the Toolbar. Because there is no link from the Toolbar to the Comments Viewer (the Viewer is not updatable, so it has no TableIO link), you need to get the handle of the Customer SDO, which is the Viewer's Data-Source, and then get the SDO's Navigation-Source, which is the Toolbar.

You also need to retrieve the value of the Country field for the selected record. You can use the columnValue function in the SDO to do this.

And finally, the code must get a list of all the TableIO-Targets of the Toolbar (which are the three updatable Viewers on pages 2, 3, and 4). This is the TableIOTarget property. Though the name is singular (**Target** not **Targets**), the data type is CHARACTER, because there may be a list of handles of multiple Targets, stored as a comma-separated list of handles in string form:

Procedure dataAvailable:

```
/*-----
Purpose:  Override of dataAvailable to reset the DisabledActions of
          the Toolbar and enabled or disable fields based on the Country.
Parameters: pcRelative AS CHARACTER
Notes:
-----*/
DEFINE INPUT  PARAMETER pcRelative AS CHARACTER  NO-UNDO.

DEFINE VARIABLE cCountry      AS CHARACTER  NO-UNDO.
DEFINE VARIABLE hDataSource   AS HANDLE     NO-UNDO.
DEFINE VARIABLE hToolbar      AS HANDLE     NO-UNDO.
DEFINE VARIABLE cTargets      AS CHARACTER  NO-UNDO.
DEFINE VARIABLE hTarget       AS HANDLE     NO-UNDO.
DEFINE VARIABLE iTarget       AS INTEGER    NO-UNDO.

ASSIGN hDataSource = DYNAMIC-FUNCTION ('getDataSource' IN TARGET-PROCEDURE)
      hToolbar     = DYNAMIC-FUNCTION ('getNavigationSource' IN hDataSource)
      cCountry     = DYNAMIC-FUNCTION ('columnValue' IN hDataSource, 'Country')
      cTargets     = DYNAMIC-FUNCTION ('getTableIOTarget' IN hToolbar).
```

Next the code checks the value of the Country field and runs `modifyDisabledActions` to either add or remove the actions. Remember that the goal is to allow users to edit and delete only records related to Customers in the USA. The `modifyDisabledActions` procedure takes two arguments:

- **Add or Remove** — To indicate whether to add actions to or remove actions from the list of disabled actions
- **Action List** — A comma-separated list of action names.

If the Country is USA, the user is allowed to do updates and deletes, so the actions are removed from the disabled list:

```
IF cCountry EQ "USA" THEN
DO:
    DYNAMIC-FUNCTION ('modifyDisabledActions' IN hToolbar,
                      'Remove', 'FolderUpdate,FolderView,Delete').
```

Using enableFields and disableFields

Then the code goes through the list of TableIO-Targets, converts each back to a handle, and runs the enableFields procedure in it. In order to avoid doing this unnecessarily, it first checks the value of the Viewer's FieldsEnabled property to see whether the fields are already in the right state:

```
DO iTarget = 1 TO NUM-ENTRIES(cTargets):
    hTarget = WIDGET-HANDLE(ENTRY(iTarget, cTargets)).
    IF NOT DYNAMIC-FUNCTION('getFieldsEnabled' IN hTarget) THEN
        RUN enableFields IN hTarget.
    END.
END.
```

Correspondingly, if the Country is not equal to USA, then the actions are added to the disabled list, and the code must run the disableFields procedure in the Viewer. This is another case where you have to be careful to get the calling sequence right. Unlike enableFields, the disableFields procedure takes an INPUT parameter, which can be All or Create to indicate whether all fields are to be disabled, or just those associated with creating a new record:

```
ELSE DO:
    DYNAMIC-FUNCTION ('modifyDisabledActions' IN hToolbar,
        'Add', 'FolderUpdate,FolderView,Delete').
    DO iTarget = 1 TO NUM-ENTRIES(cTargets):
        hTarget = WIDGET-HANDLE(ENTRY(iTarget, cTargets)).
        IF DYNAMIC-FUNCTION('getFieldsEnabled' IN hTarget) THEN
            RUN disableFields IN hTarget ('All').
        END.
    END.
```

This is the end of the code for the Viewer’s localization of dataAvailable. Be sure to register the super procedure in the repository, and associate it with the Viewer. You can set the custom super procedure for the Viewer in the Repository Maintenance tool, or more conveniently, in the AppBuilder, by opening the Viewer and going into its property sheet, as shown in [Figure 3–12](#).

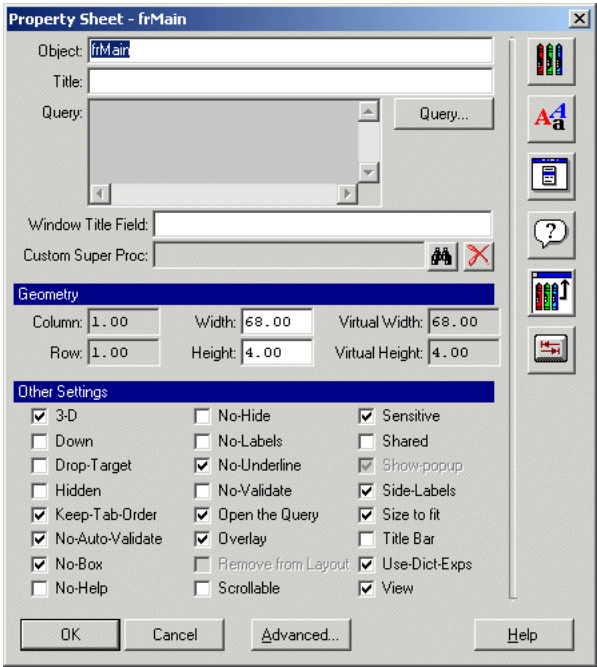


Figure 3–12: Property sheet for frMain

Testing the window with DisabledActions

Now when you run the application window, and select a Customer not in the USA, all the Viewers are disabled, and the Delete, View, and Modify actions in the Toolbar are also disabled, as shown in [Figure 3-13](#).

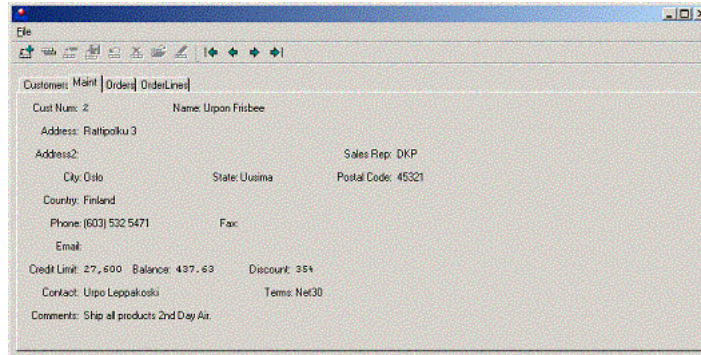


Figure 3-13: Test of window with DisabledActions

Using the toolbar's reset procedures

The toolbar uses a set of event procedures to reset the action state for various types of actions. These procedures are executed at times when it seems correct to check for a state change, and as a result you do not normally need to run them yourself. For instance, they are run on a page change, which is why the action state is corrected when you switch pages.

The event procedures that the Toolbar supports are `resetTableIo`, `resetCommit`, and `resetNavigation`. These are published by objects like the Viewer, and generally you do not need to invoke them explicitly. If you find that the Toolbar state is not being reset properly at some particular place in your application such as this Viewer, you can simply publish the event from the Viewer, and the Toolbar will receive it. Remember to publish it FROM TARGET-PROCEDURE, so that the event is associated with the Viewer itself and not its super procedure. This is what the first block of code looks like with an example of publishing `resetTableIo` to make sure that the update buttons are reset properly:

```
IF cCountry EQ "USA" THEN
DO:
    DYNAMIC-FUNCTION ('modifyDisabledActions' IN hToolbar,
                      'Remove', 'FolderUpdate,FolderView,Delete').
    PUBLISH 'resetTableIo' FROM TARGET-PROCEDURE.
    DO iTarget = 1 TO NUM-ENTRIES(cTargets):
        hTarget = WIDGET-HANDLE(ENTRY(iTarget, cTargets)).
        IF NOT DYNAMIC-FUNCTION('getFieldsEnabled' IN hTarget) THEN
            RUN enableFields IN hTarget.
    END.
END.
```

You can also publish the general-purpose reset procedure `resetTargetActions` to reset actions related to other link types, including custom links of your own. The procedure takes the name of the link as an `INPUT` parameter.

3.3.2 Putting the value check into the window code

Now that the example is complete, you can look at an alternative way of doing the same thing. Consider the procedures that you have just created. Most of the work is done in a custom super procedure attached to the read-only Customer Viewer on Page 1 of this window. It is important to keep in mind that you can define a custom super procedure for an Object only at the Master level, for the Object itself, and not for a single Instance of the Object in a particular window. In this case, it may not be appropriate to have the `dataAvailable` code associated with that Viewer, because the special behavior it defines is really specific to this one window. In fact, the code would generate numerous errors if the Viewer were in a different window, because the code as it stands assumes various things about what Objects are in the window and what links there are between them. This in itself is worth correcting by improving the code to deal gracefully with its context if the window design is changed in some way.

If you do not want to associate the behavior specifically with the `custcommentsv` Viewer, you can instead tie it to the window by putting the code into the `oemaintwinsuper.p` procedure. Let's look at that alternative and see what changes you have to make to the code in its new location.

First look at the `initializeObject` procedure in `oemaintwinsuper.p`. The reason it seemed more natural to add the new behavior to the Customer Viewer is that it required localizing the `dataAvailable` event, which the Viewer subscribes to in the SDO. The container window does not subscribe to this event, and therefore does not normally receive it. This is the first thing you have to change in order to move the code to the window.

You can intercept this event in the window simply by adding a statement to subscribe to it. So this new version of the `initializeObject` code has a few additional lines in it. The first line gets the handle of the Customer SDO by retrieving the `Navigation-Target` of the Toolbar. The `NavigationTarget` property is of datatype `CHARACTER`, so you need to retrieve it into a `CHARACTER` variable and then convert that to a handle.

NOTE: The `DYNAMIC-FUNCTION` operation that takes place inside the `{get}` is forgiving enough that it actually casts the `CHARACTER` output from the `getNavigationTarget` function directly into the `HANDLE` variable `hSDO`. You would be well advised **not** to take advantage of the runtime engine's generosity in this regard. In particular, if the `{get}` is turned into a direct lookup in the properties temp-table instead of a function call, you will get an error from your attempt to combine the two steps.

Keep in mind that the reason the NavigationTarget property is of type CHARACTER is that there could be more than one Navigation Target for the Toolbar, in which case the multiple targets would be represented as a list. In this case there is only one, because at the time initializeObject gets run, only Page 0 and Page 1 of the folder have been created, so there is only one SDO in the window. Try not to make this kind of assumption in your own finished application code, so as to avoid errors if the window you are working with changes later on.

The code you need to add to what was already in this initializeObject procedure is highlighted in bold:

```

Procedure initializeObject:
/*-----
  Purpose:      Override of initializeObject to reset the TableIOType
                  property value to 'update' so that Viewers are initially
                  disabled.
  Parameters:   <none>
-----*/
DEFINE VARIABLE hToolbar AS HANDLE      NO-UNDO.
DEFINE VARIABLE cSDO      AS CHARACTER NO-UNDO.
DEFINE VARIABLE hSDO      AS HANDLE NO-UNDO.
  {get ContainerToolbarSource hToolbar}.
  {set TableIoType 'update' hToolbar}.
  {get NavigationTarget cSDO hToolbar}.
  hSDO = WIDGET-HANDLE(cSDO).
  SUBSCRIBE PROCEDURE TARGET-PROCEDURE TO 'dataAvailable' IN hSDO.
  RUN SUPER.
END PROCEDURE.

```

Once you have the handle of the SDO you can subscribe to dataAvailable. Again, remember to subscribe the TARGET-PROCEDURE, not the super procedure itself. Now the window will get dataAvailable events just as the Viewer does.

The next step is to move the dataAvailable code itself from the Viewer's super procedure to the window's. You can then delete the Viewer's super procedure and remove its association from the Viewer in the Repository Maintenance tool.

Because the dataAvailable code is now executed from the window's perspective and not the Viewer's, there are a few changes you need to make. Let's take a look at those.

Identifying an object among all the contained objects

First, just as with the code in initializeObject, this version of dataAvailable must locate the Customer SDO so that it can get the value of the Country field. And in this case, the warning about allowing for multiple Navigation-Targets becomes a reality. By the time this is executed, the other pages in the folder have likely been enabled, and the Toolbar now has multiple Navigation-Targets. So you have no choice but to retrieve that value as a list and search through it for the Customer SDO.

NOTE: If you neglect to do this, you will get the error message, `Field Value too large for Integer`, because the run-time engine tries to convert some string such as `12345,24356,45675` into a handle, which is treated internally as a kind of integer.

You could do this by checking the `LogicalObjectName` property and comparing that to `custcommentsv`. This seems a little too restrictive, though, since it means that you have to edit the code if you ever change the name of the Viewer on Page 1. Since the code is checking the value of a field in the Customer table, it seems safer to check for the presence of the Customer table in the SDO, which is in its `Tables` property. The code allows for the possibility that the Customer table might be joined to some other table in the SDO, in which case the `Tables` property would be a list:

```
ASSIGN hToolbar = DYNAMIC-FUNCTION ('getContainerToolbarSource' IN
TARGET-PROCEDURE)
      cTargets = DYNAMIC-FUNCTION ('getNavigationTarget' IN hToolbar).

DO iTarget = 1 TO NUM-ENTRIES(cTargets):
  hTarget = WIDGET-HANDLE(ENTRY(iTarget, cTargets)).
  IF LOOKUP('Customer',
            DYNAMIC-FUNCTION('getTables' IN hTarget)) NE 0 THEN
    DO:
      hDataSource = hTarget.
      LEAVE.
    END.
  END.
END.
```

Scoping variables in super procedures

Here it is worth making another digression to discuss an important point. You may have realized that once `initializeObject` has determined the SDO handle, it could just stash it in a variable defined in the Definitions section of the super procedure, and then this code would not have to locate it again. This is true, but it is also very dangerous. In the context of the example, this would work fine, because the super procedure is serving only the `oemaintwin` window, so the value of the SDO handle would be valid for the life of the window. But what if there were some circumstance under which the user could run two copies of the window at the same time? This might easily be allowed in the application, and they would both share a single running instance of the super procedure. The value of the SDO handle saved away by one running instance of the window would not be valid for the other. The kinds of errors that this results in can be very insidious and difficult to track down. Or look at another possibility. What if you realize later that another window requires similar support for disabling actions based on a value check? If you are doing your job right, you will not create a new procedure to handle that and just copy code from the first one. You will start with the existing procedure, take the elements that are subject to change (such as the name of the field and its value) and turn them into parameters or properties of the Object, and let the single procedure work for all windows requiring this type of support.

Now it is very likely indeed that two different windows of this kind might be running at the same time, in which case they would be sharing a single instance of the super procedure, since much of the purpose of super procedures is to reduce memory use as well as r-code. The moral is that it is very bad form to hold any values across calls to different methods in a super procedure unless you are positive that a single instance of the super procedure will **never** be used to support two different Objects at the same time.

Redirecting the PUBLISH statement from the window

The next step remains the same as before: Get the Country value, and reset the cTargets variable to hold a list of the TableIO-Targets:

```
ASSIGN cCountry    = DYNAMIC-FUNCTION ('columnValue' IN hDataSource, 'Country')
      cTargets      = DYNAMIC-FUNCTION ('getTableIOTarget' IN hToolbar).
```

Look at the next statement:

```
PUBLISH 'resetTableIO' FROM TARGET-PROCEDURE.
```

When this code was in the Viewer's super procedure, the TARGET-PROCEDURE was the Viewer itself. But now the code is executing from the window. Rather than going to the trouble of tracking down the handle of the custcommentsv Viewer, it is simpler just to RUN resetTableIO directly in the Toolbar, whose handle you do have, since this has exactly the same effect. Remember that a PUBLISH statement is basically the same as a RUN statement except that the procedure handle the event gets run in is not specified in the PUBLISH statement. So you should change the statement to this:

```
RUN resetTableIO IN hToolbar.
```

Reasons not to RUN SUPER from the window

The final change is also important to think about for a moment. Ordinarily, you must remember to include the RUN SUPER statement in your local procedure. Here you must remember to leave it out! The reason is that the window is not a native subscriber to the dataAvailable event at all. It just needs to intercept it to take some action on behalf of other Objects. For this reason, no super procedure of the window implements dataAvailable, and if you try to invoke it with a RUN SUPER statement, you will get an error at runtime, much as if you had put in a statement that said "RUN dataAvailable IN <non-existent-handle>".

So you must remove the RUN SUPER statement as you convert the code to work in the window's super procedure:

```
/* RUN SUPER (pcRelative). */  
END PROCEDURE.
```

In some cases, especially where code is designed to work in a variety of situations, you might be faced with a case where there might or might not be a super procedure with the event in it. In this case it is perfectly OK to write "RUN SUPER NO-ERROR".

Reasons not to use enableActions and disableActions

You may notice that there are two functions defined for Toolbars in the Panel class called enableActions and disableActions. You might be tempted to run those functions directly rather than setting the DisabledActions property the example in this section described. This is generally not a good idea, as these functions are intended for internal use only.

In particular, they are very short-term in their effect. That is, if you use disableActions to disable, say, the Delete action, it will have an immediate effect, but as soon as any operation occurs that resets the Toolbar, such as a page change, the action will be enabled again. This is probably not what you want, and this is why it is better to use the DisabledActions **property** to change the settings until you need to change them again.

3.4 Defining action rules for menu and toolbar items

When you define items for a Progress Dynamics menu or toolbar, you can define a set of action rules that determine when the item (that is, the toolbar button or menu item) is enabled, when it is hidden, and when an alternate image for a toolbar button should be displayed. In fact, this capability is now a standard part of the ADM2 code, and determines the behavior of all of the standard Navigation, Commit and TableIO items in SmartPanels and Toolbars. The code to support this replaces the setButtons SmartPanel procedure that has been used in earlier versions of the ADM. Application code can still call setButtons, but it is no longer used as a routine part of the action of enabling and disabling buttons and menu items.

An action rule can be an expression containing a combination of SmartObject property references and function references. The expression controls different actions as follows:

- If the expression for the **enable rule** returns True, then the item is enabled; otherwise it is disabled.
- If the expression for the **hide rule** returns True, then the item is hidden, that is, actually removed from the menu or the toolbar; otherwise it is viewed.
- If the expression for the **alternate image rule** returns True, then the alternate image or Image 2 is displayed for a toolbar button; otherwise the primary image (Image 1) is displayed.

These rules are evaluated whenever the state of the Toolbar's window changes in a way that could require a re-evaluation of the rules, for example, on a page change, when a record is selected or navigated to, or when one of the toolbar buttons or menu items is selected.

You can also programmatically force a reevaluation of a rule. You get a handle of the toolbar and run the resetTargetActions procedure. You can add the following in your super of your container:.

```
{getContainerToolbarSource ctHandles}  
  
DO i = 1 to NUM-ENTRIES(ctHandles):  
    hHandle = WIDGET-HANDLE(ENTRY,ctHandles)  
    RUN resetTargetActions('myLink') IN hHandle.  
END.
```

This code will re-evaluate all rules for items having an item link of *mylink*Target.

You define action rules in the Toolbar and Menu Designer shown in [Figure 3–14](#). If you look at the rules for some existing items, principally in the Navigation, TableIO, and Commit groups, you can see examples of action rules that are defined as a standard part of the framework.

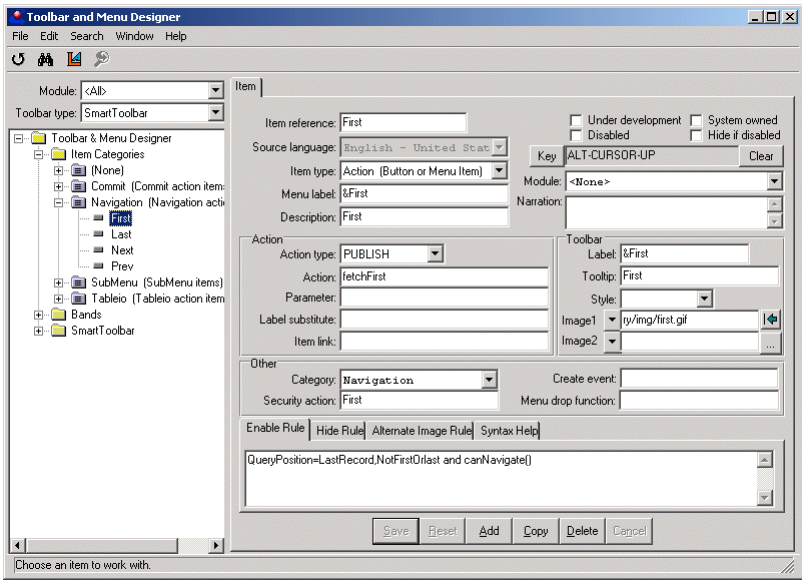


Figure 3–14: Toolbar and Menu Designer

This figure shows the enable rule for the First button or menu item in the Navigation group. What it means is that if the query position is either on the last record or on some record that is not first or last, and the window is in a state where the user is allowed to navigate records, then enable the First item.

QueryPosition is an SDO property that is set whenever a query is opened and whenever the user selects a record or navigates to a new record. Its values are discussed below, along with other useful properties that are frequently used in action rules.

The **canNavigate** function does a fairly complex analysis of the state of all the Objects in the window to determine whether the user should be permitted to move to another record. If an update is pending in a record in the SDO being navigated or in any dependent SDO, for example, then navigation is disabled in the window until the update is completed. This function returns True or False depending on the state of things, and this helps the Toolbar determine whether the navigation buttons should be enabled.

Another example of an **enable rule** is the Save button in the TableIO group:

`NewRecord=add,copy or DataModified`


This means that if the NewRecord property indicates that there is either an Add or a Copy in process, or the Data Modified property indicates that an existing record has been modified, then enable the Save button.

There are no actively used items that have a hide rule, but you can define your own rules to hide unwanted items based on the state of the application at runtime.

An example of an **alternate image rule** is this one for the Comments item:

```
hasActiveComments()
```

This invokes a special function that returns True if the current record has an active Comments record associated with it. If it does, then it displays its alternate image, a Comments button with

a checkmark on it  so that the user knows that there are Comments to be viewed.

Be cautious about modifying the action rules for standard buttons in the framework, since you change their behavior everywhere they are used. However, there are cases where you might want to modify the action rules for standard buttons. For example, if you want to keep the SAVE button enabled at all times and in all places in your application when there is anything updatable in the window, rather than having it enabled only when a user has begun to enter changes into a field, you can do this by altering the action rule for the Save button from:

```
NewRecord=add, copy or DataModified
```

To this:

```
ObjectMode=Update,Modify
```

The ObjectMode property of a Viewer can be view if it is read-only, update if it can be enabled by pressing a button, or modify if it is always enabled for input. You can force the Save button to be always enabled by checking for either Update or Modify as the ObjectMode value.

NOTE: Since menu and toolbar information is cached, it is necessary to clear the cache to notice your changes. You can either restart the session or clear the cache by deleting the persistent procedure adm2/toolbar.p using the Procedure Object viewer. Ensure no windows that use a toolbar are open in the AppBuilder when deleting that procedure.

3.4.1 Understanding the role of the item link

It is important to understand where these functions are executed and where property values are retrieved. When you define an item in the Toolbar and Menu Designer shown in [Figure 3–15](#), you must define an Item Link for it. Any Toolbar that uses this item must be a Source for that link type in order for the item to be active. The evaluation of the properties and functions in action rules is done in the Target for the link. In many cases the Item Category allows you to define a default value for the Item Link that applies to all items in that category unless they specify otherwise.

The Item Link Default for the Navigation category, for example, is Navigation. The Navigation buttons communicate with objects that are Navigation-Targets of the Toolbar.

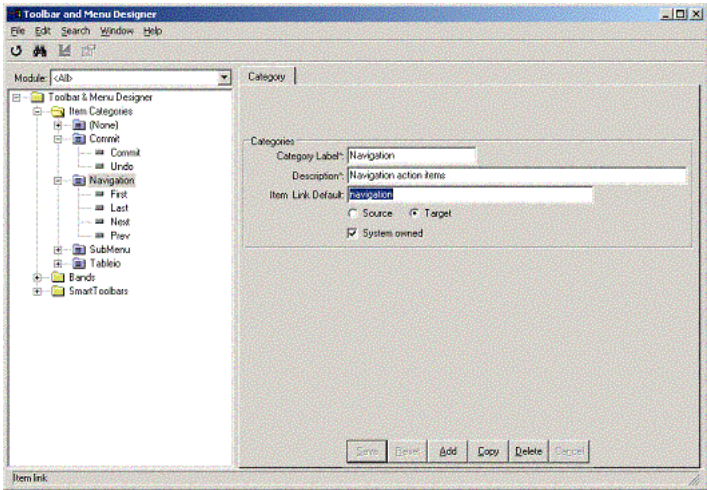


Figure 3–15: Toolbar and Menu Designer: item link definition

If an action does not have an item link defined, and its category also does not have an item link defined, then the event is processed on the toolbar’s container by default.

If there is no default specified for the Category, then you must define an Item Link, as shown in [Figure 3–16](#), for the item if it has action rules, or if it publishes an event as its action, as the Comments item does.

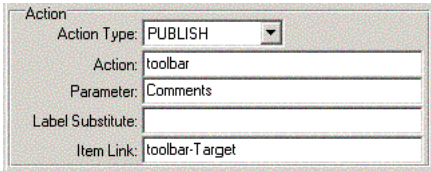


Figure 3–16: Defining an item link

In this case, when you enter the Item Link for an individual action, you enter the full link name including the –**Target** extension.

3.4.2 Syntax of the action rules

An **action rule** contains a delimited list of function references and properties that return a logical result. The syntax for rules is as follows:

```
[ property | function ] = list [ AND | OR ] ...
```

A **property** is the name of a SmartObject property that is retrieved across the specified Item link. If the property is of datatype LOGICAL, and you want the TRUE value to be used in the expression, then you simply include the name of the property in the rule. For example, **DataModified** means **If DataModified is True**. If you want to use the negation of the property in the expression, then you must include it as a value, following an equal sign, as in **DataModified=no**.

If the property returns some other value, then you specify a value or a comma-separated list of values to match. A property reference with a list of values evaluates to True if the property value matches any value in the list.

A **function** is the name of a Progress 4GL function that is executed across the specified Item link. You can only use functions that return a LOGICAL value. To differentiate a function reference from a property, follow the name of the function with empty parentheses, as in **canNavigate()**.

A **rule** can be built up from any number of elements, each involving a single property or function. Separate the elements by the keyword AND or OR. Complex expressions requiring grouping of elements with parentheses cannot be used in action rules. If you need to perform a calculation complex enough that the action rule syntax is not sufficient for it, then you should define a function of your own in the Item Link target that does the calculation and returns True or False, and use it in your action rule.

3.4.3 Properties typically used in action rules

You can use any property or function you want in an action rule, but there are a handful that are most likely to be useful. These are all used in existing action rules in the Navigation, TableIO, or Commit categories. Here are some useful properties:

- **QueryPosition** — This SDO property identifies where in the dataset the cursor is positioned. It is generally used to identify which navigation items to enable. Its possible values are:
 - **FirstRecord**
 - **LastRecord**
 - **OnlyRecord** — There is only one record in the dataset.
 - **NotFirstOrLast** — The cursor is somewhere in the middle of the dataset.
 - **NoRecordAvailable** — The dataset is empty.
- **RowObjectState** — This SDO property identifies whether a modified record has been saved on the client without being returned to the server. This would normally be true only if there is a Commit-Source (such as Toolbar with a band of Commit and Undo buttons) in the window that causes updates to be cached on the client until a Commit event happens. So the Commit and Undo buttons use these two values to determine whether they should be enabled or not:
 - **NoUpdates** — No updates have been saved without being committed (so Commit and Undo buttons should be disabled, for example).
 - **RowUpdated** — At least one record has been saved without being committed (so Commit and Update buttons should be enabled).
- **RecordState** — This property, defined for data visualization SmartObjects, indicates whether there is a record available to display or update. Its possible values are:
 - **RecordAvailable** — There is a record available for display or update.
 - **NoRecordAvailable** — There is no record available for display or update. This normally means that the dataset of the associated SDO is empty. This could be the case, for example, when the user is adding OrderLines for an Order and the first OrderLine has not yet been added. In this state the application can allow an Add operation, but not an Update or Delete.

- **NoRecordAvailableExt** — This special value indicates that not only is there no record available in the current dataset, but there is no record available in the parent dataset either (The ext suffix means extended). This is useful information because it signals to an update band that it is not appropriate to enable even an Add operation. For example, if the current SDO is an OrderLine SDO, and is a child of an Order SDO, and there is no currently selected Order (perhaps because the current Customer in the SDO yet another level above has no Orders), then it is not appropriate to allow the user to add an OrderLine because there is no Order key to assign to it. This is why the action rule for the Add button is this:

```
RecordState=RecordAvailable,NoRecordAvailable and Editable and
DataModified=no and CanNavigate()
```

At first it may not seem sensible to check that the Record State is either RecordAvailable or NoRecordAvailable, but it is for exactly this reason that this is the case. The one remaining condition is that the Record State is NoRecordAvailableExt, and in this case the Add button is not enabled.

- **Editable** — This LOGICAL property is defined for data visualization SmartObjects, and is True if the current target SmartObject is editable, that is, can be used for an Add, Copy, Update, or Delete. This is normally the case if the Object has any enabled fields, or if it is a GroupAssign-Source for some other Object that has enabled fields.
- **DataModified** — This LOGICAL property is defined for SDOs, and is True if the current record in the dataset has been modified but not saved. As soon as a user begins to type into an enabled field in a Viewer or an enabled cell in a Browser, a trigger sets this property to True and notifies the Toolbar that is its TableIO-Source. The result is that the Save and Cancel buttons are immediately enabled.
- **NewRecord** — This property is defined for data visualization SmartObjects. It indicates whether the currently displayed record is a newly created record, that is, one that has not yet been saved back to the database. It is a CHARACTER property rather than LOGICAL because it can have three possible values:
 - **No** — For an existing record
 - **Add** — For a new record created by an Add operation
 - **Copy** — For a record newly created by a Copy operation

- **ObjectMode** — This data visualization property is new to the ADM2 with Progress Dynamics. It is used by some of the Toolbar types that contain View and Modify buttons that toggle the enabled state of the Object, as well as a Save button to actually save changes. Its three possible values are:
 - **Modify** — The Object is in Modify mode, with fields enabled but no changes underway.
 - **View** — The Object is in View mode, with fields disabled.
 - **Update** — The Object is in Save mode, with fields enabled and changes in process, so that the Save button is also enabled to save the changes.
- **FilterActive** — This LOGICAL property is True if there is currently a filter applied to the SDO that is being browsed. This is used, for example, in the alternate image rule for the Filter button itself, to display the image with the checkmark. You might also use it to determine whether all data for a table is being retrieved or only a filtered subset of the data.

3.4.4 Functions typically used in action rules

There are also several functions that are used in existing rules. Remember that a function used in a rule is identified by its parentheses and must return a LOGICAL value.

- **canNavigate()** — This LOGICAL function determines whether the user should be free to navigate to another record in the dataset (whether by pressing a Navigation button or selecting a record in a Browser). The function returns No if an update is in progress and unsaved anywhere within the chain of dependent data objects.
- **hasActiveAudit()** — This LOGICAL function returns True if the current record has one or more active audit records associated with it.
- **hasActiveComment()** — This LOGICAL function returns True if the current record has one or more active comment records associated with it.

3.4.5 Defining rules that use your own functions and properties

You can define new properties or functions to use in your own action rules, if existing ones are not sufficient to express the criteria you need to enable and disable, hide and view, or change the image on Toolbar items. In particular, if your application logic needs to perform some fairly specific or complex operation to determine the proper state of Toolbar items, you may want to define a function that does the calculation and returns True or False accordingly. This is similar to the `modifyDisabledActions` discussion and example earlier in this chapter.

If you are modifying the behavior of existing buttons such as the `TableIO` buttons that are used throughout the framework, it is almost certainly not a good idea to change the definition of their rules in the Toolbar and Menu Designer. In fact, one of the key points you must keep in mind when you define action rules is that all of the elements of the rule must be available for evaluation anywhere the item is used. If the property or function is not defined, it will be considered to be False, which means for example that an enable rule with such an expression would never be True and the button would never be enabled anywhere in the framework or any application module where it is used.

It is more appropriate to define action rules for new Toolbar items you create for your application. Then you can associate whatever properties and functions you need, as long as you are sure they will be available to evaluate wherever the item is used.

Caching Application Data on the Client

Caching of certain types of application data on the client improves application performance. This chapter discusses the types of caching provided by the Dynamics framework and strategies for using the features. It includes these topics:

- [Why cache?](#)
- [Data types and cache types](#)
- [Mechanics of caching](#)
- [Enabling caching for SDOs](#)
- [Enabling caching for a dynamic lookups](#)
- [Enabling caching for a dynamic combo](#)
- [Using the Dynamic Launcher](#)
- [Cache APIs](#)

NOTE: This chapter discusses caching dynamic combos and dynamic lookups. If you have pre-Version 2.1B custom code that uses the ADM2 APIs for DynCombo and DynLookup, you need to be aware that these APIs have changed. See the [Progress Dynamics Update Bulletin Version 2.1B](#) for a complete listing of the changes. Progress strongly recommends updating your existing code to use the new APIs as quickly as possible. In addition, in the performance section of the [Progress Dynamics Administration Guide](#), there is more information on these features and overall performance.

4.1 Why cache?

The frequency of data requests and the amount of data sent across the wire impact the performance of distributed and thin network clients. An application's performance can be drastically improved if frequently accessed data is cached on the client, allowing the application's data objects to operate on data in the cache rather than relying on repeated server calls. This approach not only reduces network traffic, but it also decreases the demand on the AppServer's resources and thereby increases the application's scalability.

The two most important development issues with caching are:

- Data in the cache becomes stale as the underlying database data changes.
- Too much data in the cache may have a negative impact on performance.

Therefore, caching is most applicable and valuable for data that are frequently used, relatively static, and have low volume. Since there is no generic way to identify data that will most benefit from caching, it is up to you to enable caching for suitable data and thereby improve the overall performance of your application.

Prior to Progress Dynamics Version 2.1B, the framework supported caching features devoted to managing data navigated by dynamic lookups and dynamic combos. Version 2.1B implements a new type of caching at the SmartDataObject level. Dynamic lookups can continue to use the existing lookup-based caching, and this functionality has been enhanced to provide better performance in Version 2.1B. SDO-based caching does not apply to dynamic lookups. Dynamic combos can continue to use the combo-level caching, or they can use the new SDO-level caching.

4.2 Data types and cache types

To understand the caching capabilities of the framework, let's start with some common definitions. First, here's a way of categorizing your data to assess its suitability for caching. Categories of data include:

- **System data and administration data** — This is the read-only data that defines your application, such as entities and user preferences. In Dynamics, any caching of this information is handled by the Repository and the framework. This type of data and caching is not covered here. See the chapters on Dynamics managers for more information.

- **Non-transactional data** — Application data that can change on the client, but when it does, it is unlikely to affect other data records (for example, records holding descriptive information). This information is a good candidate for Dynamics caching if it meets the following criteria:
 - It is infrequently modified.
 - It is frequently accessed.
 - There is a low volume of the data.
 - Loading this data into the cache at application startup does not degrade start-up performance below your required benchmarks.

NOTE: Although the framework caching support makes a somewhat suitable SDO for handling genuinely static data, there are simpler solutions. For example, true constant data like a list of weekdays or a list of months can be defined in widget or object attributes, global variables, external files, hard-coded source and so on. This approach avoids AppServer hits completely.

- **Transactional data** — The data that results from application transactions. Transactional data is application data found on the client, but it may not be well suited for caching, because:
 - It has high volume.
 - It may be child data.
 - It may be frequently modified.

All these reasons mean that the data will change frequently, force frequent updates of the cache, become a performance drain, and thereby defeat the purpose of client-side caching.

Next, there are many types of caching schemes, each with its uses. Types include:

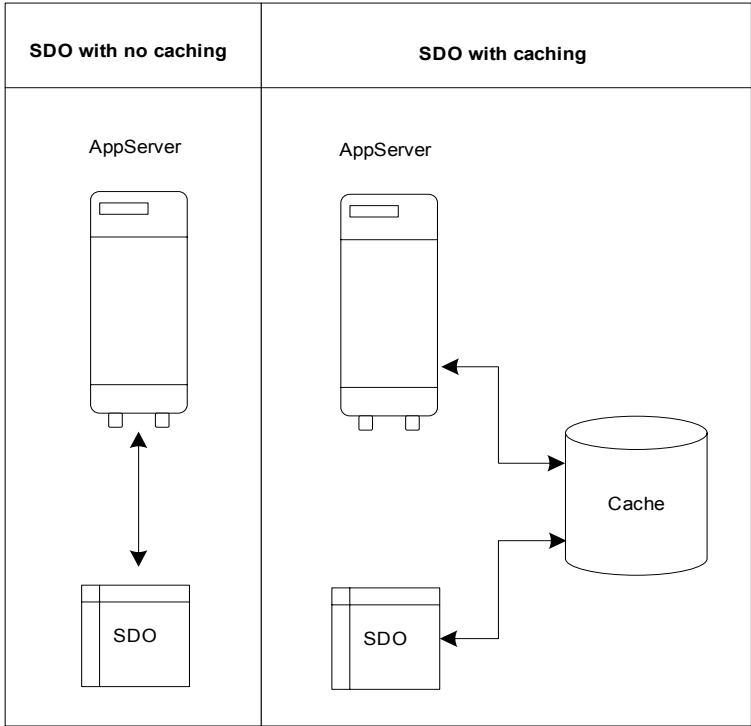
- **Persistent cache** — Data can be stored on the client side persistently and will be accessible at the client side from session to session. Progress Dynamics does not support persistent caching with SDO-based application data caching. There are cache APIs that will make it easier for you to develop your own persistent caching scheme. See the API section at the end of the chapter for an overview of the available hooks.
- **Session cache** — All data in the cache is destroyed when the session ends. When a new session starts, data needed again retrieved from the server. The framework cache for client-side application data is a session cache.

- **Timed cache** — Cached data is only valid for a specific time and must be flushed after this time. Either a persistent cache or a session cached can be a timed cache. In Dynamics you can configure the session cache to be a timed cache or a cache that lasts the entire session.

NOTE: Framework caching or application data is a **runtime** feature. Since no application caching is done in development mode, you'll need to test your caching schemes during runtime.

4.3 Mechanics of caching

The SDO is the data managing interface between the AppServer and your client application. Enabling a cache is like placing a bucket between the AppServer and the SDO, as shown in the Figure below:



The result set of the SDOs query is stored as a temp-table in the cache and any reference to that SDO forces the SDO to check the cache, find the temp-table, and retrieve the data. When the necessary data is not available or has expired, the cache will get what it needs from the AppServer.

The framework accomplishes this caching not by implementing a true physical cache in memory, but by using the seamless versatility of temp-tables. Since dynamic temp-tables received as a parameter are scoped to the session by default, they are completely capable of being used as a cache. Temp-tables are active, so SDOs can operate directly on cached data. There is no need to duplicate data and no need to synchronize updates with the cache. The cached data can be navigated and browsed independently by multiple instances by utilizing the 4GL's ability to define separate buffers on a temp-table

For SDOs that are defined with statically defined temp-tables, they will not be able to store or use data from the cache as efficiently as dynamic SDOs. A static temp-table is scoped to the procedure that defines it, so it is necessary to either keep the defining procedure running or keep a dynamic copy of the temp-table in the cache. In the cases where a SDO with a static temp-table definition is running on the client session, the static data will be copied to and from the cache.

4.3.1 Cache versus a shared cache

There are two important notions of how the cache should operate when confronted with multiple instances of one SDO. The first notion is *simple caching*. Here, the first instance of an SDO creates a record in the cache and that record will remain in the cache for the entire session, or, optionally, for a specified period of time. Each subsequent instance of the SDO ignores existing cache records and creates its own cache record. This essentially means that each new instance of the SDO will hit the AppServer for its own result set and store it in the cache. This is not a common use case.

Simple caching is enabled and configured with the `CacheDuration` property of the SDO. This property accepts an integer value:

- If this property has a value of zero, then simple caching is disabled (the default).
- If it has an integer value greater than zero, then the SDO's cache becomes a timed cache with a duration equal to the number of seconds specified by the integer. The cache records will be destroyed when two things happen: the last instance using the cache record is destroyed and the amount of time specified expires.

NOTE: Should two instances specify different values for a timed cache, the shortest duration specified is the duration the `CacheManager` will use.

- If it is set to the Progress unknown value (?), then the data in the SDO is cached for the life of the session.

The second notion is called *sharing*. If a cache is shared, then the first instance of the SDO will create the cache record, and each subsequent instance will use the data in that cache record. If When all instances of the SDO are closed, the cache record is also destroyed. The first new instance of the SDO will create a new cache record.

Shared caching is enabled with the logical Shared property of the SDO:

- If this property is true, then the SDO's cache is a shared cache.
- If this property is false, then the SDO's cache is not a shared cache.

NOTE: When both types of caching are enabled, the caching occurs at the SDO-level and the data is shared among all instances. This cache will be valid for the session.

4.3.2 Interaction of the CacheDuration and Shared properties

The most likely use case will be setting both of the properties. When the CacheDuration property is not equal to zero and the Shared property is true, then cache data is shared, but the cache is not cleared when the last instance of an SDO is closed. The cache data will be available when the next instance of the SDO runs.

Normally, with Dynamic SDOs, you would expect both properties to be set. There may be a case for setting CacheDuration and setting Shared to false. It is conceivable that data values may vary due to differences in instances, for example if a calculated field expression includes external instance data. This is also the only valid cache mode for cases where the static proxy is running on the client, or in sessions with database connections with the full static SDO running on the client.

4.4 Enabling caching for SDOs

The SmartDataObject Properties dialog for an SDO instance allows you to configure caching for an SDO. You use the **Share data** (Shared property) and **Cache data** (CacheDuration property) options to accomplish this.

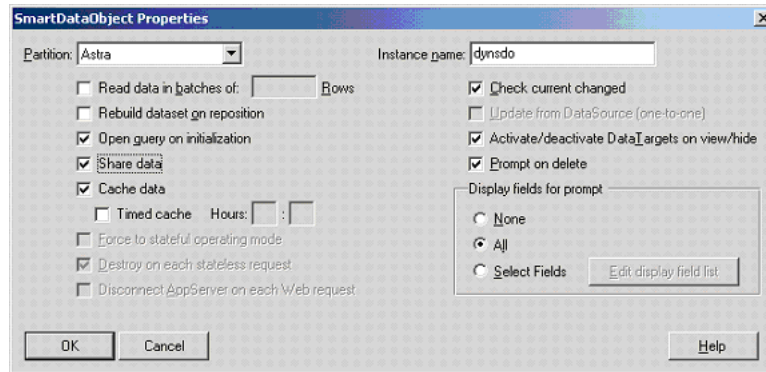
First, these two options are not available if:

- **Partition** is set to **None**.
- **Read data in batches** option is selected.

In other words, caching does not make sense if no AppServer partition is available and if the SDO will be batched.

To configure SDO caching:

- 1 ♦ Select an SDO instance, right click, and choose **Object Properties**. The SmartDataObject Properties dialog appears:



- 2 ♦ Select **Cache data**. SDO data will now be cached for the whole session.
- 3 ♦ (Optional) If you want a timed cache, select **Timed cache**, and enter a value for **Hours**, for minutes, or both.
- 4 ♦ Select **Share data**. Cached SDO data will now be shared among all instances.

4.5 Turning on caching for dynamic lookups and combos in a session

You can control whether or not caching is enabled for dynamic combos and lookups on the field-level. To start out, caching of dynamic lookups and dynamic combos is disabled. For each session type where you want to use caching for either or both, you need to set a session property on the applicable session type.

Set `field_cache_options` to one of the following values:

- **ALL** — Enables caching for all field types. Currently this is only DynCombo and DynLookup.
- **DynCombo** — Enables caching for DynCombo only.
- **DynLookup** — Enables caching for DynLookup only.

- NOTE:** Caching batched data or child data is not supported by the framework.

Dynamics supports a type of behind-the-scenes caching for dynamic lookups. The result set of the query for the SmartDataField is kept in one temp-table and another temp-table manages the subsets of this data being used in each instance of the dynamic lookup. It improves performance by ensuring each instance does not need to store the entire database result set.

1 ♦ Select **Build→Smart Data Field Maintenance**.

- 2 ♦ Enable the **Use cache** option.
- 3 ♦ **Save** and exit.

4.7 Enabling caching for a dynamic combo

The dynamic combo has two caching mechanisms available. The first caching mechanism uses temp-tables to cache data at the SDF level. The dynamic combo allows you to specify an SDO as its data source, and then SDO-level caching is available to the object.

4.7.1 Enabling SDF-level caching

Dynamics supports a type of behind-the-scenes caching for dynamic combos. The result set of the database query for the SmartDataField is kept in one temp-table and another temp-table manages the subsets of this data being used in each instance of the combo. It improves performance by ensuring each instance does not need to store the entire database result set.

To enable this type of caching for a dynamic combo:

- 1 ♦ Select **Build→Smart Data Field Maintenance**.

The screenshot shows the "SmartDataField Maintenance (Master Properties)" window. The "SmartDataField:" dropdown is set to "SalesRepCombo". The "SDF type:" is "DynCombo". The "Super procedure:" field is empty. The "Description:" is "Salesrep Combo". The "Data source:" has two options: "Database query" (selected) and "DataObject". The "DataObject name:" field is empty.

The "Product module:" is "sm-module / smoke module". The "Field name:" is empty. There are three checkboxes: "Use cache" (checked), "Where Used" (disabled), and "Synch Instances ..." (disabled).

The "Dynamic Lookup" tab is selected. Under "Specify base query string (FOR EACH)", it says "for each salesrep no-lock". To the right, under "Query tables", there is a list containing "salesrep" and buttons for "Clear" and "Refresh".

Field name	Display seq.	Column label	Data type	Format
salesrep.MonthQuota[10]	0	Region	integer	
salesrep.MonthQuota[11]	0	Region	integer	
salesrep.MonthQuota[12]	0	Region	integer	
salesrep.MonthQuota[1]	0	Region	integer	

At the bottom, the "Details" tab is selected. It contains several fields: "Key field:" (set to "salesrep.Salesrep_Obj"), "Description substitute:" (&1 / &2), "Field label:" (set to "SalesRep" with a "No-label" checkbox), "Tooltip:" (set to "Pick a SalesRep"), and "Inner lines:" (set to "5"). On the right side of the "Details" tab, there are more settings: "Datatype:" (decimal), "Format:" (a long string of ">" characters), "Field width:" (20.00 with an "Enable field" checkbox), "Build sequence:" (1 with a "Display field" checkbox), and a "Sort" checkbox which is unchecked.

- 2 ♦ Make sure the **Data source** radio set has **Database query** selected.

- 3 ♦ Enable the **Use cache** option, which is actually enabled by default.
- 4 ♦ Save and exit.

4.7.2 Enabling SDO-level caching

SDO-level caching improves on SDF-level caching in some situations. When the following behavior is desired or improves performance, use SDO-level caching. Use it when:

- Visual objects that are not combo-boxes may also use the data.
- Combo boxes with different list formatting may use the same data.
- The initial retrieval of data needs to be done in the same AppServer hit as the rest of the data in the container. (A single lookup in a viewer will negate the reduction of Appserver hits.)
- You need SDO behavior like calculated fields or temp-table support, and so on.

The main difference with SDO-level caching is that the dynamic combo list (list-item-pairs) need to be built each time a dynamic combo is instantiated. SDF-level caching builds this list on the server and caches the list. Each instance then copies the list exactly. Using an SDO-level cache data incurs a slight overhead since it needs to build the list for each instance at start up.

To set SDO-level caching:

- 1 ♦ Select **Build**→**Smart Data Field Maintenance**.

SmartDataField Maintenance (Master Properties)

File Window Help

SmartDataField: Product module:

SDF type: Field name:

Super procedure:

Description:

Data source: ☐ Database query ☒ DataObject

DataObject name:

Dynamic Lookup:

Specify base query string (FOR EACH)

Query tables:

Clear Refresh

Field name	Display seq	Column label	Data type	Format
Contact	2	Contact	character	x(30)
Country	0	Country	character	x(20)
CreditLimit	0	Credit Limit	decimal	->.>>>.>>9
CustNum	0	Customer number	integer	>>>>9

Details Other

Key field: Data type:

Description substitute: Format:

Field label: ☐ No-label

Field width: ☒ Enable field

Tooltip: Build sequence: ☒ Display field

Inner lines: ☐ Sort

- 2 ♦ From the **Data source** radio set, select **DataObject**. The Query section is disabled, and the SDF Maintenance tool enables the **DataObject name** field.

NOTE: When changing an existing dynamic combo from a database query to an SDO, the tool attempts to retain the fields displayed and key field of the dynamic combo. It removes the table name qualifier from the fields displayed and from the key field and attempts to find matching columns in the SDO. If it finds matching columns, it uses those. If it does not find matching columns, it discards the query source data. When changing from an SDO data source to a database query, the tool makes no attempt to retain the fields displayed or the key field of the dynamic combo.

- 3 ♦ Enter the SDO name in the **DataObject name** field. When you enter a valid SDO, the tool updates the field list below to display the columns of the selected SDO. The **Key** field combo box contains the columns of the selected SDO.

- 4 ♦ Unselect the **Use cache** option. For an explanation of what occurs when **Use cache** is enabled with **DataObject** selected, see the next section.
- 5 ♦ Save and exit.

4.7.3 Enabling both

When both types of caching are enabled, the caching occurs at the SDO-level and the data is shared among all instances. This cache will be valid for the session. [Table 4–1](#) describes the other valid combinations.

Table 4–1: Caching feature combinations

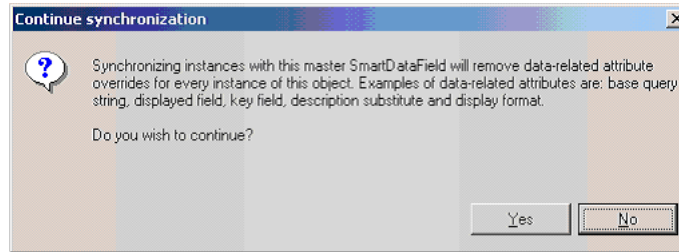
Use Cache	SDO data source	Cache/Shared
Yes	Yes	SDO-level cached and shared. Cache duration is ? (valid for session).
No	Yes	Shared and not cached. Cache duration is 0.
No	No	Not cached and not shared.
Yes	No	Field-level cached and not shared.

4.7.4 Synching instances

When you change a dynamic combo from a database query data source to an SDO data source, instance overrides of data-related properties may cause the dynamic combo to fail at runtime.

Consider the following example: a master dynamic combo has a `DisplayedField` definition set to “customer.custnum,customer.name”. An instance of the dynamic combo overrides this value with “customer.name,customer.contact”. Changing the master to use an SDO data source changes `DisplayedField` to “custnum,name”. The instance override of `DisplayedField` will now fail at runtime because the field names are qualified with the table name and should not be.

The **Synch Instances** button is a tool for removing all overrides of date-related properties in all instances of the SDF. When clicked, it displays a warning message explaining how it works:



Data-related attributes includes these properties:

- BaseQueryString
- DescSubstitute
- DisplayDataType
- DisplayedField
- DisplayFormat
- KeyDataType
- KeyField
- KeyFormat
- QueryTables
- DataSourceName
- BrowseFieldDataTypes
- BrowseFieldFormats
- BrowseFields

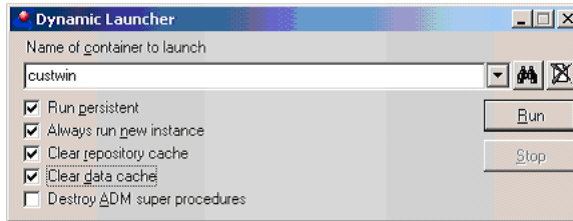
This **Synch Instances** function works with dynamic lookups, too, although its real purpose is to help convert dynamic combos to SDO data sources.

4.8 Using the Dynamic Launcher

The latest version of the Dynamic Launcher has features useful for working with data caching in the Development environment.

First, to clear existing data in a cache, enable the **Clear data cache** option before clicking **Run**.

Second, in versions prior to Version 2.1B, the Dynamic Launcher could only launch a single instance. Now, if you enable **Always run new instance**, when you click **Run** again for the same object, Dynamics launches a second instance. This feature is enabled by default.



4.9 Cache APIs

The cache interface provides methods to allow dynamic temp-tables to serve as a client cache. It is implemented in `adm2/cache.p`, which is added as a method library to the data class. A call to `getManagerHandle('CacheManager')` in any smart object will return the data class procedure, allowing you to call the cache APIs directly in the data class as if it were a manager. Note that Dynamics `getManagerHandle` will not return this handle. The cache interface can, however, be extended and implemented as a Dynamics manager with the name 'CacheManager'. In this case, the ADM2 calls to the cache interface go to the extended manager.

4.9.1 Programming notes

The following information is provided to help you evaluate how the cache can be extended:

- Application data caching is not a generic option for caching temp-tables.
- Caching is configured by enabling it on each SDO you want to cache. The framework does not automatically cache SDO data.
- Cached data should be based on result sets (for example, `RowObject` temp-tables), not physical database tables.
- Data is cached on demand.
- Cached data can be cleared or refreshed.

- Client side modifications must synchronize with the cache.
- A modified local cache will not refresh the cache for other users using their own local cache.
- The framework does not support refreshing local caches from the server. To implement this type of feature, you will need message-oriented middleware, like Sonic MQ.

4.9.2 Cache key

All the APIs need a Key to identify the SDO and thereby access the cache associated with that SDO.

- The framework uses the `LogicalObjectName` of the SDO as the Key to the cache.
- If the `LogicalObjectName` is not defined, such as when running the code without Dynamics, the framework uses the master file name.

4.9.3 registerCacheItem

Registers the referenced temp-table in the data cache with the passed key as the identifier. Use the key later to reference the item.

NOTE: The assumption is that the table is scoped to the session, as is the case after a call to the server with the table as an output parameter.

SYNTAX

`registerCacheItem (Key, Handle, Time, NumRecords, Context)`

Parameters:

- **Key** — Identifier.
- **Handle** — The temp-table handle to add to the cache.
- **TimeSpan** — Specifies the number of seconds the data in the cache should be considered valid. The framework will pass this from the `CacheDuration` property on the `DataObject`.
- **NumRecords** — (Optional) Specifies the number of records in the data.
- **Context** — (Optional) Info needed for server to refresh the cache.

4.9.4 findCacheItem

Returns true if the framework finds an unexpired item with the passed key.

SYNTAX

```
findCacheItem returns logical (Key, MaxAge)
```

Parameters:

- **Key** — Identifier.
- **MaxAge** — Optional maximum age in seconds. Normally, this value should be shared by all instances of the object, but instances can override it and determine how long to cache data.

4.9.5 fetchDataFromCache

This procedure copies data from the cache and returns a table-handle with all the copied data.

SYNTAX

```
fetchDataFromCache (Key, OUTPUT table-handle)
```

Parameters:

- **Key** — Identifier.
- **table-handle**—The found data returned as a table handle.

4.9.6 createSharedBuffer

Creates a shared buffer from the cache and increments the count of current SDO instances. The framework uses the SDO instance count to correctly track whether or not the data is in use.

SYNTAX

```
createSharedBuffer returns handle (Key, CurrentBuffer)
```

Parameters:

- **Key** — Identifier.

4.9.7 destroySharedBuffer

Call this function when you no longer need a shared buffer from the cache. The function decrements the number of instances and deletes the data from the cache if the number of instances is 0.

SYNTAX

```
destroySharedBuffer returns logical(CurrentBuffer)
```

Parameters:

- ***CurrentBuffer*** — The buffer you want to remove.

4.9.8 clearCache

Use this function to clear the cache.

A shared cache (active) requires logic to deal with SDO instances running when you clear the cache. Cached, shared temp-tables still in use by at least one instance will not be physically deleted. Instead, they will be marked as dirty. With this scheme, running SDO instances continue to operate. Any new request for data will refresh the cache. The dirty cache will be physically deleted when the last instance that shares its data is destroyed.

SYNTAX

```
clearCache (Key)
```

Parameters:

- ***Key*** — Identifier. Supports '*' as a signal to clear all cached data.

NOTE: A new instance will not share data with old instances that are running with a dirty cache.

4.9.9 Modifying the instance properties

When an SDF uses an SDO as its data source, neither the master nor instance properties of the SDO are available for modification, guaranteeing a fast start up for the dynamic combo. You can, however, programatically alter the SDO instance attributes. The dynamic combo stores the SDO name in the DataSourceName property. With this in hand, you can override the createDataSource() function of the dynamic combo. The createDataSource function starts the SDO as an instance in the container (Viewer) and adds the data link from the SDO to the dynamic combo. The defaults of the available SDO instance properties are shown in [Table 4–2](#).

Table 4–2: Instance properties and defaults of an SDO data source

Property	Default
RowsToBatch	0
Shared	True
CacheDuration	<p>If the dynamic combo’s UseCache property is true, then the unknown value (?) is the default. This denotes an SDO that uses an untimed session cache.</p> <p>Otherwise, the default is zero, denoting that there is no cache.</p> <p>NOTE: The DynCombo class UseCache property has a default of true.</p>
DataLogicProcedure	<p>Not used.</p> <p>The framework uses a thin dynamic proxy without a DataLogicProcedure on the client for all SDOs, including static ones.</p>

Using ADM2 Properties and Methods In Progress Dynamics

This chapter provides an overview of the properties and methods of the Application Development Model, Version 2 (ADM2), that are essential to Progress Dynamics application builders.

- Effectively use the Progress Dynamics framework.
- Quickly distinguish the properties and methods meant for public use from those meant to support the internal workings of the framework or specialized applications.
- Take advantage of useful supporting methods (internal procedures and functions) that you can call directly from your code.
- Understand which methods are typically overridden and which are called directly.

The chapter also provides guidance on how methods are typically used. Some methods are simply internal procedures and functions for you to run from your code. Others are candidates for overriding, meaning that typically you write a local version of the method to extend its behavior and then invoke the standard behavior with a RUN SUPER statement. Another type of method is an *event procedure*. An event procedure is implemented as an internal procedure that responds to PUBLISH statements in the framework. Event procedures are localized or overridden. This chapter describes the event procedures that are the most likely candidates for localization and provides guidance on how to localize them.

The chapter organizes the information by task, but you are sure to find uses for some of the properties and methods in many other contexts. The following sections describe properties and methods in these task categories:

- **Getting basic information** — many properties, and some methods, return useful information about objects, their class, whether they are enabled, their contents, and so on.
- **Starting Progress Dynamics application windows** — understanding the sequence of events that occurs when the framework creates and initializes objects in a container allows you to intercept the process and customize or extend it for your needs. This section describes the relevant methods that you can call and properties that you can set or read to affect application startup.
- **Managing links in Progress Dynamics applications** — This section reviews the properties and methods that the framework uses to manage the links between Objects. It also describes how you can use and extend the properties and methods for your applications.
- **Customizing and managing queries** — This section instructs you on the best ways to modify the WHERE clause of a query; determine when it is opened and closed; position and refresh queries; and so on.
- **Paging methods and properties** — A number of properties and methods specifically support multi-page application windows, and this section reviews those.
- **Special functions that manage properties** — Several functions let you define new properties at run time and also return information about standard object properties.
- **General-purpose methods** — The framework provides a few special methods to set and retrieve properties whose values are complex enough that they cannot easily be used directly. This section discusses these methods.

For more information on ADM2 properties and methods, refer to the *[Progress Dynamics ADM2 API Reference](#)*.

5.1 Getting basic information

Properties and methods return useful information about objects, including their classes, whether or not they are enabled, and their contents. This section contains information about the following object functions:

- [Object names and types](#)
- [Testing whether objects and fields are enabled](#)
- [Field and widget lists](#)
- [Useful object handles](#)
- [Functions that return field values and attributes](#)
- [Other useful data object functions](#)

5.1.1 Object names and types

This section describes properties that provide information about types and names.

ObjectType

This CHARACTER property, available for all SmartObjects, returns the generic SmartObject class name for the object. For example: SmartWindow™, SmartDataObject, SmartDataViewer, SmartDataBrowser™, SmartDataField, and so on.

ObjectName

The ObjectName property is the unique instance name of the Object for that Container, or it defaults to the actual procedure file without path and extension. In the case where an Object is used more than once, this may have a numeric digit added to it to distinguish among different instances, or its instance name may be changed by the developer in assembling the Container.

PhysicalObjectName

This CHARACTER property, available for all SmartObjects, returns the name of the procedure file executed to run the object, including the filename extension, but not including its relative pathname. For a static object, this is the name of the object procedure, such as `custviewer.w`. For a dynamic object, it returns the name of the driver procedure that instantiates objects of that type. For example, it returns `rydynview.w` for dynamic Viewers and `rydyncontw.w` for dynamic windows, and so on.

LogicalObjectName

This CHARACTER property, returns the logical name of the object, which is its name as represented in the repository. For a dynamic object, this is the name the object was given when it was created, either by the Object Generator, or by a developer using the AppBuilder. This property may be blank for objects running without a repository.

For example, it could be `custbrowsewin` for a dynamic window or `customerfullb` for a dynamic SmartDataBrowser.

ContainerType

This CHARACTER property, available for all SmartObjects, indicates whether the object is a container or not, and if so, what kind of container it is. For non-container objects such as Browsers and SDOs, its value is blank. For Viewers and SmartFrames™, it returns FRAME. For SmartWindows, it returns WINDOW. For non-visual container objects such as SBOs, it returns VIRTUAL.

NOTE: Remember that SmartDataViewers are considered containers because you can drop field-level objects such as SmartDataFields into them.

QueryObject

This LOGICAL property, available for all SmartObjects, returns TRUE if the object manages a query. In other words, it returns TRUE for SDOs and SBOs and FALSE for all other standard objects.

5.1.2 Testing whether objects and fields are enabled

Your application enables and disables objects and the fields and widgets they contain during the course of an application session. The ObjectEnabled and FieldsEnabled properties let you query the current state of the objects, fields, and widgets.

ObjectEnabled

This LOGICAL property, available for all visual SmartObjects, returns TRUE if the object itself has been enabled, and FALSE if it has not.

Normally, the framework enables every SmartObject as the application window initializes, unless the DisableOnInit property has been set to TRUE. Compare this property with the similar FieldsEnabled property for Viewers and Browsers. When the framework initializes a Browser or Viewer, its ObjectEnabled property is set to TRUE. If it has no enabled columns or fields, however, or if it is in View mode rather than Modify mode, then its FieldsEnabled property is FALSE. Thus the FieldsEnabled property toggles back and forth between TRUE and FALSE depending on the state of the window and the toolbar buttons, while the ObjectEnabled property is normally set to TRUE once and left that way.

FieldsEnabled

This LOGICAL property is TRUE when the fields or columns of a Viewer or Browser are enabled for data entry, and FALSE otherwise.

5.1.3 Field and widget lists

Application code often needs to identify and get the handles of the fields and other objects inside another object. This section describes some properties that give you that information.

DisplayedFields and FieldHandles

These CHARACTER properties are defined for Datavis objects, that is, SmartObjects that visualize data from a query (normally from an SDO). For a Viewer, the **DisplayedFields** property returns a comma-separated list of all the fields in the Viewer that come from the SDO, and the **FieldHandles** property returns the WIDGET-HANDLES of those fields. The fields may be any type that can display data, including FILL-IN, EDITOR, RADIO-SET, and so on.

The **FieldHandles** property returns a comma-separated list as well, with each handle converted to a text string. To retrieve the native handle from the list, you need to apply the WIDGET-HANDLE 4GL function to the appropriate entry in the list. Both lists are always in the same order, so that the standard way to get the handle of a field is to look it up in the **DisplayedFields** list and then locate the corresponding entry in the **FieldHandles** list. For example:

```
DEFINE VARIABLE cFields AS CHARACTER NO-UNDO.
DEFINE VARIABLE cHandles AS CHARACTER NO-UNDO.
DEFINE VARIABLE iField AS INTEGER NO-UNDO.
DEFINE VARIABLE hField AS HANDLE NO-UNDO.

{get DisplayedFields cFields}.
{get FieldHandles cHandles}.
iField = LOOKUP('Address', cFields).
IF iField NE 0 THEN
DO:
    hField = WIDGET-HANDLE(ENTRY(iField, cHandles)).
    MESSAGE hField:SCREEN-VALUE.
END.
```

This bit of code yields the display shown in [Figure 5–1](#) at run time for the first Customer record.

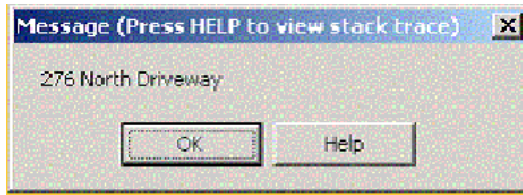


Figure 5–1: Customer record message box

In the case of a SmartDataField in a Viewer, which is a SmartObject procedure in its own right representing a single field value, the entry in FieldHandles is its procedure handle. The corresponding entry in the DisplayedFields list always points to the field name in the DataSource.

For example, code that manipulates the handle must always check the TYPE attribute:

```
DEFINE VARIABLE cFields AS CHARACTER NO-UNDO.
DEFINE VARIABLE cHandles AS CHARACTER NO-UNDO.
DEFINE VARIABLE iField AS INTEGER NO-UNDO.
DEFINE VARIABLE hField AS HANDLE NO-UNDO.

{get DisplayedFields cFields}.
{get FieldHandles cHandles}.
iField = LOOKUP('SalesRep', cFields).
IF iField NE 0 THEN
DO:
    hField = WIDGET-HANDLE(ENTRY(iField, cHandles)).
    IF hField:TYPE='Procedure' THEN
        MESSAGE DYNAMIC-FUNCTION('getDataValue' IN hField).
    ELSE MESSAGE hField:SCREEN-VALUE.
END.
```

The SmartDataField supports various internal functions that provide parallels to the properties of ordinary field-level widgets. For more information, refer to the [Progress Dynamics ADM2 API Reference](#).

For a Browser, the DisplayedFields and FieldHandles properties return the list of columns in the browse itself and the handles to the browse cells for those columns.

EnabledFields and EnabledHandles

These two CHARACTER properties correspond to the DisplayedFields and FieldHandles properties, except that they return only fields or browse cells that are enabled for input.

AllFieldNames and AllFieldHandles

These CHARACTER properties, available for all visual objects (datavis class), return a list of all widgets in the SmartObject's frame. This list includes the data fields that the DisplayedFields property returns as well as all other objects in the frame that are not derived from an SDO. For example, there might be non-SDO fields, along with field labels, rectangles, and buttons. Each of these is a separate widget with its own handle. For the SmartDataField, the procedure handle appears in the handle list.

These properties can be useful if you need to manipulate objects in a Viewer that are outside the standard field list, such as buttons. You can also use it for visual containers that have no SDO fields, such as a SmartWindow.

Currently, note that for Browsers, these properties return the names and handles of the objects in the frame, not the columns in the Browser. Thus just a single name (brtable) comes back for the browse widget itself in the Names list, and its widget handle in the Handles list, along with the names and handles of any other objects that might be in the SmartBrowser's frame along with the browse control. If you need the column list, you must use the **DisplayedFields** and **FieldHandles** properties.

5.1.4 Useful object handles

The properties described in this section return useful object handles to you.

ContainerHandle

This HANDLE property, available for all SmartObjects, is only meaningful for objects that have a window or frame. For a frame-based object, such as a Browser or Viewer, it returns the frame handle. For a window-based object, such as a SmartWindow, it returns the window handle. Use this handle when you need to query or manipulate the attributes of the container widget itself.

BrowseHandle

This HANDLE property returns the browser widget handle for a SmartBrowser. Use this handle when you need to query or manipulate the attributes of the browse itself.

BufferHandles and QueryHandle

These two properties return handles from the database query that is used to populate an SDO. The **BufferHandles** CHARACTER property is a comma-separated list of the handles of the database record buffers the framework uses to populate the SDO temp-table.

The **QueryHandle** HANDLE property is the handle of the database query. Note that these handles are the database object handles, not SDO temp-table or buffer handles. They are valid only on the server side of a divided SDO, and should be used only in cases where your code needs to reference the database data as the framework loads it into the temp-table.

QueryRowObject

This HANDLE property returns the buffer handle to the SDO's temp-table buffer (the RowObject buffer in other words) that holds the data values for the currently selected row in a Browser. This can be useful when you need to query those values within the ROW-DISPLAY trigger for the Browser, for example. The SCREEN-VALUES of the individual cells cannot be queried within this 4GL event, so in order to write logic that reacts to a cell's value, by changing the background color or some other attribute of the cell or the row, you need to be able to refer back to the underlying data buffer and its fields. For more information, see [Chapter 1, "Writing Super Procedures for Progress Dynamics Objects"](#).

Note that the WIDGET-HANDLE is implemented to hide this complexity.

As an example, look at this code from the widgetValue function that is part of the proposed client-side API for retrieving and setting values in dynamic objects from custom super procedures. The variable hField holds the handle to a field from the FieldHandles list:

```
IF CAN-QUERY(hField, "FILE-NAME") THEN
    /* This is a SmartDataField. Return its DataValue property.
       Note that this is the underlying "key" value that is meaningful
       to the code, not the "DisplayValue" shown to the user. */
    {get DataValue cValue hField}.
ELSE IF CAN-QUERY(hField, "SCREEN-VALUE") THEN
    cValue = hField:SCREEN-VALUE.
ELSE DO:
    {get QueryRowObject hBuffer} NO-ERROR.
    IF VALID-HANDLE(hBuffer) THEN
        ASSIGN hField = hBuffer:BUFFER-FIELD(cField)
        cValue = STRING(hField:BUFFER-VALUE) NO-ERROR.
END. /* END ELSE DO IF NOT CAN-QUERY SCREEN-VALUE (Browse) */
```

5.1.5 Functions that return field values and attributes

The Query class that supports SDOs provides a large number of functions that you can use to retrieve data values and the properties of those fields.

column<attribute> functions

There is a whole set of functions that return individual 4GL attributes from fields in a data object (one for which the QueryObject property is TRUE, basically SDOs or SBOs).

These functions are located in an adjunct SDO super procedure called `dataextcols.p`. Both the query class and the data class have enough supporting functions that the code to support them causes compilation problems in some versions of Progress. So the get- and set- functions for properties are in separate super procedures called `queryext.p` and `dataext.p` respectively. In addition, the `dataextcols.p` procedure supports all these special column attribute functions.

These are all run along with the standard super procedures `query.p` and `data.p` and made part of the super procedure stack.

These functions return values in the appropriate data type for the attribute. All the functions have names of *column* plus the attribute name. The prefix, `column`, is not entirely appropriate here, because these return field values from the current row in the SDO buffer, not browse column values.

They take a single input parameter, which is the name of the field. This can be either:

- A simple SDO field name.
- A field name qualified by `RowObject`.
- A field name qualified by the database table it is derived from.

Column attribute functions include:

- **columnColumnLabel** — Returns the Column-Label of the field. Note the word **column** **does** appear twice in the function name.
- **columnDataType** — Returns the data type of the underlying database field represented by this column.
- **columnDBCColumn** — Returns the name of the column in the database, in case it has been renamed in the SDO.
- **columnDBName** — Returns the name of the database that the field is derived from.
- **columnHandle** — Returns the handle of the field in the `RowObject` buffer.
- **columnHelp** — Returns the value of the field's Help text.
- **columnInitial** — Returns the field's initial value.
- **columnLabel** — Returns the field's label.
- **columnModified** — Returns a LOGICAL flag indicating whether the field has been modified since it was read into the `RowObject` table.
- **columnPrivateData** — Returns the Progress Private-Data attribute of a specified column.
- **columnTable** — Returns the name of the database table the column is derived from. It is qualified according to how the SDO is defined.
- **columnWidth** — The width is returned in characters.

There are also equivalent **assignColumn<attribute>** functions for most of these, which take the field name and the attribute value as input parameters.

There are also a few more specialized column functions that are useful, as described in the sections below.

columnQuerySelection

This CHARACTER function, supported for data objects, takes a column name as input and returns a CHR(1)-delimited string with all operators and values that have been added to the query for this field using the **assignQuerySelection** method (described later in this chapter). For example, if the query contains `custnum > 5` and `custnum < 9`, this function returns `>|5|<|9` (where CHR(1) is shown as |). Use this function to do analysis on the current query or to display information to the user.

columnValue

This CHARACTER function, supported for data objects, takes the name of the field as input and returns its value, in character form, but without any formatting characters added. This is equivalent to applying the STRING function to the BUFFER-VALUE of the field.

columnStringValue

This CHARACTER function, supported for data objects, takes the name of the field as input and returns its STRING-VALUE, which is the fully formatted value as it would appear on the screen (so it is the buffer equivalent to the SCREEN-VALUE of a displayed field or browse column).

colValues

This CHARACTER function, supported for data objects, takes as input a comma-separated list of field names in an SDO, and returns a CHR(1)-delimited list of the formatted values (STRING-VALUES, in effect) for those columns for the current row, preceded by a string called the RowIdent that holds the database RowID of the record the columns are derived from. If the SDO involves a join, then this first RowIdent entry in the list is itself a comma-separated list of the RowIDs for the records in the join. This is the standard function used internally by Viewers in particular to retrieve the values to display as the SCREEN-VALUES of the fields in the Viewer.

Note that because the RowIdent is always the first entry in the CHR(1)-delimited list, your code must always look for the value for a column that was in the *n*th position in the input list, in the *n+1*th position in the return value.

This function can be useful for retrieving multiple values at a time, but the **columnValue** and **columnStringValue** functions were designed for application use as a more convenient way to retrieve individual values at a time.

colStringValues

This CHARACTER function provides an alternative to colValues as a way to retrieve multiple values in a single call, and with flexible formatting. It does not return the RowIdent of the database records as colValues does. The function takes three INPUT parameters:

- **pcColumnList** — This is a comma-delimited list of RowObject column names to return values for.
- **pcFormatOption** — This parameter allows you to specify the format of returned values. It can be:
 - **Blank or ?** — No formatting is done; unformatted buffer values are returned.
 - **Formatted** — Returned values are formatted according to the columnFormat with right-justified numeric values
 - **TrimNumeric** — Returned values are formatted according to the columnFormat with left-justified numeric values
- **pcDelimiter** — This optional parameter specifies a delimiter between values; the default delimiter is CHR(1).

5.1.6 Other useful data object functions

The following additional functions are supported for SDOs, including functions that return index information and functions that retrieve logic procedures:

indexInformation

This CHARACTER function is supported for SDOs. It returns index information for all the buffers in an SDO's query. The indexes are separated by CHR(1). Field information is either qualified with the database and table name or CHR(2) is used as table separator. The INPUT parameters for this function are:

- **pcQuery** — This parameter indicates what information is needed:
 - **All** — All indexed fields
 - **Standard** or **'** — All indexed fields excluding word indexes
 - **Word** — Word Indexes
 - **Unique** — Unique indexes

- **NonUnique** — Non-unique indexes
- **Primary** — Primary index
- **Info** — (meaningless if **pcIndexInfo** has a value ?)
- **plUseTableSep** — This LOGICAL parameter indicates whether the information should be returned using a special table separator. If this parameter is TRUE then the data is returned with CHR(1) as the separator between indexes and CHR(2) as the separator between tables. If the following pcIndexinfo parameter is the unknown value (?) then the returned field names are qualified; otherwise they remain as they are in pcIndexInfo.
- **pcIndexInfo** — This CHARACTER parameter is either a signal to use the SDO query or a string of previously retrieved information. If the parameter has the unknown value (?) then the function uses the query as the basis for returning information. In this case, if the **plUseTableSep** parameter is TRUE then the field is returned with CHR(1) and CHR(2) qualifiers. Otherwise this parameter must be the index information in the **exact same format** as was returned from this same function in a previous call with the syntax **indexInformation** ('info', TRUE, ?). in addition, there is a parallel **IndexInformation property** that allows the function to be used with no database connection, or without forcing a call from client to server to retrieve information. The property value is set in the client SDO when it is initialized, so its value is always available on the client. You can use the value of the IndexInformation property as the INPUT pcIndexInfo parameter value to this function, in which case the function acts as a filtering and formatting mechanism to return to you just the information you need, in an appropriate format.

dataLogicProcedure, dataLogicObject

Use these SDO functions to retrieve the name of the SDO's logic procedure, or its procedure handle, respectively.

Note on frame properties

There are several frame properties that are not supported in the static property sheet. Typically these properties have values that should not be changed for a Progress Dynamics style application and will therefore be deprecated in the future. These attributes include: NO-BOX, TITLE COLORS, NO-UNDERLINE, Virtual Height and Width, Three-D, SIDE-LABELS. These attributes may also unavailable for other objects types when they should be (for example, NO-BOX should be available in the customization of editors and isn't.)

5.2 Starting Progress Dynamics application windows

When you create a static SmartWindow in the AppBuilder and drop objects onto it, the AppBuilder generates a procedure called **adm-create-objects** to create all of those objects at runtime. In Progress Dynamics, most windows are dynamic objects, so one dynamic window procedure, `rydyncontw.w`, has an empty version of `adm-create-objects` that acts like a placeholder. The procedure calls to create the window, and its contents are executed at runtime rather than being pre-generated into a source code file. The sequence of steps that occurs and the procedures that are run remain the same as for static windows. The `adm-create-objects` procedure retains the older naming style of `adm-` plus a procedure name from an older version of the ADM. The reason for this is that in static windows, we need to allow for the AppBuilder-generated code, which the developer should not edit directly, plus a possible local override procedure that can do additional work when objects are created. Because these are both in the same source procedure in a static environment, they need to have different names.

As a result, the actual ADM event that is run in a container and published in any child containers is **createObjects**. The standard code for `createObjects`, found in the super procedure `containr.p`, runs `adm-create-objects`, which executes the AppBuilder-generated code for a static window. If the developer wants to write custom code for this stage of the application, this goes into a local version of the `createObjects` procedure:

Procedure `createObjects`:

Parameters: <none>

Candidate for: **localization**See below for information on creating a local version of `createObjects`

Here is an excerpt from an AppBuilder-generated **adm-create-objects** procedure. It can serve as a model for running some of the same support procedures in your own code:

(1 of 2)

PROCEDURE adm-create-objects:

```

/*-----
Purpose:
Create handles for all SmartObjects used in this procedure.
After SmartObjects are initialized, then SmartLinks are added.
Parameters: <none>
-----*/

DEFINE VARIABLE currentPage AS INTEGER NO-UNDO.
ASSIGN currentPage = getCurrentPage().

CASE currentPage:
  WHEN 0 THEN DO:
    RUN constructObject (
      INPUT 'adm2/pnavlbl.w':U ,
      INPUT FRAME fMain:HANDLE ,
      INPUT
'EdgePixels_2_PanelType_Nav-Label_HideOnInit_no_DisableOnInit_no_ObjectLayout_
t_':U ,
      OUTPUT h_pnavlbl ).
    RUN repositionObject IN h_pnavlbl ( 1.71 , 22.00 ) NO-ERROR.
    RUN resizeObject IN h_pnavlbl ( 1.76 , 34.00 ) NO-ERROR.
  ...
  RUN constructObject (
    INPUT 'dcust.w_DB-AWARE':U ,
    INPUT FRAME fMain:HANDLE ,
    INPUT
'AppService_ASUsePrompt_ASInfo_ForeignFields_RowsToBatch_2_CheckCurrentC
hanged_yes_RebuildOnRepos_no_ServerOperatingMode_NONE_DestroyStateless_no_Di
sconnectAppServer_no':U ,
    OUTPUT h_dcust ).
    RUN repositionObject IN h_dcust ( 1.95 , 6.00 ) NO-ERROR.
    /* Size in AB: ( 1.86 , 10.80 ) */

    RUN constructObject (
      INPUT 'adm2/pupdsav.w':U ,
      INPUT FRAME fMain:HANDLE ,
      INPUT
'AddFunction_One-Record_EdgePixels_2_PanelType_Save_HideOnInit_no_DisableOnI
nit_no_ObjectLayout_':U ,
      OUTPUT h_pupdsav ).
    RUN repositionObject IN h_pupdsav ( 4.33 , 20.00 ) NO-ERROR.
    RUN resizeObject IN h_pupdsav ( 1.76 , 56.00 ) NO-ERROR.

    RUN constructObject (
      INPUT 'vcust.w':U ,
      INPUT FRAME fMain:HANDLE ,
      INPUT 'HideOnInit_no_DisableOnInit_no_ObjectLayout_':U ,

```

```

OUTPUT h_vcust ).
  RUN repositionObject IN h_vcust ( 7.91 , 4.00 ) NO-ERROR.
  /* Size in AB: ( 6.86 , 66.00 ) */

  /* Links to SmartDataObject h_dcust. */
  RUN addLink ( h_pnavlbl , 'Navigation':U , h_dcust ).

  /* Links to SmartDataViewer h_vcust. */
  RUN addLink ( h_dcust , 'Data':U , h_vcust ).
  RUN addLink ( h_vcust , 'Update':U , h_dcust ).
  RUN addLink ( h_pupdsav , 'TableIO':U , h_vcust ).

/* Adjust the tab order of the smart objects. */
  RUN adjustTabOrder ( h_pupdsav ,
    h_pnavlbl , 'AFTER':U ).
  RUN adjustTabOrder ( h_vcust ,
    h_pupdsav , 'AFTER':U ).
  END. /* Page 0 */
END CASE.
END PROCEDURE.

```

The static adm-create-objects procedure, or the createObjects code for dynamic windows, does several things. First, for each SmartObject in the Container, it runs constructObject, which takes the SmartObject procedure name, its parent Frame handle, and its list of Instance Property settings as INPUT parameters, and returns the procedure handle of the new SmartObject as an OUTPUT parameter. The constructObject procedure runs the SmartObject as a persistent procedure, parents its default Frame (if any) to the Frame handle passed in, and runs the set<property> function for each Instance Property name/value pair passed in. Instance Properties are SmartObject properties for which initial values can be defined for a particular instance of the Object as used in some Container. These properties can be set in the Instance Property Dialog of the SmartObject as the Container is assembled, and they are initialized here:

Procedure constructObject:

Parameters:

```

  (INPUT pcProcName /* CHARACTER -- SmartObject name */,
   INPUT phParent   /* HANDLE -- parent Frame handle */,
   INPUT pcPropList /* CHARACTER -- Instance Properties */,
   OUTPUT phObject  /* HANDLE -- new procedure handle */).

```

Candidate for: **calling**

You can call constructObject from your own code to create additional objects in a window at runtime based on application-specific or user-specific factors.

The procedure handle of the new SmartObject instance is returned to adm-create-objects, which uses it in several other procedure calls.

For any SmartObject with a visualization, `repositionObject` is run to position it at runtime. Also, for any Object that is resizable, `resizeObject` is run to size the object appropriately. The presence of the `resizeObject` procedure in the Object (or its super procedures) determines whether it should be made resizable. In Dynamics, of course, object positions and sizes are not fixed at design time. The layout manager determines the sizes and relative positions of objects based on the overall window size and on which objects are resizable. But you can use these procedures where needed in your code to adjust sizes and positions:

Procedure `repositionObject`:

Parameters:

(INPUT `pdRow` /* DECIMAL - Row number */,
INPUT `pdCol` /* DECIMAL - Column number */).

Candidate for: **calling**

You could run `repositionObject` from your code to adjust default object positions or to position objects you created outside the window definition stored in the repository:

Procedure `resizeObject`:

Parameters:

(INPUT `pd_height` /* DECIMAL - height in Rows*/,
INPUT `pd_width` /* DECIMAL - width in Characters */).

Candidate for: **calling**

You could resize an object based on application requirements, or to make adjustments outside the standard layout manager support.

Once all the SmartObjects have been created, any SmartLinks between those objects are defined. Use the `addLink` procedure to do this. This is described in the [“Managing links in Progress Dynamics applications”](#) section of this chapter.

The AppBuilder also generates calls to **`adjustTabOrder`** to assure that the tabbing order between SmartObjects is either be left-to-right / top-to-bottom (the default), or as reset by the developer in the AppBuilder:

Procedure `adjustTabOrder`:

Parameters:

(INPUT `phObject` /* HANDLE - Object to re-order */,
INPUT `phAnchor` /* HANDLE - Object to re-order relative to */,
INPUT `pcPosition` /* CHARACTER - move “BEFORE” or “AFTER”*/).

Candidate for: **calling**

Your code can adjust tab order based on user preferences or application specific requirements.

There are two additional properties that are used in Dynamics to size objects proportionally in dynamic windows, **resizeHorizontal** and **resizeVertical**. The LOGICAL properties indicate whether a visual object can be resized in one dimension or the other or both when the window is initially sized, or when it is resized by a user. The properties are defined for all visual objects and have the default values shown in [Table 5-1](#).

Table 5-1: Default values of resize properties for visual objects

Object	resizeHorizontal properties	resizeVertical properties
Viewer	FALSE	FALSE
Browser	TRUE	TRUE
Window	TRUE	TRUE
Toolbar	TRUE	FALSE

Note that you can override these default values. You can set these properties for individual objects to change the default resizing, for example if you want a Browser to be a fixed size in one or both dimensions.

You can set these properties in the Container Builder in the Instance section of the main window.

Now when you run the Customer browse window, the Browser is a fixed height as you can see in [Figure 5-2](#).

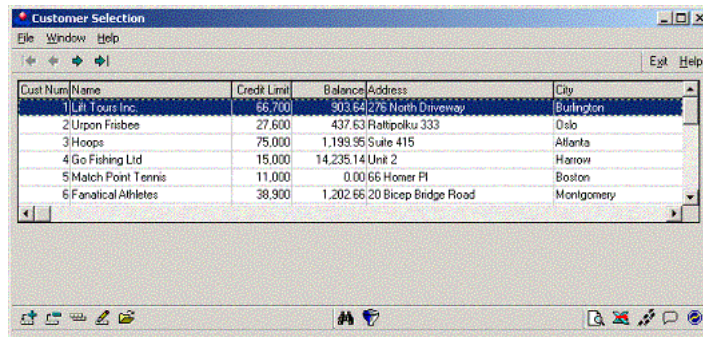


Figure 5-2: Customer selection window

The createObjects and adm-create-objects procedures are part of a sequence that occurs when a Container is run. It's important to know the steps in this sequence and where the appropriate places are to put custom application code.

When you run SmartObject code from `constructObject`, any code in its Main Block is executed first. Custom code that should be executed as soon as the Object is run, before anything else happens, can be placed in the Main Block. Remember that at this time the object has no links to other objects, so you cannot refer to its Data-Source or other such connections; its Instance Property values also are not yet set, so you cannot refer to them. If you try to set them, the values set by the Container override your settings. If you create static Objects and need code to be executed immediately when the Object is run, you can put it in the Main Block (this is true for any persistent procedure). If you're using dynamic Objects, then you'll have to write a super procedure where you override standard procedures or respond to events.

For the remainder of this chapter, any references to local application code should be taken to mean either code written in a static SmartObject procedure or (preferably in most cases) code written into a custom super procedure for a dynamic Object.

Custom code for a window that needs to reference its contained SmartObjects as soon as they are created should go into a local `createObjects` procedure. If you look at the same `adm-create-objects` procedure above, note that it references the `CurrentPage` property, and then executes a block of a CASE statement depending on the current page number. When the window is first run, its `CurrentPage` is zero. As each additional page (if any) is first referenced, `createObjects` and `adm-create-objects` are run again to create the objects on that page. So if you define a local `createObjects` procedure, remember that it runs multiple times if the window has more than just the default page zero, check the `ObjectCreated` property. It will be false the first time and true on each subsequent call. You can also check the `CurrentPage` property in the same way that `adm-create-objects` does. Each CASE should contain a RUN SUPER statement to invoke the code in `adm-create-objects` for that page, with the custom code added either before or after that statement, as appropriate.

Code that needs to do something immediately **before** the objects on a page are created can be placed in `createObjects` before the RUN SUPER statement for that page. Code that needs to be executed **after** the objects are created (which seems more likely, since of course you cannot access the objects on a page before they are run) must be written after the RUN SUPER statement for that CASE. For example, a `createObjects` procedure can add additional links between objects that have just been created (based on dynamic application requirements); or it can set additional property values that are not Instance Properties or that need to be set conditionally based on application requirements; it might reposition or reorder objects; or other such things. At this point in the window's life, all the objects on the current page have been created, positioned and resized; their frame handles have been parented to the window's frame; their Instance Property values have been set; and their links and tab order have been established. But they not have been initialized; that is, the `initializeObject` event procedure has not been run, so visual objects have not been viewed or enabled; queries have not been opened, and so on.

5.2.1 Initializing SmartObjects

You start a SmartObject application in two major steps. Creating the object instance is the first of those steps. This section describes the second step.

Once createObjects is done, the window publishes **initializeObject**, which cascades down as an event to all its children (and their children if any, recursively) and then initializes the window itself:

Procedure initializeObject:

Parameters: <none>

Candidate for: **localization**

Application code often must extend what happens at startup by localizing the initializeObject procedure. If your code is creating objects the framework doesn't manage for you, then the code will also need to *run* initializeObject in them.

Each type of SmartObject does its own initialization, and then passes control up the stack of super procedures with a RUN SUPER. Among the major events that happen along the way are described in the following paragraphs.

The initialization runs enableObject unless it is instructed not to, that is, unless the DisableOnInit property is set to TRUE, which is not its default. It then runs viewObject unless instructed not to by the setting of the HideOnInit property, whose default value is also FALSE.

enableObject enables any handles in the SmartObject that are not associated with fields from an SDO (these could be extra developer-defined fill-in fields and so on). It sets the ObjectEnabled property to YES. It also runs the procedure enable_UI for AppBuilder-generated objects; this is not something you would write or change yourself. Its counterpart for disabling an object is the event procedure disableObject:

Procedure enableObject/disableObject:

Parameters: <none>

Candidate for: **localization** and **calling**

If there is extra work to do when the SmartObject is enabled or disabled, you can do this by creating a local version of the procedure.

If code needs to enable an Object explicitly at the appropriate time, your code may run enableObject or disableObject directly. For example, enableObject must be run for an Object by application code if its DisableOnInit Property was initially set to YES.

The **viewObject** event procedure is actually run for every type of SmartObject, including non-visual ones, since the concept of being viewed (which sets the **ObjectHidden** property to FALSE) is used essentially as a synonym for active for SmartObjects. For this reason it is supported for all SmartObjects, along with its counterpart, the procedure **hideObject**. The viewObject procedure actually views the Object's default frame if it has one, sets the ObjectHidden property to NO, and publishes the linkState event with a parameter of active to signal to other objects such as SmartPanels and Toolbars that they are now active (so buttons should be enabled which were disabled when the target object was hidden by hideObject, and linkState was published with inactive):

Procedure viewObject/hideObject:

Parameters: <none>

Candidate for **localization** and **calling**

If there is extra work to do when the SmartObject is viewed or hidden, you could write a local override of the procedure.

If code needs to view or hide an Object explicitly at the appropriate time, then your code could run the procedure directly. For example, viewObject must be run for an Object by application code if its HideOnInit Property was initially set to YES.

The initialization sequence then runs **displayObjects** to display any non-database fields which may be defined. The displayObjects event procedure displays the values of any fields or other basic objects in a visual SmartObject that are not associated with SDO fields. It runs during initialization:

Procedure displayObjects:

Parameters: <none>

Candidate for **calling**

If the values of non-data fields need be refreshed other than when a record is displayed in the Object, then your code might call displayObjects to do this.

The initialization sequence then runs **enableFields** if appropriate (if there is a TableIO-Source in Save mode, which allows updates to be entered into the Viewer's fields at any time). Then it runs dataAvailable to see if there is a row in the Data-Source waiting to be displayed.

The enableFields event procedure enables those Viewer fields or Browser columns that are associated with SDO data fields (using the EnabledHandles property), setting their SENSITIVE attribute to yes. (Editors have their READ-ONLY attribute set to no; they are always left SENSITIVE to allow them to be scrolled.). The FieldsEnabled property is set to YES, and the enableFields event is published to send it on to any contained SmartObjects.

In the case of a SmartDataField, the fields could include widgets not associated with the data field. The enableObjFields property holds a comma separated list of these widgets which will be enabled by default. Conversely, enableObjFieldsToDisable contains a comma-separated list of similar field names which will be disabled by default.

The counterpart to enableFields for disabling data fields is the event procedure disableFields:

Procedure enableFields/disableFields:

Parameters: <none>

Candidate for: **localization** and **calling**

If additional code is needed for an object each time its fields are enabled or disabled, you could create a local version of the procedure.

If data-related fields need to be enabled or disabled when it doesn't happen by default, then code could call the enable or disable procedure directly.

The **dataAvailable** event procedure is run each time a new (different) row is positioned to in the SDO query, and when an Update completes. In the case of initializeObject it is run to catch the case where the Data-Source may have been initialized first and has already published the event before the Viewer was ready to receive it. dataAvailable has two basically different jobs, depending on the SmartObject type: For visual SmartObjects it causes the field values for the newly selected row to be displayed. For data objects such as SDOs it is passed on to a child SDO in a parent-child (master-detail) relationship when the parent SDO moves to a different row. In this case it causes the child SDO's query to be re-prepared with the foreign key values from the parent row, and the query to be re-opened:

Procedure dataAvailable:

Parameters: pcRelative AS CHARACTER

Candidate for: **localization**

If additional code is needed for an object each time a different row is Selected, this could be done in a local version of dataAvailable.

The **pcRelative** parameter to dataAvailable can have these values:

- **Different** — This value indicates that a different row in the SDO query has been positioned to. Viewers and Browsers will display that row, and dependent SDOs will re-open their queries based on the key values from the new row.
- **“?”** — The unknown value indicates that it is not know whether or not there is a new row (when called from initializeObject, for instance).
- **SAME** — This value signals that the current row has been updated. A Viewer or Browser will display the row's new values, but a dependent Query Object will not bother to re-open its query just because field values have changed in its parent.

- **RESET** — This value causes the procedure to check whether the Foreign Values for the SDO's ForeignFields key values have changed since the last call do dataAvailable. If so, then a dependent SDO re-opens its query accordingly. If not, it does nothing. This parameter value in effect combines the behavior of the SAME and DIFFERENT values, checking intelligently to see which is the case. It is therefore the most efficient value to pass when it is not always known if a new record with a new set of key values is present.
- **FIRST, NEXT, PREV, LAST** — These values signal repositions within a SmartDataBrowser to coordinate with an associated SmartPanel.

The **openQuery** function opens the database query and populates the SDO's RowObject temp-table with the first batch of rows from the database. First it runs the **closeQuery** function if the query has previously been opened. Then it runs the **prepareQuery** function to re-prepare the query if the **QueryString** property has been set (described in a later section). If the SDO is divided between client and AppServer, the client side invokes **openQuery** on the server to open the actual database query.

It then runs the **fetchFirst** procedure to populate the temp-table and position to the first row. **fetchFirst** runs the **sendRows** procedure, which has both a client and server version, to request the first batch of rows. This in turn runs the **transferDBRow** procedure once for each row in the batch. **transferDBRow** basically does a BUFFER-COPY of the database record or records used in the query into the server-side RowObject Temp-Table. It in turn runs the AppBuilder-generated procedure **Data.Calculate** (if it is defined), which does the calculations for any calculated fields in the SDO.

Finally, **fetchFirst** on the client side sets the **QueryPosition** property to **FIRST** and then publishes the **dataAvailable** event with an argument of **FIRST** to cause client-side objects to display the first row:

Function openQuery:

Parameters: <none>

Candidate for: **calling**

If the application has prepared a new query definition with a different where clause, or if the application wants to refresh the current query with the latest data, then code can call **openQuery** directly.

The **closeQuery** function closes both the (client-side) RowObject temp-table query, emptying the temp-table, and also the (server-side) database query. It publishes 'dataAvailable' to cause any Data-Targets to erase the current record display (or to close a dependent query if there is one). Your code does not normally call **closeQuery** directly.

The **prepareQuery** function prepares a new query if it has been modified by the functions used to change the where clause. It is intended to be run only internally. Developers should use the functions described in the section on customizing queries, which in turn invoke this function at the proper time.

The **fetchFirst** procedure, along with **fetchNext**, **fetchPrev**, and **fetchLast**, repositions the RowObject query to the corresponding row. These procedures also request a new batch of rows from the database if needed. They reset the QueryPosition property to the appropriate value ('FirstRecord', 'LastRecord', 'OnlyRecord', or 'NotFirstOrLast'), and publish dataAvailable to alert other objects that there is a different row to display or open a dependent query with. As described, fetchFirst is run from openQuery. Otherwise the four procedures are normally published from a Navigation Toolbar band when the corresponding navigation button is pressed:

Procedure fetchFirst/fetchNext/fetchPrev/fetchLast:

Parameters: <none>

Candidate for: **calling**

If repositioning other than from a Toolbar is needed, then code could possibly run these procedures directly

The **sendRows** procedure (along with its two sub-procedures clientSendRows and serverSendRows) transfers batches of rows from server to client. Your code should not normally run this directly.

The dataAvailable procedure in a data display (Datavis) object such as a Viewer or Browser runs displayFields, which takes as its input parameter a list of values in the form returned by the colValues function, and moves them into the screen values of the Viewer or Browser. The displayFields procedure also runs displayObjects to display any non-data-related fields:

Procedure displayFields:

Parameters: pcValueList (CHR(1)-delimited list of values preceded by the RowIdent

Candidate for: **localization**

If application code needs to adjust values before they are displayed or do something else each time a row is displayed, you can do this in a local displayFields.

5.2.2 Other methods related to startup and shutdown

The following event procedures are also related to starting up and shutting down SDOs.

destroyObject

This is the standard event procedure that runs to shut down an Object. It deletes links properly and performs other cleanup tasks before deleting its own persistent procedure:

Procedure destroyObject:

Parameters: <none>

Candidate for: **localization**

If application code needs to do other work just before an Object is destroyed, it can be done *before* the RUN SUPER statement in a local version of destroyObject.

exitObject

For an application window to shut down properly, its Objects should be deleted from the inside out. That is, the contained Objects must clean up and delete themselves before the window itself is deleted. An event such as pressing a Done or Exit button inside a container can initiate this by publishing the exitObject event. The convention is that the event passes the exit request to its Container-Source. The container that can initiate the exit defines a local version and does not call the standard one.

That local exitObject is built into the SmartWindow support code:

Procedure exitObject:

Parameters: <none>

Candidate for: **calling** and possibly **localization**

Any button or other object inside a container that wants to close the container should run exitObject in its immediate container (a Viewer for example).

If application code needs to do other work just before a window is destroyed, it can be done *before* the RUN SUPER statement in a local version of exitObject.

launchFolderWindow

This procedure runs routinely from a dynamic Browser when the user double-clicks on a row or selects the Edit or Modify button. It retrieves the value of the FolderWindowToLaunch property for the Browser, which is part of the Browser definition, and invokes the launchContainer procedure in the Session Manager. For more information on launchContainer and examples of its use, see [Chapter 5, “Using ADM2 Properties and Methods In Progress Dynamics.”](#)

Because `launchFolderWindow` is specifically intended to be invoked inside a Browser, you normally use the `launchContainer` procedure itself when you want to invoke containers from your application code.

5.2.3 Other initialization properties

This section describes the following properties that are related to starting objects.

HideOnInit

This LOGICAL property is defined for all visual Objects and has a default value of FALSE. If your initialization code sets it to TRUE for an Object before the Object is initialized, then that Object is be initially hidden when it is created. This is an example of a property you can set from a local version of `createObjects`, after the objects on a page have been created but before their `initializeObject` code is run.

If you set **HideOnInit** for an Object then you must run **viewObject** when you want it to appear.

DisableOnInit

This LOGICAL property is defined for all visual Objects and has a default value of FALSE. If your initialization code sets it to TRUE for an Object before the Object is initialized, then that Object is initially disabled when it is created. This is another example of a property that you can set from a local version of `createObjects`, after the objects on a page have been created but before their `initializeObject` code is run.

If you set **DisableOnInit** for an Object then you must run **enableObject** when you want it to appear.

ObjectInitialized

This LOGICAL property is set to TRUE when the Object's **initializeObject** procedure has run. You should not set it yourself, but you can check its value to determine whether an Object has already been initialized, to avoid running custom initialization code a second time, for example.

OpenOnInit

This LOGICAL property is defined for data objects whose **QueryObject** property is TRUE, and has a default value of TRUE. If you set the value to FALSE before the Object is initialized, then its query does not open and its temp-table is not populated. If you want the user to enter filter data to reduce the size of the dataset before any rows are returned, you can set the `OpenOnInit` property to FALSE. In this case the application does not needlessly open the query, populate the temp-table, and return the table to the client simply to display the first rows in the table, which the user may not want to see. This is another property that you can set in a local `createObjects`.

FolderWindowToLaunch

This CHARACTER property is defined for dynamic Browsers. When you define a dynamic Browser you can use **FolderWindowToLaunch** to specify the maintenance window to launch when the user selects a row in the Browser. Note that the Folder Window To Launch property does not actually have to be a window with a folder in it. It **does** need to be a window that expects to receive a key value from the calling window, so that the child window can display the data for that row.

Figure 5–3 shows the AppBuilder property sheet for the dynamic Browser, where you can set this property.

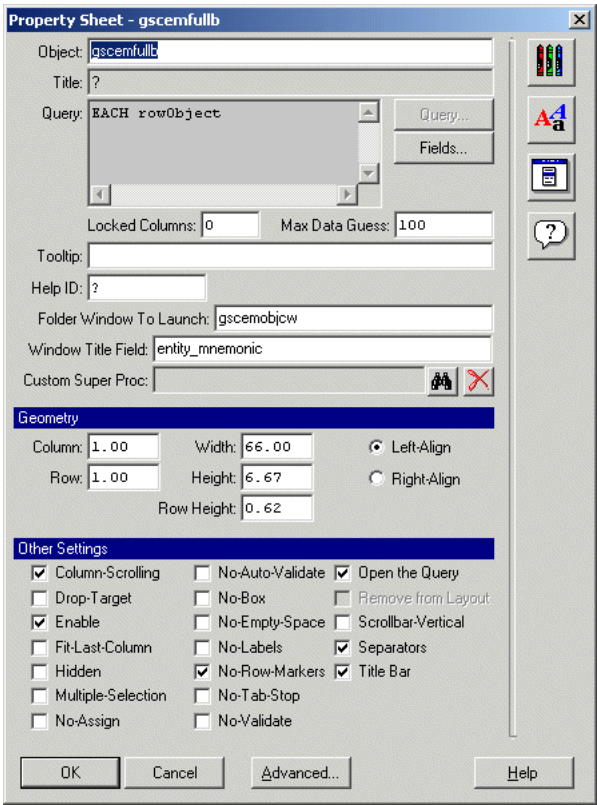


Figure 5–3: Property Sheet Master

Your code could also programmatically set this property value at runtime if the child window to launch depends on some application particulars.

5.3 Managing links in Progress Dynamics applications

This section describes the following properties and functions that support the ADM SmartLink mechanism:

- [SmartObject SupportedLinks and the addLink procedure](#)
- [Defining custom SmartLinks and events](#)
- [Link support methods](#)

5.3.1 SmartObject SupportedLinks and the addLink procedure

There are standard links associated with each SmartObject class. The **SupportedLinks** ADM property for the class names these links. Each link is defined with a set of ADM property values and functions to support getting and setting those properties and are described in this section.

SmartLinks are established when a SmartObject is created at runtime, when the createObjects procedure runs **addLink**. All these supporting elements are already present for all of the standard shipped SmartObjects. This description is here to help developers understand how the ADM link mechanism works and how it can be extended for new SmartObject types.

SmartLinks are typically established when you define links in the Container Builder while assembling SmartWindows or other Containers. When you define a template container in the Container Builder, you also define links between the objects in the template. When you define a specific window using that template, you can add additional links. For example, you can add a link that connects objects on different pages that you assemble. For each link, a record is added to the repository to define the link. At runtime, the dynamic window procedure runs addLink to create those links:

```
PROCEDURE addLink:
```

```
Parameters:
```

```
    (INPUT phSource /* HANDLE - Source procedure handle */,
```

```
    INPUT pcLink /* CHARACTER - Link name */,
```

```
    INPUT phTarget /* HANDLE - Target procedure handle */).
```

```
Candidate for: calling
```

```
    If you need to create your own additional links at runtime based on  
    application requirements, you do this by running addLink.
```

The `addLink` procedure requires that three basic properties and supporting functions for those properties be pre-defined in the SmartObject support files (its include files and super procedures). In the Source procedure for the link there must be a property that holds the handle or handles of the Targets for each Supported Link. In the Target for the link there must be a property that holds the handle or handles of the link Source for each Supported Link. These Source and Target procedures use functions that set property values in linked SmartObjects or return the handles of those linked objects when needed. At startup time, `addLink` sets these properties to the appropriate procedure handles. The names of the properties must be `<LinkType>Target` in the Source SmartObject, and `<LinkType>Source` in the Target object.

In addition, **`addLink`** requires that there be a property in the Target SmartObject listing the events that the Target object wants to be notified of when those events are published by the Source. The name of this property must be `<LinkType>SourceEvents`, and must be a comma-separated list of event names. If the Source of the link also wants to be notified of events in the Target, then there must be a `<LinkType>TargetEvents` property in the Source SmartObject.

The `addLink` procedure looks at the **propertyType** of the `<LinkType>Target` and `<LinkType>Source` properties (which returns the data-type of the property) to determine whether the link supports a single object or multiple objects. Typically a SmartLink supports only a single Source, but can support more than one Target. For example, a Container can contain many other SmartObjects, which are its Container-Targets, but each of those Objects has a single Container-Source. Likewise, a SmartObject such as a SmartDataObject or SmartDataViewer can have only a single Data-Source, but an SDO can have multiple Data-Targets. Additional user-defined SmartLinks for new SmartObjects can be defined to support single or multiple Sources and single or multiple Targets. This is done in the definition of the properties where the handles are stored.

If the data type of the property is `HANDLE` (which is typically the case for the `<LinkType>Source` property), then this tells `addLink` that only a single object is permitted on that end of the link, and it runs the `set<LinkType>Source` function to assign the property value to the Source procedure handle.

If the data type of the property is `CHARACTER` (which is typically the case for the `<LinkType>Target` property), then this tells `addLink` that *multiple* objects are permitted on that end of the link, and it runs the `modifyListProperty` procedure to add the Target procedure handle to the list of handles of objects which are Targets of that link (or the Source if the Source supports multiple SmartObjects). See the [Special functions that manage properties](#) section for a description of `propertyType` and `modifyListProperty`.

The **addLink** procedure then looks at the <LinkType>SourceEvents property in the Target SmartObject, and subscribes the Target procedure to each of those events in the Source procedure handle. This causes the event procedure of the same name as the event to be run in the Target each time the Source publishes that event. In some cases the Source of a link also receives messages from the Target. In this case there is also a <LinkType>TargetEvents property in the Source SmartObject, and addLink subscribes the Source to those events in the Target.

The lists of the events to which Source Events and Target Events of each standard Supported Link are subscribed are in a series of tables in the SmartLinks chapter of the ProgressVersion 9 ADM documentation.

To look at a specific example, examine the **Container** link. This link is not on the list of **SupportedLinks** for any SmartObject because **every** SmartObject can be a Container-Target, and every SmartContainer™ can be a Container-Source, so this link is created automatically when a SmartContainer is run and its contents are started (the addLink call to establish the Container link is in **constructObject**).

Because every SmartObject can be a Container-Target, the properties that define the Source handle and events are in the property include file that all SmartObjectss share, `smrtprop.i`. They are of course also defined in the Dynamics repository. The **ContainerSource** property is defined as data-type HANDLE, to indicate that there can be only a single Container-Source for a SmartObject. The ContainerSourceEvents property lists the events to which each SmartObject is subscribed in its parent Container. This list includes initializeObject, hideObject, viewObject, destroyObject, enableObject, and confirmExit. When a Container is initialized, for example, it publishes initializeObject, which causes each of its child SmartObjects to be initialized as they receive that event and run their own initializeObject procedure. The supporting functions getContainerTarget, setContainerTarget, getContainerTargetEvents and setContainerTargetEvents are defined in `smart.p`, the super procedure used by all SmartObjects.

Only a Container can be a Container-Source, so the properties describing the Target are defined in the property include file for Containers, `cntnprop.i`, and also in the repository. There is a ContainerTarget property (of type CHARACTER), and a ContainerTargetEvents property, because each Container subscribes to a single event in each of its child objects, exitObject. The super procedure for Containers, `containr.p`, contains the get and set functions for these properties.

5.3.2 Defining custom SmartLinks and events

In some cases custom code added to SmartObjects needs to send an event from one SmartObject to another. If there is just one event type to send, you can define a dynamic SmartLink. To do this, add your own call to the `addLink` procedure at the appropriate place in your application code, and use the name of the event to be subscribed to, which must be the same as the internal procedure name to be executed when the event occurs. When `addLink` sees this reference to a link that is not in the `SupportedLinks` list for either the Source or Target Object, it registers a single `SUBSCRIBE` statement for the Target procedure for an event of the same name as the link, and subscribes to the event in the Source Object. Because there is no predefined Object property for the event and no *get* or *set* functions to manage the property value, a support procedure called `modifyUserLinks` stores the Source and Target handles in the `ADM-DATA` procedure attribute of the two Objects, so that the `linkHandles` function (described later) can return the object handles when they are requested. The exact format of the storage of these handles should not concern the developer; always use the functions provided to get at these and other ADM values.

5.3.3 Link support methods

There are several additional methods that are used to modify and support links, including the following:

removeLink, removeAllLinks, removeUserLinks

These functions delete one or links from an Object. The `removeLink` function removes a specific link between two objects:

Procedure removeLink:

```
Parameters:  INPUT phSource AS HANDLE -- source procedure handle,
              INPUT pcLink   AS CHARACTER -- link type name,
              INPUT phTarget AS HANDLE -- link target object handle
```

Candidate for: **calling**

If your code needs to remove a specific link between two objects, it should use this call.

Your code does not normally run `removeAllLinks`, which deletes all standard (supported) links from an object. This includes deleting link definitions in the objects at the other end of the links, which is done automatically when the object is destroyed.

The procedure `removeUserLinks` removes user-defined links, which are not on the `SupportedLinks` list.

assignLinkProperty

This LOGICAL property sets a property value in an object at the other end of a specified link, relative to the current object (TARGET-PROCEDURE):

Function assignLinkProperty()

Parameters: INPUT pcLink AS CHARACTER -- Link Type,
INPUT pcPropName AS CHARACTER -- Property Name,
INPUT pcPropValue AS CHARACTER -- Property Value.

Returns: LOGICAL (success flag)

Candidate for: **calling**

You can use this function when you know the name of the link that connects the current Object to another, but not the handle of the Object at the other end of the link. This allows you to set the property based on the relationship between the objects without having to retrieve the handles. If there's more than one object at the other end of the link, the property is set in all of them.

linkHandles

This CHARACTER function takes a link name and returns a list of handles of objects at the other end of that link, relative to the current object (the TARGET-PROCEDURE):

Function linkHandles()

Params: pcLink AS CHARACTER -- Link name (including "-SOURCE" or "-TARGET")

Candidate for: **calling**

Your code can use this function to retrieve a list of the procedure handles of the object or objects connected to the current object (the one the function is invoked in) by the link you specify. This could be useful if you need to do something beyond just setting a property value (for which you could use assignLinkProperty).

linkProperty

This CHARACTER function returns the value of the requested property in the object at the other end of the specified link, relative to the TARGET-PROCEDURE. Parallel to the assignLinkProperty function, this one retrieves the value of a property using the link name rather than the Object's handle. If there is more than one object at the other end of the link (or none), the function returns blank:

Function linkProperty()

Params: INPUT pcLink AS CHARACTER -- Link name,
INPUT pcPropName AS CHARACTER -- Property name.

Candidate for: **calling**

You can invoke this function in your code to retrieve a property value when you know the link that associates two objects but not the handle of the object at the other end of the link.

5.3.4 Miscellaneous link properties

This section describes some useful link properties.

SupportedLinks

This CHARACTER property holds the list of all links routinely created for an object. It is a comma-separated list of link names, including the –Source or –Target suffix. As described earlier, you need a <link>Source or <link>Target property to hold the handle(s) of the object(s) that use the link, along with a <link>SourceEvents or <link>TargetEvents property to hold the list of the events that the Object should subscribe to in the related Object.

This property is initialized in the property include files for the various SmartObjectss. Because of the dependencies on other property names, you should not modify it. The user link mechanism allows you to create additional links dynamically when you need them.

<link>Source and <link>Target

For each link in an Object's **SupportedLinks** list that is a <link>–Source, there must be a <link>Target property for the Object. This holds the handle of the Target for the link at runtime. If the Object can have multiple Targets for the link, then the datatype of the property must be CHARACTER, to hold the list of Object handles. If the Object can have only one Target for the link, then the datatype of the property should be HANDLE, to hold that one procedure handle in native form. Likewise, for each link in an Object's SupportedLinks list that is a <link>–Target, there must be a <link>Source property for the Object, which holds the handle of the Source at runtime. The same datatype rules apply.

<link>SourceEvents and <link>TargetEvents

As described earlier in this section, these properties store the list of events that the Object should be subscribed to in the handle of the Object at the other end of the link. The **constructObject** procedure uses these properties together to accomplish all of the linking of objects, using the 4GL PUBLISH and SUBSCRIBE statements.

InactiveLinks

This CHARACTER property contains a list of all links and handles for an Object that are currently inactive. See the States and Actions section for more information.

5.4 Customizing and managing queries

The *Progress Dynamics Developer's Guide* has a chapter on Business Logic that reviews how SmartDataObjects function and how to use the built-in entry points for various kinds of basic validation logic. Other chapters discuss more advanced forms of business logic. This section reviews the following useful functions and properties that you can use to customize the WHERE clause and other aspects of the queries in your application:

- [Modifying the WHERE clause at runtime](#)
- [Using addQueryWhere and assignQuerySelection](#)
- [Repositioning the database query](#)
- [Refreshing the database query](#)
- [Using resortFiltering and sorting the RowObject query](#)
- [Additional query methods](#)
- [Other query properties](#)

5.4.1 Modifying the WHERE clause at runtime

You can modify the base database query WHERE clause in many ways at runtime. When you link multiple SmartDataObjects together in a parent-child relationship, this modification is done automatically.

Using ForeignFields to filter a dependent query

If you want to browse through Customers and then through Orders of a selected Customer, you can do this by creating separate SDOs, one for Customers and one for Orders, and then linking them together. When you do this, the Order SDO uses a property called **ForeignFields** to modify the WHERE clause dynamically, so that the Order SDO that was originally defined with the simple query “FOR EACH Order” is now refined to select “EACH Order WHERE Order.CustNum = <CustNum>”, in which the value of the <CustNum> field is retrieved from the parent Customer SDO each time a new Customer is selected, and plugged into the Order query before that query is re-opened. You can define the appropriate ForeignFields value either in the AppBuilder for static window procedures, or in the Container Builder for dynamic windows.

For the Customer/Order example, the initialization code for the Order SDO assigns the value `Order.CustNum,CustNum` to the **ForeignFields** property. Each time a new Customer is selected at runtime, the **dataAvailable** event is published, and that event procedure in the Order SDO queries the Customer SDO for the current value of the Foreign Field *CustNum*. Note that this field refers to a field in the Customer RowObject table, and therefore is not qualified by a table name (it could in fact be renamed from the actual database field it is derived from). This field's value is used to set the property **ForeignValues**, which holds the current values of a SmartDataObject's ForeignFields. The WHERE clause of the Order SDO is then modified for you to insert the phrase `Order.CustNum = <CustNum>`, where `<CustNum>` is filled in from the ForeignValues property, and the query is prepared and opened. A dependent query can contain multiple Foreign Fields, and the values for them are kept in sync with the list of fields that make up the foreign key.

The functions `setQueryWhere` and `setQuerySort`

The earliest versions of the Version 9 SmartObjects supported two basic functions to manipulate the WHERE clause: **setQueryWhere** and **setQuerySort**. These work well in straightforward cases, but `setQueryWhere`, in particular, does not deal well with successive changes to the WHERE clause or other more complex needs. Therefore we do not recommend that you use `setQueryWhere` in new application code. The `setQuerySort` function can be used to change the sort sequence of a query, but remember that any query that has a WHERE clause or other filter applied to it is sorted automatically based on the indexes used to satisfy the query, and this is typically the most appropriate sort sequence.

If you want to allow users to sort a result set in different ways after it has been retrieved, you can do this more efficiently by manipulating the temp-table query instead of sorting and reopening the database query, as we discuss later in this section.

Instead, you should generally use the `addQueryWhere` and `assignQuerySelection` functions described below. These are much more flexible and can operate more efficiently.

The `resortQuery` function

The `resortQuery()` function takes an existing query and resorts it according to criteria you specify. For example:

```
DYNAMIC-FUNCTION('resortQuery':U IN h_dorder, INPUT 'BY ordernum DESCEND').
```

The **setBaseQuery** function

There is an additional function that may be of use in some cases, if you want to set a basic filter on a dataset so that it cannot be lost by changes to the WHERE clause later on. The property that stores the base query is **BaseQuery**, whose initial value is the query defined when the SDO is created. To apply a filter at runtime that is not removed by other changes, you can reset this property, effectively overriding the basic definition of the SDO's query in that instance. In the following example we have run **setOpenQuery** to change the basic query to return only Customers where the State field equals MA. All calls to the other WHERE clause functions then append their WHERE clause to this new basic query definition. Note that because we're resetting the entire Open Query statement, we need to specify the query starting with FOR EACH.

NOTE: SDOs do not support the use of this function to change the database tables the query operates on.

You can override another default behavior of the SDO, which is to open its database query as soon as the SDO is initialized. You may not want to wait for that to happen if the SDO is always filtered before the data is actually used. In this case you can reset the **OpenOnInit** property of the SDO, mentioned earlier in this chapter, to **FALSE** to cause it to wait until the **openQuery** function is run after initialization.

5.4.2 Using **addQueryWhere** and **assignQuerySelection**

Use the **addQueryWhere** function and the **assignQuery Selection** function to build more complex and efficient queries.

addQueryWhere

The **addQueryWhere** function takes three INPUT parameters:

- The new **WHERE clause** to be added **to** the existing one.
- An optional **buffer name** to specify which buffer in a join to append the WHERE clause to.
- An optional **Operator** to connect multiple WHERE clause fragments passed in successive calls. The default is AND, but you can also specify OR.

If you do not specify the buffer name parameter, then you must qualify the field names in your WHERE clause with their table name in order to allow the function to build the WHERE clause most efficiently, with each phrase appended to the query join clause where that table appears in the join sequence. The SDO property **QueryString** stores successive changes to the WHERE clause. This property is stored in the client SDO (if the SDO is split between client and server). When **openQuery** is run, it checks to see if the QueryString property has a value, and if so, sends it to the server, prepares the database query using that value, and then opens the query. This allows your code to build up a complex WHERE clause efficiently, without preparing or opening the intermediate steps (or even sending them to the server) until the signal is given to open the database query.

For example, if you create a new SmartWindow called `waddwhere.w`, you can use the `addQueryWhere` function to allow filtering of a query. To show how the function operates on a joined query, use a SmartDataObject called `djoin.w`, which joins “EACH Order, Customer OF Order, SalesRep OF Order”. There is a field called `Where-Field` in the Window which you can use to enter a new WHERE clause phrase, with this trigger code:

```
/* ON LEAVE OF Where-Field */
DO:
  IF Where-Field:SCREEN-VALUE NE '' then
    DO:
      DYNAMIC-FUNCTION('addQueryWhere' IN h_djoin,
        Where-Field:SCREEN-VALUE, '', '').
      Editor-1:SCREEN-VALUE IN FRAME {&FRAME-NAME}
        = DYNAMIC-FUNCTION('getQueryString' IN h_djoin).
    END.
  END.
```

AssignQuerySelection and removeQuerySelection

The **assignQuerySelection** and **removeQuerySelection** functions are designed to make it easy to map values to fields to create a WHERE clause. These functions are especially suited to application procedures such as a Query By Form, where you enter a value for one or more fields and expect to see database records matching those values.

The first of these functions, **assignQuerySelection**, takes three INPUT parameters:

- Comma-separated list of **fieldnames** to be used in the WHERE clause.
- A CHR(1)-delimited list of **values** for those fields.

- An **operator** or list of operators to apply to the values. If this third parameter is not specified, EQUALS is the default. If a single operator is supplied for that parameter, it is applied to all field-value pairs. The special value of EQ/BEGINS means that numeric fields should use the EQ operator, and CHARACTER fields the BEGINS operator. Or a comma-separated list of operators can be supplied, and each operator in the list is applied to the corresponding field-value pair.

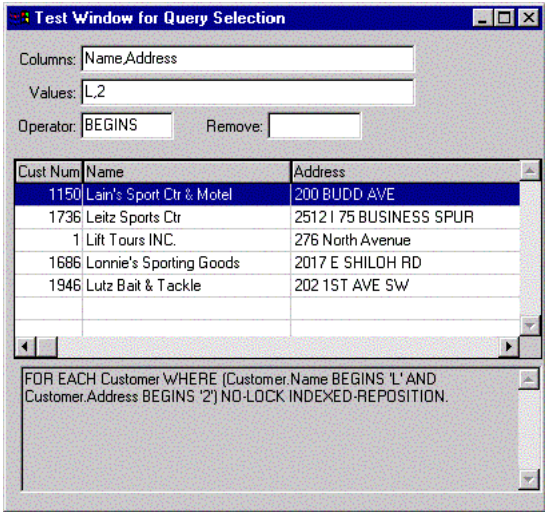
For example, you can use a SmartWindow called `wselection.w` to apply these functions to a Customer SDO. You can enter a list of fields in the Column-Field, and you can enter a list of values in the Values-Field. To simplify the example enter the values as a comma-separated list and replace the commas with `CHR(1)` when you use the list. Enter an operator in the third field. The leave trigger puts the pieces together and opens the SDO query with the values specified:

```
/* ON LEAVE OF Operator-Field */
DO:
  DYNAMIC-FUNCTION('assignQuerySelection' IN h_dcust,
                    Column-Field:SCREEN-VALUE,
                    REPLACE(Values-Field:SCREEN-VALUE, ",", CHR(1)),
                    Operator-Field:SCREEN-VALUE).
  EDITOR-2:SCREEN-VALUE =
    DYNAMIC-FUNCTION('getQueryString' IN h_dcust).
  DYNAMIC-FUNCTION('openQuery' IN h_dcust).
END.
```

There is also a Remove-Field, which allows you to name a field that contains a WHERE clause fragment that you want to remove. Remove-Field executes the **removeQuerySelection** function, which takes two INPUT parameters: the name of the field and the operator associated with that field:

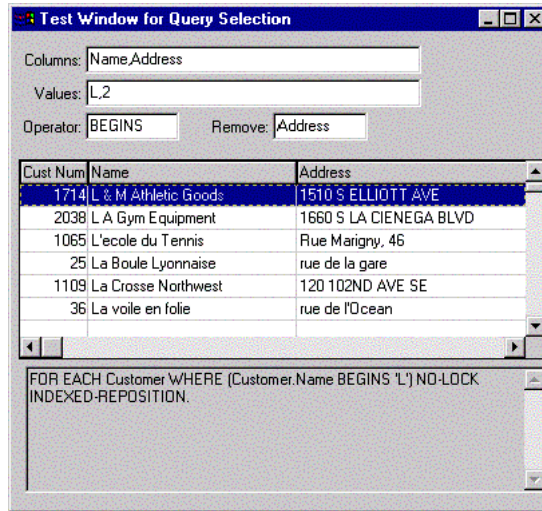
```
/* ON CHOOSE OF Remove-Field */
DO:
  DYNAMIC-FUNCTION('removeQuerySelection' IN h_dcust,
                    Remove-Field:SCREEN-VALUE,
                    Operator-Field:SCREEN-VALUE).
  EDITOR-2:SCREEN-VALUE =
    DYNAMIC-FUNCTION('getQueryString' IN h_dcust).
  DYNAMIC-FUNCTION('openQuery' IN h_dcust).
END.
```

For an example of how these functions work, enter two field names and values for those fields, plus the BEGINS operator, and see all Customer where the Name BEGINS with **L** and the Address BEGINS with **2**:



Note that one feature of the assignQuerySelection function is that it handles quotation marks properly. The values entered into the list **must** not be in quotation marks. It is easier to assemble a list of values that come out of a list of fill-in fields for different database fields without having to add code to put quotes around those which need it (CHARACTER and DATE types); assignQuerySelection does this for you. Note that assignQuerySelection also identifies the table that each field comes from, and qualifies the field name references. If there is a join involved, it also distributes the where clause phrases properly for maximum efficiency.

Entering Address into the Remove field runs `removeQuerySelection`, which removes that part of the WHERE clause from the query:



5.4.3 Repositioning the database query

Often an application needs to reposition a query to a particular row. Once a row in the `RowObject` table has been populated from the database, its `RowIdent` field, which is constructed from the database ROWID(s) of the record(s) it is derived from, serves as a useful key. Several utility functions and procedures make use of this key field. This section introduces you to some of them.

Frequently an application needs to allow a user to select a record from the database based on a key value. For large data-sets this is obviously more reasonable than using the Next and Prev buttons in a Toolbar. These buttons are intended for navigating through small numbers of records. The `rowidWhere` and `fetchRowIdent` functions make it easy to support selection of a record in a large data set.

The **`rowidWhere`** function takes as its one INPUT parameter a WHERE clause for the SDO's database query. It then returns the database ROWID of the first record in the query that matches the Where clause. Normally this is useful if the WHERE clause defines a unique key value, such as `CustNum = n` for the Customer table. You can then turn around and pass this ROWID to another function, **`fetchRowIdent`**, which takes the ROWID as an INPUT parameter, along with an optional list of SDO columns for which values should be returned. The `fetchRowIdent` function not only returns those column values, but also repositions the SDO query to that record, while fetching additional batches of rows from the database if a particular record is not in the current batch. These functions can thus be used in sequence to reposition to a desired key value.

For example, the SmartWindow called `wrepos.w` demonstrates these functions. The trigger on the `Cust-Field` inserts the number entered for `CustNum` into the `WHERE` clause “`CustNum = <>`” and passes it to `rowidWhere`. It then passes the result to `fetchRowIdent`. Note that no column values are requested of `fetchRowIdent`:

```
/* ON LEAVE OF Cust-Field */
DO:
  DEFINE VARIABLE cRowId AS CHAR NO-UNDO.

  cRowId = DYNAMIC-FUNCTION('rowidWhere' IN h_dcust,
    "CustNum = " + Cust-Field:SCREEN-VALUE).
  IF cRowId NE ? THEN
    DYNAMIC-FUNCTION ('fetchRowIdent' IN h_dcust, cRowId, '').
END.
```

When it repositions the query to the new row, the standard SDO mechanisms send the values for that row to the SmartDataViewer in our Window, so they are displayed without any additional code needed, as you can see in [Figure 5-4](#).

The screenshot shows a window titled "Test Window for repositioning" with a menu bar (File, Navigation, Window, Help) and a toolbar. The form contains the following fields and controls:

- CustNum:** A text box containing "26". To its right are four buttons: "Mark this Row", "Modify Row", "Return to Mark", and "Refresh Row".
- Cust Num:** A text box containing "26".
- Name:** A text box containing "Bulls Eye Sports".
- Address:** A text box containing "Highscore House".
- City:** A text box containing "Leeds".
- West Yorkshire:** A text box containing "West Yorkshire".
- Country:** A text box containing "United Kingdom".
- Postal Code:** A text box containing "LS17 8TS".
- Sales Rep:** A dropdown menu showing "SLS / Smith, Spike Louise (West)".
- Credit Limit:** A text box containing "47,700".
- Balance:** A text box containing "1,236.54".
- Terms:** A text box containing "Net30".
- Discount:** A text box containing "50%".
- Fax:** An empty text box.
- Phone:** A text box containing "0347 21348".
- Email:** An empty text box.
- Contact:** A text box containing "Kevin Balter".
- Comments:** A large empty text area.

Figure 5-4: Test window for repositioning

This Window has buttons labelled Mark this Row and Return to Mark. Use the Mark this Row button to mark a row which you want to return to later and the Return to Mark button to return to the marked row. The Mark this Row button uses the **colValues** function. The colValues function takes as an INPUT parameter a list of SDO column names for which values should be returned. The first value returned is always the RowIdent of the current row. We save this off in a variable to refer to later:

```
/* ON CHOOSE OF Mark-Row */
DO:
  cRowIdent = DYNAMIC-FUNCTION('colValues' IN h_dcust, '').
END.
```

The Return to Mark button uses the saved value to get fetchRowIdent to reposition to that row again:

```
/* ON CHOOSE OF Return-Btn */
DO:
  DYNAMIC-FUNCTION('fetchRowIdent' IN h_dcust,
    ENTRY (2, ENTRY(1, cRowIdent, CHR(1))), '').
END.
```

The term RowIdent can have two slightly different meanings. First, the RowObject temp-table definition for an SDO includes a CHARACTER field called RowIdent. As each row from the database query is read, the ROWID(s) of the database record(s) that make up that row are written into this field in string form, or as a comma-separated list if there is a join involved. Therefore, our Customer SDO would have just a single ROWID in the RowIdent field for each RowObject record representing the database ROWID of the Customer record that the temp-table record came from. This field is used to re-retrieve the database record when an Update is done.

When the RowIdent is passed to other client objects such as our Viewer, the RowIdent field from the RowObject record is prepended with the ROWID of the RowObject Temp-Table record itself. The first entry in the list of column values returned by the colValues function, for example, uses this form. Therefore this first value returned for our Customer SDO has **two** ROWIDS, the ROWID of the RowObject record followed by the ROWID of the Customer record in the database. This is what we have stored in the cRowIdent field in the Mark-Row trigger. To re-retrieve the correct record through fetchRowIdent, first extract the first entry from the list of column values which colValues returned, which is CHR(1)-delimited, which is the RowIdent, and then extract the **second** entry from that (comma-separated) list, which is the database ROWID which fetchRowIdent expects.

When you press the Mark this Row button, its RowIdent is saved, and when you press Return to Mark, the query is repositioned to that row.

There are other functions that can also be useful in repositioning the query.

The **findRowWhere** function locates and repositions to a row in the dataset in the most efficient way, checking first on the client to see if the row is already present there. The function takes three INPUT parameters:

- **pcColumns** — This CHARACTER parameter is a comma-separated list of field names for which values are provided in the next parameter. Each name is a fieldname in a database table that is part of the SDO's query. This can be in the form Table.Fieldname or DB.Table.Fieldname. If the fieldname is not qualified, the function checks the tables in the SDO's Tables property and uses the first with a matching fieldname.
- **pcValues** — This CHARACTER parameter is a CHR(1)-delimited list of values for the fields in the pcColumns parameter. This combination of fields and values is used to identify uniquely a row in the dataset to reposition to.
- **pcOperators** — This CHARACTER parameter specifies the comparison operator(s) to be used for the field values. It can have these values:
 - blank — defaults to (EQ)
 - A single comparison operator indicates that this operator should be used for all field/values comparisons. Use a slash to define an alternative string operator, as in EQ/BEGINS.
 - If a different comparison operator is needed for each field/value pair, then specify those as a comma-separated list, one for each field.

The findRowWhere function checks first on the client to see if the row is already in the SDO's client-side temp-table. If so, it repositions to that row. Otherwise it uses the fetchRowIdent function to obtain the row from the server. As such it is a more efficient alternative to using that function.

The findRow function is a simpler alternative to using findRowWhere if the WHERE clause to identify the row simply uses the table's key as defined in the KeyFields property. It takes a single INPUT parameter, which is either a comma-separated or CHR(1)-separated list of values for the table key fields. This is used to reposition to the row. The findRow function in fact turns around and uses findRowWhere to locate the row, matching up the KeyFields with the input list of values.

Both of these functions return a LOGICAL value: True if the row was located and False if no unique row exists satisfying the matching criteria.

5.4.4 Refreshing the database query

One of the most important things to keep in mind when using SDOs is that they always operate on a temp-table that is a snapshot of the database at the time the temp-table was built. This represents a compromise of some of the immediacy that older Progress applications could take advantage of. In an older host-based or client-server application, you always knew that the database record you were looking at was one you were actually positioned to, and that you could lock a record while you worked on it. The realities of distributed applications require us to design record processing in which the client is not so tightly coupled to the database. If you want to be able to run your application in a distributed environment, with databases on numerous machines in different locations, your client objects cannot sit on the **actual** database record, which is in a database they are not even connected to.

Therefore you sometimes need to refresh the RowObject query your client objects are looking at. For example, a part of the application may be using a row when another part of your application signals that a row has been modified. One way to do this is simply to invoke the **openQuery** function. This reopens the database query and reloads the RowObject Temp-Table. If you wish, you can save off the RowIdent of the row your client is currently positioned to and reposition to that record after the query is reopened, using code similar to that in our Mark and Return example above.

If you simply want to refresh the row your client is currently looking at, you can refer to the third example in the sample reposition window. The Modify Row and Refresh Row buttons modify and then refresh a database record. The code under the Modify Row button simulates the situation of having a database record changed by another user while you are looking at it. It retrieves the current Customer record and changes the name:

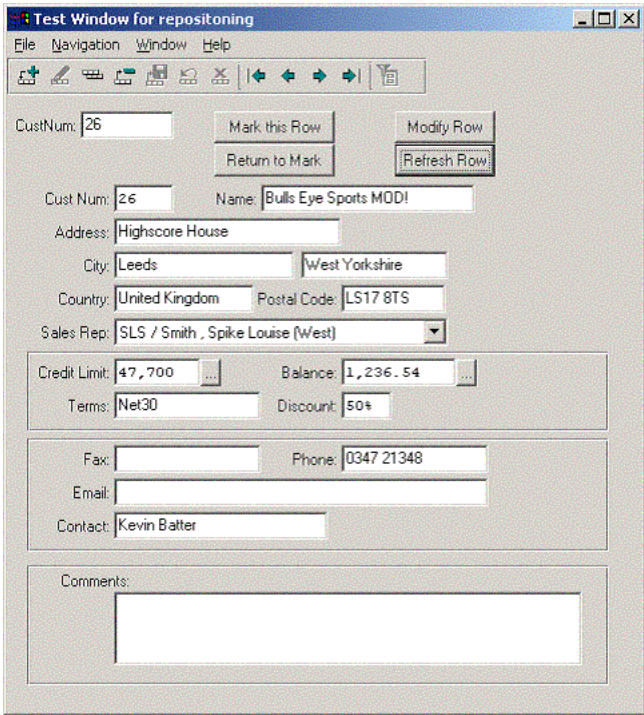
```
/* ON CHOOSE OF Modify-Btn */
DO:
  FIND customer WHERE customer.custNum = INT(Cust-Field:SCREEN-VALUE)
  NO-ERROR.
  IF AVAILABLE(Customer) THEN
    Customer.NAME = Customer.NAME + " MOD!".
  END.
```

NOTE: Keep in mind that this is just a test procedure. In a real application window you would not do a database retrieval like this directly.

You do not see this change because it does not immediately affect your RowObject dataset. If you want to confirm that you are looking at the latest values for a record, you can press the Refresh Row button. This runs the **refreshRow** procedure, which re-reads the current RowObject row out of the database and writes its current field values into the RowObject table:

```
/* ON CHOOSE OF Refresh-Btn */
DO:
  RUN refreshRow IN h_dcust.
END.
```

It also publishes the **dataAvailable** event, so that your Viewer requests those fresh values and redisplay them. Press the dataAvailable button to see the “MOD!” to the Customer record:



5.4.5 Using resortFiltering and sorting the RowObject query

All the functions shown so far apply to the database query, causing it to be reopened, and rebuilding the RowObject temp-table dataset. You may also need to filter or sort the RowObject dataset after it has been retrieved. This eliminates the need to re-retrieve records from the database and reload the RowObject table each time you want a different variation of the dataset. The SmartDataObject interface currently provides functions for filtering and re-opening the RowObject dataset, making this is easy to do. The handle of the RowObject query is available as the **DataHandle** property, and by using this handle application code can re-prepare the query and re-open it as needed. However, a supporting property and function take care of this for you, without the need for you to manipulate the query yourself. The DATAQueryString property holds the query for the SDO's temp-table (always beginning FOR EACH RowObject). The openDataQuery function prepares and opens the temp-table query based on the value of the property. It takes an INPUT parameter whose value must be First or Last, depending on whether you want to position the query to the first or the last row in the dataset after it is opened.

NOTE: Most of the techniques in this section can be done more easily with the resortQuery function.

This section shows a simple example of how to use these to filter the client-side SDO temp-table.

This sample SmartWindow (called `wdatafilter.w`) has fill-in fields for the WHERE clause and sort (BY) phrase for the RowObject query, and a Reset button to reset the query to its original state.

Make sure that all database records for the dataset have been retrieved and moved into the RowObject table. If only some batches of an incomplete dataset have been retrieved, a sort or subset of the first part of the complete dataset is not likely to be meaningful.

For this reason, these examples check first that all rows have been retrieved by checking the **LastRowNum** property of the SDO. If this has been set, it means that all rows in the database query have been retrieved and copied into the RowObject table. If this is not the case, the trigger code for the Where and Sort fields runs the **fetchLast** procedure in the SDO. This loads all remaining batches of rows into the RowObject table until the last row of the database query is reached. Remember that simply executing the GET-LAST method on the RowObject query does not accomplish this; GET-LAST only repositions to the last of the rows that have already been retrieved. The fetchLast procedure contains the code to go back to the database for more batches of rows as needed. Note also that for this to work properly, the **RebuildOnRepos** property of the SDO must be set to false. This tells the SDO code to load successive batches of rows until the last row is reached. If the RebuildOnRepos property is set to true, then the database query jumps directly to the end and the RowObject table contains only a batch of rows representing the last rows of the database query.

So the LEAVE trigger code for the Where-Field checks the LastRowNum property, runs fetchLast if necessary, and then appends the WHERE clause entered to the RowObject RowObject query definition, assigns this to the DataQueryString property, and prepares and opens that dynamic query by running the openDataQuery function. Because this nullifies any previously entered sort phrase, the Sort-Field is blanked out to reflect this:

```
/* ON LEAVE OF Where-Field */
DO:

    DEFINE VARIABLE lReturn      AS LOGICAL NO-UNDO.
    DEFINE VARIABLE cWhere AS CHARACTER NO-UNDO.

    IF Where-Field:SCREEN-VALUE NE "" THEN
    DO:
        Sort-Field:SCREEN-VALUE = "".
        IF DYNAMIC-FUNCTION('getLastRowNum' IN h_dcust) EQ ? THEN.
            RUN fetchLast IN h_dcust.
            cWhere = "FOR EACH RowObject WHERE " + Where-Field:SCREEN-VALUE.
            {set DataQueryString cWhere h_dcust}.
            DYNAMIC-FUNCTION('openDataQuery' IN h_dcust, 'first').
        END.
    END.
```

The Sort-Field trigger has similar code, which takes the current RowObject query, including any WHERE clause that may already have been added to it, and adds a BY phrase to the end, replacing any existing BY phrase from an earlier sort:

```
/* ON LEAVE OF Sort-Field */
DO:
    DEFINE VARIABLE cPrepare      AS CHAR      NO-UNDO.
    DEFINE VARIABLE lReturn      AS LOG      NO-UNDO.
    DEFINE VARIABLE iBy          AS INT      NO-UNDO.

    IF Sort-Field:SCREEN-VALUE NE '' THEN
    DO:
        IF DYNAMIC-FUNCTION('getLastRowNum' IN h_dcust) EQ ? THEN
            RUN fetchLast IN h_dcust.

        {get DataQueryString cPrepare h_dcust}..
        iBy = INDEX(cPrepare, ' BY ').
        IF iBy NE 0 THEN
            cPrepare = SUBSTR(cPrepare, 1, iBy).
            cPrepare = cPrepare + 'BY' + Sort-Field:SCREEN-VALUE.
            {DYNAMIC-FUNCTION('openDataQuery' IN h_dcust, 'first')}..
        END.
    END.
```

Finally, there is a Reset button that restores the RowObject query to its original state -- with all rows retrieved from the database, but with no additional WHERE clause. To do this, it simply resets the DataQuery String to **FOR EACH RowObject**:

```
/* ON CHOOSE OF Reset-Btn */
DO:
    DEFINE VARIABLE cQuery AS CHARACTER NO-UNDO INIT "FOR EACH RowObject".

    ASSIGN Where-Field:SCREEN-VALUE = ""
           Sort-Field:SCREEN-VALUE = ""

    {set DataQueryString cQuery h_dcust}.

    DYNAMIC-FUNCTION('openDataQuery' IN h_dcust, 'first').

END.
```

Here are a couple of simple examples of how our SmartWindow works. Note that there is a delay when the first WHERE or sort clause is entered, as the remaining batches of rows are retrieved from the database. After that, filtering or sorting the RowObject query is extremely fast.

After the window initially comes up, enter a WHERE clause for the Name field:

The screenshot shows a window titled "Test Window for Filtering RowObject". It contains two input fields: "Where:" with the text "Name BEGINS L" and "Sort By:" which is empty. A "Reset Query" button is located to the right of the "Sort By" field. Below these fields is a table with three columns: "Cust Num", "Name", and "Address". The table contains six rows of data, with the first row highlighted in blue.

Cust Num	Name	Address
1	Litt Tours INC.	276 North Avenue
25	La Boule Lyonnaise	rue de la gare
28	Lustin ja Pyora Oy	Ruutintie 7 A
36	La voile en folie	rue de l'Océan
63	Luopioisten Biljardi	Anna-Liisankatu 9
65	Lagt Kort Ligger	Bridgevagen 106

It may take a few seconds for the results of this filter to appear, because all the records in the customer table have to be retrieved and added to the temp-table first.

Apply a sort sequence to this filtered dataset. This reopens the RowObject query with the Sort phrase added to the end:

The screenshot shows a window titled "Test Window for Filtering RowObject". It has two input fields: "Where:" with the text "Name BEGINS 'L'" and "Sort By:" with the text "Address". There is a "Reset Query" button. Below the inputs is a table with three columns: "Cust Num", "Name", and "Address". The table contains six rows of data, with the first row highlighted in blue.

Cust Num	Name	Address
1823	Louisville Sports & Rec	100 URTON LN
1493	Larry Black Sporting Goods	1001 S MAIN ST
1888	Lolo Sporting Goods	1026 MAIN ST
2102	Lighthouse Grocery & Liquor	10750 N TONGASS HWY
2018	Leisure Time Sports & Video	110 SE FRONTIER AVE
1569	Lobo Store At The Pit	1111 UNIVERSITY BLVD SE

Press the Reset button to restore the complete original RowObject dataset:

The screenshot shows the same window "Test Window for Filtering RowObject". The "Where:" and "Sort By:" fields are now empty. The "Reset Query" button is still present. The table below now displays a different set of six rows, with the first row highlighted in blue.

Cust Num	Name	Address
1	Lift Tours INC.	276 North Avenue
2	Customer Two	Rattipolku 33
3	Hoopsville West	Suite 4150
4	Go Fishing Ltd	Unit 2
5	Match Point Tennis	66 Homer Place
6	Fanatical Athletes	20 Bicep Bridge Rd

All the records originally retrieved from the database are still in the RowObject temp-table; we have not actually removed them by applying the **Name BEGINS L** WHERE clause. As a result, you must keep applying this WHERE clause each time you reopen the query. If you wanted to filter the records by eliminating those that did not match the search criteria, so that the result set could be successively reduced, you could write code to delete those rows from the temp-table by creating a dynamic query on the RowObject buffer (obtainable through the SDO's *RowObject* property), and then doing a BUFFER-DELETE for each row not satisfying the WHERE clause. But once the database rows have been retrieved into the RowObject table, opening its query with a new WHERE clause is generally extremely fast. In the example here with a Customer table containing something over a thousand rows, the response for a new WHERE clause or sort sequence is more or less instantaneous. It is the time needed to load the RowObject table in the first place that causes a noticeable delay. For this reason it probably makes more sense to combine successive filters by appending them to the end of the query with AND in between than it would to bother to remove the rows from the RowObject table.

The individual application situation dictates whether it is more appropriate to filter the original database query or the RowObject query. As these examples show, it is necessary to load the entire database query result into the RowObject table before it can meaningfully be further filtered or sorted. If you are starting with an entire large table, this is not realistic – it takes too long to load the RowObject table in the first place. But once a result set with a reasonable number of rows has been loaded into the RowObject table, it may be more efficient to manipulate those further rather than going back to the database.

5.4.6 Additional query methods

Use either the `modifyNewRecord` method or the `collectChanges` method to make changes to records.

modifyNewRecord

The standard entry points for writing data validation are described in the business logic chapters of the *Progress Dynamics Developer's Guide*. However, there is an additional new entry point available to developers so that you can make changes to a newly created record before it is first displayed, normally to assign initial values. This is the **modifyNewRecord** method. There is no standard code for this procedure, but it is run NO-ERROR during record creation in the SDO. Therefore any code you write for it in your SDO data logic procedure is executed at the proper time. It takes no parameters, but you can refer to the temp-table record buffer in the same way that you do for validation procedures, using **b_** plus the name of the primary table for the SDO.

Keep in mind that because `modifyNewRecord` is executed only in the client-side SDO, it will not be effective for initializing a record using a WebSpeed front-end to the application.

Here's a simple example for our Customer SDO, which initializes several of the fields assuming that the Customer is from New Hampshire:

Procedure modifyNewRecord:

```
/*-----  
  Purpose:   Custom code for new records to initialize some of the fields  
             in the Customer table.  
  Parameters: <none>  
  Notes:     This is called automatically on Add or Copy.  
-----*/  
ASSIGN b_customer.State = 'NH'  
       b_customer.phone = '(603)'  
       b_customer.fax = '(603)'  
       b_customer.postalcode = '030xx'  
       b_customer.comments = "Here's another New Hampshire Customer!".  
  
END PROCEDURE.
```

When you run the Customer Maintenance application window and press Add, you see the initial values shown in [Figure 5-5](#).

Figure 5-5: Customer maintenance window

collectChanges

The collectChanges named event is published automatically when a Save occurs. The event cascades down through any visual objects that contribute changes to the current record, such as multiple Viewers with updatable fields from a single SDO. It takes two INPUT-OUTPUT parameters that give each of these objects (Viewers on different pages of a folder, for example) the opportunity to add their own changes and information about the changes to the growing list, passing it from procedure to procedure:

```
Procedure collectChanges:
  Params: INPUT-OUTPUT PARAMETER pcChanges AS CHARACTER
          INPUT-OUTPUT PARAMETER pcInfo    AS CHARACTER.
Candidate for: override
```

You can create a local version of this procedure if your code needs to intercept the collecting of modified fields from the different objects in the window that have them, for example to adjust the changed values, contribute additional ones, or trigger some other related event.

5.4.7 Other query properties

Other useful query properties include:

- **DataHandle** — This property returns the handle to an SDO's temp-table query. You can use this property to modify a query and re-open it, for example.
- **RowObject** and **RowObjUpd** — These HANDLE properties hold the buffer handles of the RowObject and RowObjUpd buffers, respectively.
- **RowObjectTable** and **RowObjUpdTable** — These HANDLE properties hold the handles of the RowObject and RowObjUpd temp-tables.
- **AutoCommit** — This LOGICAL property is set automatically, depending on whether there is a Commit button or other object in a window that acts as a Commit-Source for an SDO. If there is, then the property is set to FALSE. If there is no Commit-Source, then it is set to TRUE. When it is TRUE, any Save to an SDO record is sent immediately to the server for validation and update. When it is FALSE, then Saves are stored only locally in the client temp-table until the Commit event occurs. At that time all updates are sent back to the server together.

NOTE: Autocommit triggered by one SDO's update always applies to all unsaved changes in an SBO.

5.5 Paging methods and properties

There are several paging methods and paging properties that help manage objects and pages.

5.5.1 Paging methods

This section contains information about paging methods.

assignPageProperty

Use the assignPageProperty procedure to assign a property value to all the objects on the current page. If you need to set a property on a page that is not the current page, you can reset the CurrentPage property to the page you want, run the procedure, and then set it back:

Procedure assignPageProperty:

Params: INPUT PARAMETER pcProp AS CHARACTER
INPUT PARAMETER pcValue AS CHARACTER.
Candidate for **calling**

deletePage

Use the deletePage procedure to delete all the objects on the specified page:

Procedure deletePage:

Params: INPUT PARAMETER piPageNum AS INTEGER.

Candidate for: **calling**

hidePage

Use the hidePage procedure to hide all the objects on a page. Normally your code does not call this procedure directly, since the **selectPage** procedure automatically hides whatever was the previous page:

Procedure hidePage:

Params: INPUT PARAMETER piPageNum AS INTEGER.

initPages

Use the initPages procedure to initialize pages simultaneously. Normally pages are initialized when they are first viewed. The first page to be viewed is stored in the **StartPage** property, and all the objects on the page are started, initialized and viewed when the window is run. Sometimes, however, you need to initialize other pages at the same time in order to be able to create links to and from them, or to have objects prepared to receive data from objects on other pages. In this case you can call **initPages**, and pass it a list of the pages to be initialized on startup of the window:

Procedure initPages:

Params: INPUT PARAMETER pcPageList AS CHARACTER.

Candidate for: **calling**

viewPage

Use the viewPage procedure when you want another page to be displayed *without hiding* the objects on the current page. Normally this would be when the new page is in fact a different window, which you want to have appear in addition to the window containing the folder. Otherwise, when you want to hide one page and see another, run **selectPage**:

Procedure viewPage:

Params: INPUT PARAMETER piPageNum AS INTEGER.

Candidate for: **calling**

selectPage

Use the selectPage support procedure when you want to switch the folder display from one page to another. It hides the objects on the current page, resets the current page to the INPUT parameter, and views the objects on that new page:

Procedure selectPage:

Params: INPUT PARAMETER piPageNum AS INTEGER.
Candidate for: **calling**

targetPage

Use the targetPage INTEGER support function to return the page that a specified Object is on. Pass in the procedure handle of the Object:

Function targetPage()

Params: phObject AS HANDLE.
Returns: INTEGER

5.5.2 Paging properties

The section contains information about page number properties.

ObjectPage

The ObjectPage INTEGER property is defined for all SmartObjects. It holds the page number that the current instance of the object is on.

CurrentPage

The CurrentPage INTEGER property is defined for Containers. It holds the number of the currently selected page.

StartPage

The StartPage INTEGER property holds the page of the folder (beyond page 0, the background page) that should be initialized and selected on startup of the window.

InitialPageList

The InitialPageList CHARACTER property holds the list of page numbers that should be initialized on startup of the window, as a comma-separate character string.

RenderingProcedure

This procedure specifies the name of the procedure that will be used to render an object. The value of this attribute will match that of the object name in the Repository, and will be resolved into a filename with path capable of being run.

5.6 Special functions that manage properties

There are some special functions that let you define new properties at run time and also return useful information about standard Object properties. This section describes the following functions.

instancePropertyList

This CHARACTER function returns a list of the names of the object's InstanceProperties, which are properties that can be set to initial values in design mode. You can set these properties in the AppBuilder or in the Container Builder to determine the Object instance's behavior at runtime. The list of standard instance properties is part of the definition of the Object, and stored in its **InstanceProperties** property described just below. If you pass in a blank ("") for the INPUT parameter, you get back all instance properties and their values. Otherwise you can pass in a comma-separated list of property names and get the values for just those properties back. If you want values for *every* property defined for the Object, you can pass in an asterisk ("*") as the INPUT parameter.

The RETURN value is a delimited list of property name/value pairs with CHR(3) between pairs and CHR(4) between name and value:

<p>Function instancePropertyList: Params: INPUT pcPropList AS CHARACTER. Returns: property values. Candidate for: calling</p>

propertyType

The propertyType CHARACTER function returns the datatype of a property:

Function propertyType:

Params: INPUT pcPropName AS CHARACTER.

Returns: CHARACTER datatype of the property.

setUserProperty

This LOGICAL function assigns a value to a dynamically defined property and allocates a slot for the property if it does not yet exist. You can use setUserProperty to create properties for SmartObjects on the fly, without having to define functions to support them or predefine them in any way. It takes as INPUT parameters the name of the property and the value. Because these ad hoc properties and values are simply stored in a delimited string within the SmartObject, the values are always represented in CHARACTER form:

Function setUserProperty:

Params: INPUT pcPropName AS CHARACTER,

INPUT pcPropValue AS CHARACTER.

Returns: LOGICAL (always true)

Candidate for: **calling**

getUserProperty

This LOGICAL function retrieves the value of a dynamically defined property that you previously defined and set with the **setUserProperty** function. The values for these functions are always stored and returned in CHARACTER form:

Function getUserProperty:

Params: INPUT pcPropName AS CHARACTER.

Returns: CHARACTER property value.

Candidate for: **calling**

InstanceProperties

This CHARACTER property returns the list of instance properties for the Object. This list is normally hard-coded in the template for the object type as the xcInstance-Properties preprocessor, whose value is then assigned to InstanceProperties at startup of a static SmartObject. The value is also stored in the repository for use with dynamic SmartObjects. In earlier product releases, this property was important for the AppBuilder to use when it generated code for use in its adm-create-object procedure. With dynamic windows, the distinction between instance properties and other properties is not as significant, and the value may not be meaningful for most applications.

5.7 General-purpose methods

These are a few additional methods that are useful programming aids but that do not fall into any other particular category.

modifyListProperty

This internal procedure allows you to add or delete values from any object property that is a comma-separated list. There are a number of such properties, such as DisplayedFields and EnabledFields, described in this chapter, and their values may need to be changed programmatically at run time. This procedure simplifies the job of adding or removing a string from within the list:

Procedure modifyListProperty:

Params: INPUT phCaller AS HANDLE – handle of the object whose property is being

changed

INPUT pcMode AS CHARACTER -- 'ADD' or 'REMOVE'

INPUT pcListName AS CHARACTER -- name of the property

INPUT pcListvalue AS CHARACTER -- the value to add or remove

Candidate for: **calling**

signature

This CHARACTER function returns the signature, or calling sequence, of the named function or internal procedure in the format returned by the Progress GET-SIGNATURE method.

Because the entry point you need to access may in fact be implemented in a super procedure of the Object where you actually run it, this function saves you from having to search through the super procedure stack to locate it:

Function Signature:

Params: INPUT pcName AS CHARACTER -- function or procedure name.

Returns: CHARACTER: signature in Progress GET-SIGNATURE format.

Candidate for: **calling**

Using the Progress Dynamics Managers

The Progress Dynamics framework is more than just a way of defining a User Interface and logic to support a specific application. There is also a set of service procedures that support a wide range of application needs. These are called the Progress Dynamics Managers. You can learn more about the Managers in the [Progress Dynamics Administration Guide](#), the [Progress Dynamics Installation Guide](#), the [Getting Started with Progress Dynamics](#) manual, and the [Progress Dynamics Managers API Reference](#) manual. This chapter begins with a brief overview of the Managers and how they are constructed, and then provides some guidelines on how to use specific Manager API calls in your applications.

Much of what the Managers do is automatic and does not require any specific programming from you. Other parts of the Managers are supported by specific administration tools, such as the Security Manager, where you interact only indirectly with the Manager itself and its API.

This chapter focuses on the parts of each Manager's API that you can use explicitly in your application. It provides examples designed to give you a better understanding of how to use the Managers to provide support for special needs that they do not take care of automatically, without attempting to provide a comprehensive description of each Manager and all its procedures and functions.

This chapter contains the following sections:

- [Overview of the managers](#)
- [Manager architecture](#)
- [Using the Session Manager](#)
- [Using the Configuration File Manager](#)
- [Using the Connection Manager](#)
- [Using the Profile Manager](#)
- [Using the Localization Manager](#)
- [Using the Security Manager](#)
- [Using the General Manager](#)

6.1 Overview of the managers

Just what is a Manager, and what makes it special? The framework has many supporting procedures that contribute to your application's behavior, but only a few are called Managers. The Progress Dynamics Managers have all of the following key elements in common:

- Managers have both a client-side and a server-side component, which communicate with each other when necessary. This allows all database access to be restricted to the server-side procedure when the application is run in a distributed environment, and to the client-side to provide data and services within its session, to minimize overhead and network traffic caused by AppServer calls.
- Managers are designed to use a stateless AppServer connection, so that a client request can be handled by an already running Manager procedure in any server-side session, without binding that session to the client.
- Managers store needed context information in the repository, so that a series of requests from a given client session can be handled coherently, even if by many different sessions.
- Managers are responsible for data from different parts of the repository, and they present that data to the application when it is needed. Generally this means that among other things, data is cached in temp-tables on both sides of the AppServer connection, and kept in sync by the Manager. In this way the client can access repository data during the execution of the application without always going back to the server to get it.
- Managers have a standard design structure, which supports running efficiently between client and server. This structure is summarized below so that you can locate Manager code you may need to look at or specialize. For additional information on Manager design techniques, see [Chapter 7, "Creating a New Manager In Progress Dynamics."](#)
- Managers have an API of useful procedures and functions. Many of these functions are used internally by the Manager itself or called from other Managers, but some of them can also be called from application code. These calls are the focus of this chapter.
- Manager handles are global, which makes it easy to run a needed procedure in a Manager.

- Managers are generally pre-started on the AppServer and on the client by the Session Manager. The Configuration File section of the Progress Dynamics session management keeps information on all the managers, including which ones are needed by each Progress Dynamics session type. You can learn more about session types, registering managers, and how to associate managers with different sessions in the [Progress Dynamics Administration Guide](#).
- Managers function independently of the ADM, SmartObjects, the Progress Dynamics User Interface, and other specifics of the framework. They can be used from a variety of existing application code, to provide security and session management, as well as from new Progress Dynamics-specific application modules. This can help to integrate new Progress Dynamics development work into existing applications.

Figure 6–1 provides an overview of the manager architecture.

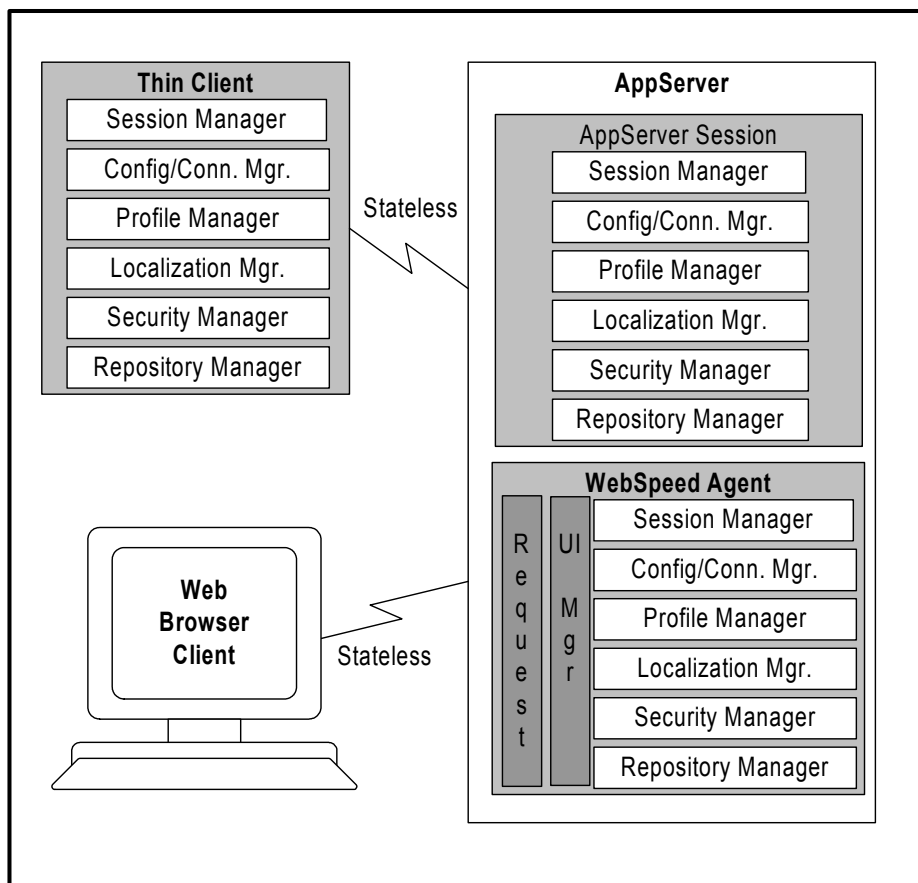


Figure 6–1: Progress Dynamics manager architecture

In Version 2.0 of Progress Dynamics, the framework supports a Web browser-based User Interface for an application. Part of this support involves generating a dynamic HTML-based UI for the browser by accessing the description of the UI through the Repository Manager. This is the same UI definition used to generate the Windows-based GUI. The Web support uses a User Interface Manager new to Version 2. In addition, a new Request Manager provides browser-based applications with access to the same Manager functions available to 4GL-based applications. The Managers available in Progress Dynamics include:

- [Session Manager](#)
- [Configuration File Manager](#)
- [Connection Manager](#)
- [Profile Manager](#)
- [Localization Manager](#)
- [Security Manager](#)
- [Repository Manager and Repository Design Manager](#)
- [General Manager](#)
- [Request Manager](#)
- [User Interface Manager](#)
- [Referential Integrity Manager](#)
- [Customization Manager](#)

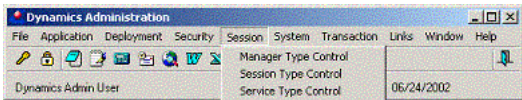
6.1.1 Session Manager

The Session Manager is responsible for starting and stopping Progress Dynamics processes. When you use the Dynamic Launcher in the AppBuilder to test a new dynamic window, it uses the `launchContainer` procedure in the Session Manager's API to start the window. When you use the `launch.i` or `dynlaunch.i` include file to invoke business logic procedures on the server, the Session Manager starts those procedures, runs entry points inside them and then stops them.

The Session Manager provides a host of other services as well. It handles the standard Progress Dynamics messaging calls described in the business logic chapters of the Developer’s Guide. It provides the context management used by all the other Managers to keep track of requests from each client session. It provides integration with the Progress Dynamics Help support. It provides an API through which you can access many system properties and define new properties of your own that your application needs.

6.1.2 Configuration File Manager

The Configuration File Manager keeps track of sessions, services, and other managers. All of its data is managed through the tools in the Session menu in the Progress Dynamics Administration window:

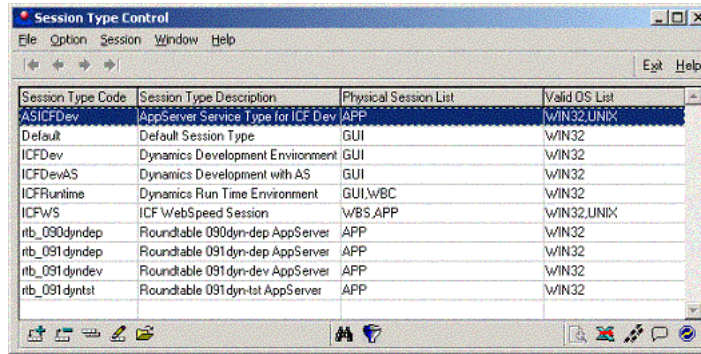


Use the Manager Type Control window to define new Manager Types and to identify which Managers to pre-start for which session types. The figure below shows all the Managers that come with the framework. Each has a Type Code that you can use as an identifier to run procedures in its API, and other information about the Manager:

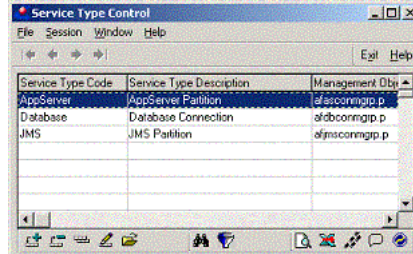
A screenshot of the 'Manager Type Control' window. It features a menu bar with File, Session, Window, and Help. Below the menu is a toolbar with navigation arrows and buttons for Exit and Help. The main area contains a table with five columns: Manager Type Code, Manager Type Name, System Owned, Write to Config, and Static. The table lists various managers such as CustomizationManager, GeneralManager, LocalizationManager, ProfileManager, RepositoryDesignManager, RepositoryManager, RequestManager, RIManager, SecurityManager, SessionManager, and UserInterfaceManager. At the bottom, there is another toolbar with icons for adding, deleting, and editing entries.

Manager Type Code	Manager Type Name	System Owned	Write to Config	Static
CustomizationManager	Customization Manager	YES	YES	NON
GeneralManager	General Manager	YES	YES	GM
LocalizationManager	Localization Manager	YES	YES	TM
ProfileManager	Profile Manager	YES	YES	PM
RepositoryDesignManager	Repository Design Manager	YES	YES	NON
RepositoryManager	Repository Manager	YES	YES	RM
RequestManager	Request Manager	YES	YES	NON
RIManager	Referential Integrity Manager	YES	YES	RI
SecurityManager	Security Manager	YES	YES	SEM
SessionManager	Session Manager	YES	YES	SM
UserInterfaceManager	User Interface Manager	YES	YES	NON

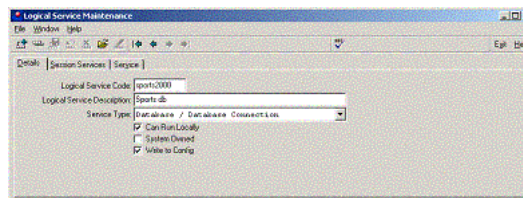
Use the Session Type Control window to define Session Types for the different ways in which your application or development environment needs to run. These built-in Session Types provide you with a variety of ways to start your client and server sessions, including a Development environment with or without AppServer and a runtime GUI environment:



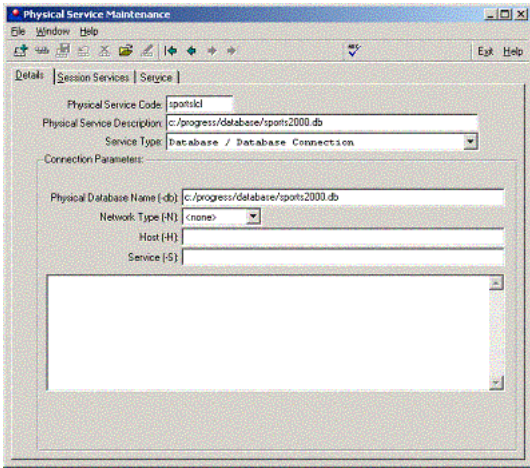
The Service Type Control window lets you define system services that require special code to start and manage them. Progress Dynamics comes with built-in service types for AppServers, for Databases, and for JMS (Java Message Service) partitions. Each Service Type defines a Management procedure and a Maintenance procedure that together provide the code to support the service, using a common API that is part of the Connection Manager:



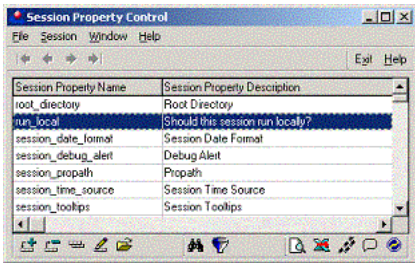
You can define any number of specific services for each different service type. For example, for each database your application needs to connect, you can define the startup parameters and other connection information for each service. In the Logical Service Maintenance window you define a logical name or Service Code for each service, that is, for each different database connection and each AppServer connection:



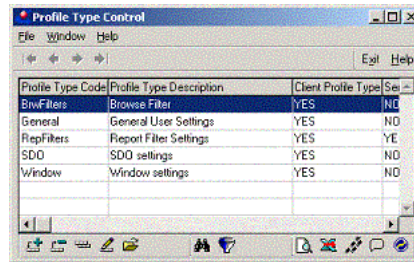
In the Physical Service Maintenance window, you can define the specific startup parameters for the service, including database connection parameters, host, service, and network names:



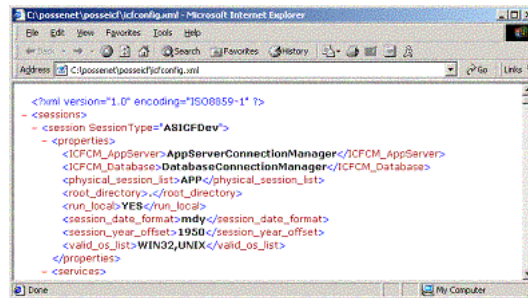
Each Session Type has properties that define characteristics of the session, such as whether it can run on the client only, its startup procedure and the Propath for the session. You can maintain these in the Session Property Control window and then assign properties and values to different Session Types:



You can define Profile Types in the Profile Type Control window. Profile Types are different categories of user profile information stored in the repository. Use them to help personalize the interface and behavior of an application for each user:



The data you maintain with these different utilities is kept in the repository, but much of it is also written out to an XML file, which allows each session to gain access to the configuration information it needs to start up. Here is a small excerpt from the standard Progress Dynamics XML representation of the configuration data, stored in **icfconfig.xml**:



6.1.3 Connection Manager

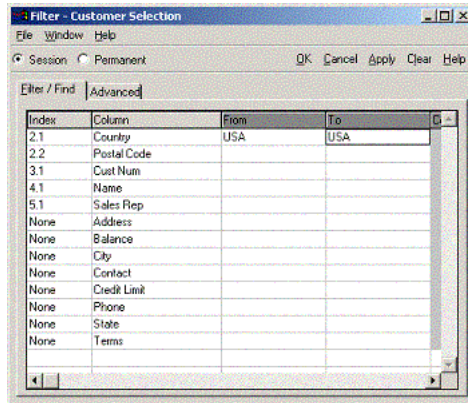
The Connection Manager defines a standard API used by each of the Service Types, which are in turn defined in the Configuration File Manager. There is a common set of support procedures for the Connection Manager, and then a custom set of procedures for each different service type, such as Database and AppServer. The Connection Manager is responsible for connecting and disconnecting services, and interpreting their specific startup parameters.

6.1.4 Profile Manager

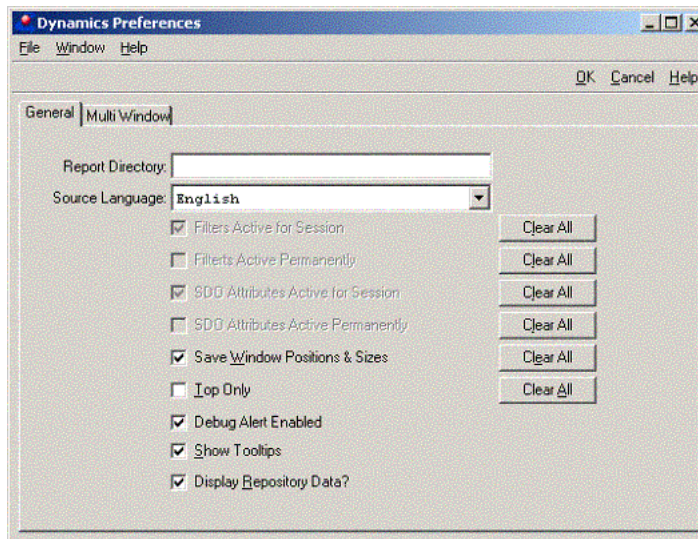
Progress Dynamics keeps track of a number of individual user preferences and settings in the repository as part of a profile for each user. These include:

- The position and size of each window a user opens, so that a window comes up in the same place and resized in the same way the next time the user opens it.

- Browser filter settings. Each time a user brings up a Progress Dynamics Filter window by pressing the Filter button on a toolbar, it provides the option of saving the filter settings, either for the session or permanently in the repository, so that the browse comes up with the same filter applied the next time it is opened by the same user:



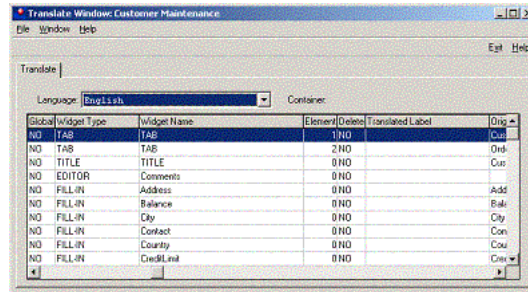
- Report filter settings.
- A variety of specific session settings. These can be accessed under the File menu in either the Progress Dynamics Development or Administration windows. Select Preferences, and the Dynamic Preferences window appears:



The Profile Manager caches profile information on the client and returns changes to it to the server, normally at the end of the session.

6.1.5 Localization Manager

The Localization Manager supports translation of text in the user interface, including window titles, folder tabs, labels, tooltips, messages, and menu items. It also supports a phrase translation glossary, so that commonly used phrases can be translated once for each language. Those translations can be accessed and used wherever they appear. You can bring up the basic translation mechanism by running any window that has the Translation menu option in its file menu, and selecting that option:



6.1.6 Security Manager

The Security Manager controls access to:

- Menus and menu items.
- Objects such as folder tabs and buttons that have named security tokens.
- Data fields.
- Specific database tables and records.

All of these types of security are maintained in the Security menu of the Administration window. The Security Manager also keeps track of application user and password definitions. There is a chapter on how to define users and then apply security restrictions for those users in the *Progress Dynamics Developer's Guide*.

6.1.7 Repository Manager and Repository Design Manager

The data used by all of the Managers is stored in the Progress Dynamics repository, but the Repository Manager is specifically in charge of all the data that defines the User Interface and all the objects that make up the application itself, including SDOs, Data Fields, Browsers, Viewers, Windows, Menus and Toolbars, and Folders. The repository maintains a class definition for each object, with default attributes for the Object. Each time you or the Object Generator creates an Object a record is stored for the master Object, along with attribute value records for each attribute that does not have the default value. Likewise, when you use an Object in a container window, its attributes can be changed or defined at that level, so there is an Instance record for each use of each Object, which has its own attribute value records for values defined or overridden at the instance level.

The Repository Manager has a complex API used not only to set these values and define all the records needed to define an application, but also to retrieve all the data for an entire window in a single call. The API then caches data for objects that have already been displayed on the client so that they come up even faster when displayed again. Because of the very different needs for design time and runtime, there are in fact two Repository Managers, one for a runtime environment, and the other, the Repository Design Manager, for the design time environment. The first has an API optimized for returning Object definitions to the procedures that realize them at runtime, in the most efficient way possible. The Design Manager has an API designed for use with the AppBuilder, the Object Generator, and other tools that populate the repository with data describing the Objects in the application.

The Object portion of the repository and the Repository Manager APIs are described in more detail in [Chapter 8, “Understanding the Object Tables In the Progress Dynamics Repository.”](#)

6.1.8 General Manager

As its name implies, the General Manager deals with a number of miscellaneous system support functions, including:

- Maintaining gapless sequences for database key values.
- Performing date and time conversions.
- Caching information on Entities, that is, database tables and their fields, and how they should be used to generate default SDOs, Browsers, and Viewers for those tables.
- Tracking database record information, including whether comment or audit records exist for a database record.
- Listing manipulation functions that retrieve values from lists (property lists for example) or insert them into lists.

6.1.9 Request Manager

The Request Manager supports requests that come in from a Web browser using the WebSpeed front-end to Progress Dynamics. The Request Manager plays two main roles in the processing of each request. The first is to evaluate the request and set up the environment in the correct and proper state for processing the actual request. The second role is the processing of the actual incoming web request by executing the procedures and managers according to the request type. This Manager mimics a large section of GUI client 4GL behavior. As new non-4GL client types are added to the framework, the Request Manager will provide a standard channel through which all client requests are handled and communicated to the standard framework support code on the server.

6.1.10 User Interface Manager

The User Interface Manager (UIM) supports the generation of the user interface by a Web browser, and in principle, by any other non-4GL client types as well. It delivers all the information required for the client to draw the UI and control the run-time execution of an application developed using the Progress Dynamics framework. This includes the following:

- User Interface (a screen, eg Order Entry)
- Application Data (for populating the user interface, eg order number)
- User Interface Changes/Behaviors (eg enable/disable field, highlight/unhighlight fields and popup messages, information, menu, toolbar, treeview, etc.)

The client is responsible for processing the output of the UIM and rendering the application UI. The Manager is also responsible for retrieving client request data. It is expected that a new version (sub class) of the UIM will be created to support each different client type in future releases, each sharing a common API, and that the UIM will eventually be used for **all** client types.

6.1.11 Referential Integrity Manager

The Referential Integrity (RI) Manager provides support for the schema triggers that are part of the application deployment process. It supports code for data versioning and the reuse of Object IDs when deployed Objects are deleted and then recreated. It is also a placeholder for a more complete referential integrity definition in the repository that will be implemented in a future release, when the relationship between tables and their keys will be expressed in repository data, replacing the need for trigger procedures to do this. There is no public API for the RI Manager at this time.

6.1.12 Customization Manager

The Customization Manager’s primary function is to provide facilities for interpreting a customization reference, so that customizations of many kinds can be defined in the repository and evaluated at runtime in such a way that all the relevant customizations to the User Interface and other application behavior are applied correctly for each individual user.

There is a section on customization and the repository tables that support it in [Chapter 8](#), “Understanding the Object Tables In the Progress Dynamics Repository.”

6.2 Manager architecture

The Progress Dynamics Managers have a common architecture that is worth studying in order to be able to locate and understand code that you may need to modify or extend in some way. See [Chapter 7](#), “Creating a New Manager In Progress Dynamics,” for more information on building your own Manager. This section describes the current Manager architecture, including:

- [Manager handles](#)
- [Organization of the manager code](#)
- [How the manager code runs on the server](#)

6.2.1 Manager handles

The principal Progress Dynamics Managers, which are the ones called most frequently from other framework code, have their procedure handles defined in an include file called **globals.i**, located in the `src/adm2` directory:

```
DEFINE NEW GLOBAL SHARED VARIABLE  gshAstraAppserver      AS HANDLE  NO-UNDO.
/* Handle to Application Server Partition */
DEFINE NEW GLOBAL SHARED VARIABLE  gshSessionManager      AS HANDLE  NO-UNDO.
/* Handle to Session Manager */
DEFINE NEW GLOBAL SHARED VARIABLE  gshSecurityManager     AS HANDLE  NO-UNDO.
/* Handle to Security Manager */
DEFINE NEW GLOBAL SHARED VARIABLE  gshProfileManager      AS HANDLE  NO-UNDO.
/* Handle to Profile Manager */
DEFINE NEW GLOBAL SHARED VARIABLE  gshRepositoryManager   AS HANDLE  NO-UNDO.
/* Handle to Repository Manager */
DEFINE NEW GLOBAL SHARED VARIABLE  gshTranslationManager  AS HANDLE  NO-UNDO.
/* Handle to Translation Manager */
DEFINE NEW GLOBAL SHARED VARIABLE  gshGenManager          AS HANDLE  NO-UNDO.
/* Handle to General Manager */
DEFINE NEW GLOBAL SHARED VARIABLE  gscSessionId           AS CHARACTER NO-UNDO.
/* Unique session id */
```

NOTE: This file is in the `src/adm2` directory not because it has anything specifically to do with the ADM2 code, which it does not, but simply to place it where many other include files used by SmartObjects are located. In older framework code, there are references to `{af/sup2/afglobals.i}`, which is equivalent. Note also that the include file contains several other application-specific handles not shown here because they are not relevant for general Progress Dynamics development.

These manager handles, along with the current Session ID, and the handle of the default AppServer session, called `AstraAppServer`, are available from every Progress Dynamics procedure, because `globals.i` is included in all the standard Progress Dynamics template procedures, and also in `src/adm2/smrtprop.i`, which makes it part of the SmartObject procedures as well. While global variables are generally to be avoided in applications, this basic set of handles is referenced so frequently that they are defined in this way for maximum efficiency. New Manager handles should be defined and accessed as properties in the Session Manager, as we discuss in the chapter on building your own Manager.

6.2.2 Organization of the manager code

The Managers are written to allow a single body of code to be compiled for use on both the Server and the Client, with the portions requiring database access segregated to the Server. For example, when using the Profile Manager, remember that a basic principle of the Managers is that there is a server version of the Manager that handles all repository database access, and a client version that other procedures in the client communicate with. The code that is common to both client and server versions is placed in an include file whose name is of the form **afxxxmgrp.i**, where **xxx** represents a three-letter abbreviation of the Manager name, such as:

- **con** — Connection Manager
- **gen** — General Manager
- **pro** — Profile Manager
- **sec** — Security Manager
- **ses** — Session Manager
- **trn** — Localization Manager

These include files are in the `af/app` directory. The `af` directory tree is where code related to supporting the framework itself is located ('af' stands for 'application framework'). The `app` subdirectory is where code goes that runs on the AppServer in a distributed environment.

Two other procedures include this file:

- The client version of the Manager is **af/sup2/afxxxc1ntp.p**. The sup2 directory is where general **support** code for the application framework goes. The 2 in sup2, as in other framework directories such as cod2, indicates that this is code specific to the Version 9, or ADM2 version of the framework, as opposed to earlier code with its origins in version 8.
- The server version of the Manager is **af/app/afxxsrvrp.p**. Again, this is in the af/app directory because it is classified as AppServer Code. Each of these two procedures basically just defines a preprocessor and then includes **af/app/afxxmgrp.i**. For example, the only code in **afproc1ntp.p** is this:

```
&global-define client-side yes
{af/app/afpromgrp.i}
```

And the only code in **afprosrvrp.p** is this:

```
&global-define server-side yes
{af/app/afpromgrp.i}
```

These two preprocessors, **client-side** and **server-side**, allow the procedures in the common include file to separate out blocks of code where needed that should be compiled only on the server side of the Manager or only on the client side, as we'll see in a moment.

The code that needs to be separated out is largely code that accesses the repository database. This code needs to be executed only on the server. Where the client needs access to the same data, it must run the same code on the AppServer.

The most efficient way to structure the Manager on the **server** is to have all of the code, including the procedures that access the database, together in one persistent procedure where it can all be pre-started and therefore running whenever it is needed. On the other hand, when the **client** code needs to access the same database procedures, the most efficient way to do that is to be able to make a single call to an external procedure (Progress **.r** file) on the Server, which takes INPUT parameters and returns whatever the result is as OUTPUT parameters. Making a single call like this to a stateless AppServer session does not bind the client to the server, and gets the results in a single AppServer call. By contrast, running an internal procedure inside a larger **.r** file requires making a call to establish the connection and run the **.r** file as a persistent procedure, then a separate call to run an internal procedure entry point inside it, and then another call to delete the server procedure. The client is bound to the AppServer session for the duration of this process.

To improve functionality, managers use a technique of isolating database access code in external .p's to be run from the client, and then including those in a version of the same code that runs on the server.

The common include file, `afpromngr.p.i`, has code in its Main Block that defines an internal procedure with the same name as each external procedure that contains server-side-only code. This set of internal procedure definitions is compiled into the procedure that includes the common code only if the server-side preprocessor is defined:

```
&IF DEFINED(server-side) <> 0 &THEN
  PROCEDURE afbldclcp:          {af/app/afbldclcp.p}      END PROCEDURE.
  PROCEDURE afchkpdexp:        {af/app/afchkpdexp.p}      END PROCEDURE.
  PROCEDURE afgetpdatp:        {af/app/afgetpdatp.p}      END PROCEDURE.
  PROCEDURE afsetpdatp:        {af/app/afsetpdatp.p}      END PROCEDURE.
  PROCEDURE afupdcadbp:        {af/app/afupdcadbp.p}      END PROCEDURE.
  PROCEDURE afdeisprop:        {af/app/afdeisprop.p}      END PROCEDURE.
&ENDIF
```

In this way the version of the Manager that runs on the server is a single large persistent procedure that incorporates all the database access code as well as the common code. This is an efficient way to operate because the Managers are pre-started and left running for the duration of the AppServer session. Note that this version of the Manager is also the one that runs if you are running locally, without an AppServer.

Each of the .p files is also compiled as a separate unit, so that its code can run from a client session. Because each must be a simple procedure call, not a call to an internal procedure inside some larger file, all the executable code for these data access procedures must be in the Main Block, the part of the file that is executed when the procedure runs, as shown in [Figure 6–2](#).

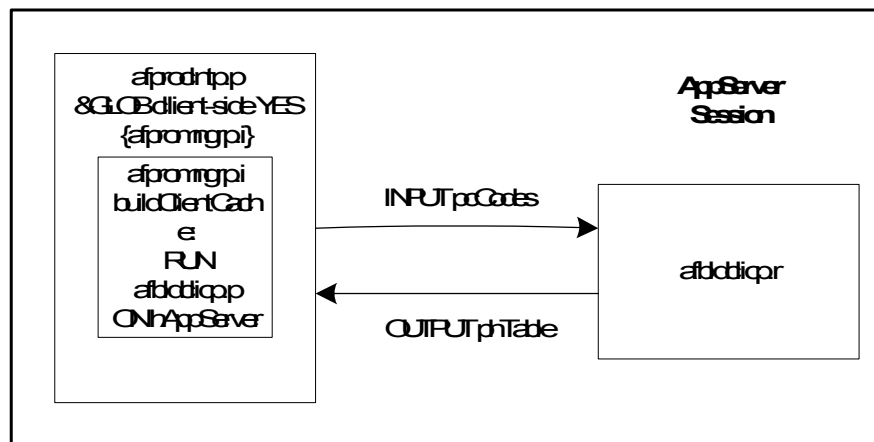


Figure 6–2: Executable code in main block

From a client session, the data access procedures are run as individual . r files on the AppServer, as shown in [Figure 6-3](#).

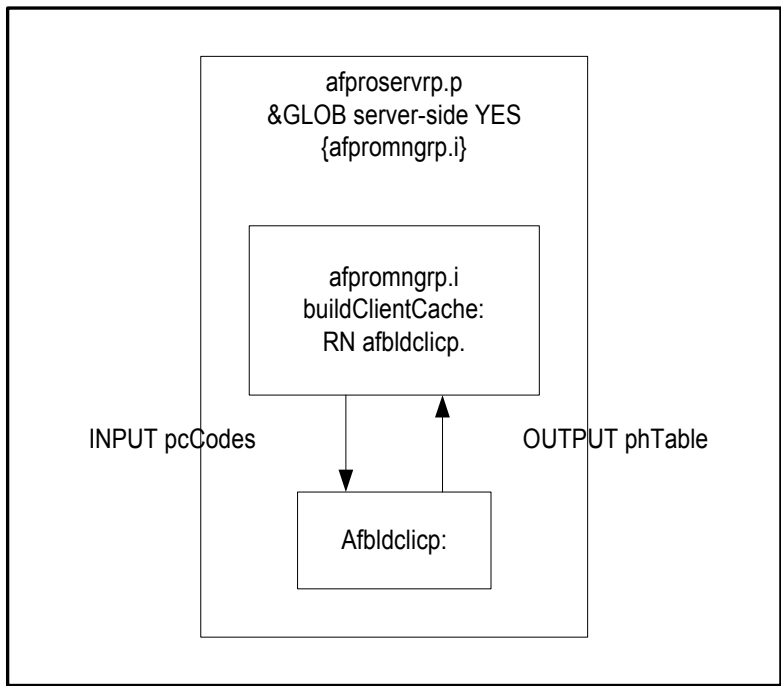


Figure 6-3: Data access file

Within the server session itself, or when there is no separate AppServer, the data access code is compiled right into the Manager itself.

For example, here is an excerpt of the code in the Main Block of the first support procedure for the Profile Manager, `afblclicp.p`, which builds the temp-table cache of profile values for a user and returns it to the session where that user is running:

```

/* ***** Main Block ***** */

DEFINE INPUT PARAMETER pcProfileTypeCodes          AS CHARACTER NO-UNDO.
DEFINE OUTPUT PARAMETER TABLE FOR ttProfileData.
...
/* loop around profile codes for profile type */
FOR EACH gsc_profile_code NO-LOCK
    WHERE gsc_profile_code.profile_type_obj =
        gsc_profile_type.profile_type_obj:
    FOR EACH gsm_profile_data NO-LOCK
        WHERE gsm_profile_data.USER_obj = dUserObj
            AND gsm_profile_data.profile_type_obj =
                gsc_profile_code.profile_type_obj
            AND gsm_profile_data.profile_code_obj =
                gsc_profile_code.profile_code_obj:

        CREATE ttProfileData.
        BUFFER-COPY gsm_profile_data TO ttProfileData
        ASSIGN cProfileTypeCode = gsc_profile_type.profile_type_code
               cProfileCode = gsc_profile_code.profile_code
               cAction = "NON":U.
    END. /* each profile data */

```

Note that the procedure can take whatever input and output parameters it needs. This one takes a list of profile type codes as input and returns the resulting temp-table of all values of those types for the current user. This is then kept on the client to be used during the session. In some cases you may find the parameters in the Definitions section of the procedure rather than the Main Block. This is just an organizational choice and doesn't affect how the procedure compiles.

Because this is code from one of the Managers itself, it makes direct references to the repository database tables. In code that you write in your application, including in custom Managers you create, you should use the available API for the Manager, along with general purpose routines such as **getEntityDescription**, to get data from the repository without direct reference to table and field names. The APIs is supported and kept compatible in the future; the specifics of the underlying table structure may be subject to change in future releases of the product.

Because this code is loading a temp-table of data for the client, it should not be run on the remote part of a distributed Manager, but only in the client part of a distributed Manager, or in a Manager that is being run without AppServer. Thus if you look at the Main Block of the common code include file, you see a reference that is found frequently in the Manager code:

```
IF NOT (SESSION:REMOTE OR SESSION:CLIENT-TYPE = "WEBSPEED":U) THEN
  RUN buildClientCache(INPUT "":U). /* load temp-table on client */
```

Checking the SESSION object tells the code whether it has been started on an AppServer session or WebSpeed agent. If this is not the case, then the code runs the internal procedure **buildClientCache**. This code is in the Main Block of the include file, so it is run as soon as the Manager is first executed on session startup.

Looking next at buildClientCache itself, you can see an example of how the **internal** versus **external** partitioning is used in the code:

```
PROCEDURE buildClientCache:

DEFINE INPUT PARAMETER  pcProfileTypeCodes          AS CHARACTER  NO-UNDO.

IF NOT (SESSION:REMOTE OR SESSION:CLIENT-TYPE = "WEBSPEED":U) THEN
DO:
  EMPTY TEMP-TABLE ttProfileData.

  &IF DEFINED(server-side) <> 0 &THEN
    RUN afbldclcp (INPUT pcProfileTypeCodes,
                  OUTPUT TABLE ttProfileData).
  &ELSE
    RUN af/app/afbldclcp.p ON gshAstraAppserver (INPUT pcProfileTypeCodes,
                  OUTPUT TABLE ttProfileData).
  &ENDIF

END.

END PROCEDURE.
```

First it checks the REMOTE parameter as before. Then it starts by emptying the Profile Data temp-table in case there is any leftover data in it. This could be the case if the session is restarted.

Next comes the code block of interest. What it says, in effect, is if this is the server-side version of the Manager, compile in a RUN statement to run the cache-loading procedure afbldclcp as an internal procedure within the Manager. Otherwise compile in a statement to run it as an external procedure on the default AppServer handle.

This is not a coding style to follow in new code that you write, because new features in the 4GL allow you to achieve the same flexibility without this structure. But within the existing Managers, it works effectively.

6.2.3 How the manager code runs on the server

This example shows one way the managers run code without binding the AppServer session, by packaging that code as individually callable external procedures. There is another technique as well that you see in the Manager code that also merits explaining.

Sometimes the code to be called from the client cannot effectively be packaged up into a separate .p file. In fact, a procedure on the client sometimes really needs to call itself on the server in order to return needed information. In other words, the client session needs to be able to run a procedure as if it were implemented locally. If the code for the procedure cannot be local, then the version of that procedure on the client needs to run a different version of itself on the server, where the code resides that, among other things, accesses the database.

In this case the server code is embedded inside an internal procedure that is part of the client-side Manager. However, the server-specific code is compiled out of the client Manager. So the entry point name exists on both sides of the server connection, but the code inside is different.

Let's look an example from the General Manager this time, in the source file af/app/rygenmgrp.i. The procedure **getEntityDescription** is a very useful general-purpose routine that can return the value of any field in the database, given the mnemonic or dump name of the table, the name of the field, and the unique Object ID key value for the record within the table.

Because getEntityDescription can return the value of any field, there is no practical way for the client to cache all the data it might need to look at. For this reason, if code runs the procedure on the client, the client code turns around and runs the same procedure name on the server. So first the procedure defines the parameters common to both client and server code:

```
PROCEDURE getEntityDescription:
```

```
DEFINE INPUT  PARAMETER pcEntityMnemonic      AS CHARACTER  NO-UNDO.  
DEFINE INPUT  PARAMETER pdEntityObj           AS DECIMAL    NO-UNDO.  
DEFINE INPUT  PARAMETER pcFieldName           AS CHARACTER  NO-UNDO.  
DEFINE OUTPUT PARAMETER pcEntityDescriptor    AS CHARACTER  NO-UNDO.
```

Then it uses the **server-side** preprocessor to compile in a call to the server if server-side is not defined. The include file `dynlaunch.i` that drives the call is the most efficient way to make a call to an internal procedure inside a server-side program, and return values from that procedure. Its named include file arguments include the name of the procedure or Manager to run or access on the server; the name of the internal procedure to run; and a set of three arguments for each parameter to the internal procedure that specify the parameter's INPUT, OUTPUT, or INPUT-OUTPUT mode, the parameter name, and its datatype. The `dynlaunch.i` include file turns this into a dynamic reference to the internal procedure that does all its work with a single AppServer hit:

```
&IF DEFINED(server-side) = 0 &THEN
{
  dynlaunch.i &PLIP           = ''GeneralManager''
              &iProc          = ''getEntityDescription''
              &compileStaticCall = NO
              &mode1 = INPUT  &parm1 = pcEntityMnemonic
                              &dataType1 = CHARACTER
              &mode2 = INPUT  &parm2 = pdEntityObj
                              &dataType2 = DECIMAL
              &mode3 = INPUT  &parm3 = pcFieldName
                              &dataType3 = CHARACTER
              &mode4 = OUTPUT &parm4 = pcEntityDescriptor
                              &dataType4 = CHARACTER
}
  IF ERROR-STATUS:ERROR OR RETURN-VALUE <> ''':U THEN RETURN ERROR
RETURN-VALUE.
```

This is different from the previous example, where the server code is run as a separate procedure when needed and then goes away. In this case, the procedure you want to run is inside the server-side General Manager that is already up and running, so you do not want to run a new copy of it. The `dynlaunch.i` file works whether the server-side program is already running as a persistent procedure or Manager, or whether it needs to be started for the request and then terminated when it returns.

This is the end of the code that gets run if the procedure is compiled for the client. Following this is an &ELSE block that is compiled if the code is compiled for the server. This is the guts of getEntityDescription, the code that does all the work:

```
&ELSE
  DEFINE VARIABLE cQueryPrepareString          AS CHARACTER    NO-UNDO.
  DEFINE VARIABLE cTableObjectName             AS CHARACTER    NO-UNDO.
  DEFINE VARIABLE cTableBase                   AS CHARACTER    NO-UNDO.
  DEFINE VARIABLE hQuery                       AS HANDLE        NO-UNDO.
  DEFINE VARIABLE hBuffer                      AS HANDLE        NO-UNDO.
  DEFINE VARIABLE hDescriptionField             AS HANDLE        NO-UNDO.
  DEFINE VARIABLE hCurrentField                AS HANDLE        NO-UNDO.
  DEFINE VARIABLE iFieldLoop                   AS INTEGER        NO-UNDO.

  FIND ttEntityMnemonic WHERE
    ttEntityMnemonic.entity_mnemonic = pcEntityMnemonic
  NO-ERROR.

  ...etc.
```

In this way code in a single file (the common code include file in this case) can be compiled two different ways to execute in a coordinated fashion between client and server.

6.3 Using the Session Manager

The Progress Dynamics Session Manager handles a wide range of application functions. For a complete description of each Manager, see the [Progress Dynamics Managers API Reference](#) manual. Major areas covered by the Session Manager include the support for context management, setting and getting properties, launching procedures, error and messaging management, help management, and email and session logon support. This section includes:

- [Context management](#)
- [Property management](#)
- [Managing procedures and containers](#)

6.3.1 Context management

The Session Manager keeps track of information that must span the client/server divide, i.e. information that must be available to business logic regardless of whether it is running client or server side. Business logic should not have to be concerned with where it is being run and should function exactly the same in any environment.

To facilitate this the repository database has a context database table, shown in [Figure 6–4](#), called `gsm_server_context`. This table is a generic table used to store context information of any kind between stateless AppServer connections.

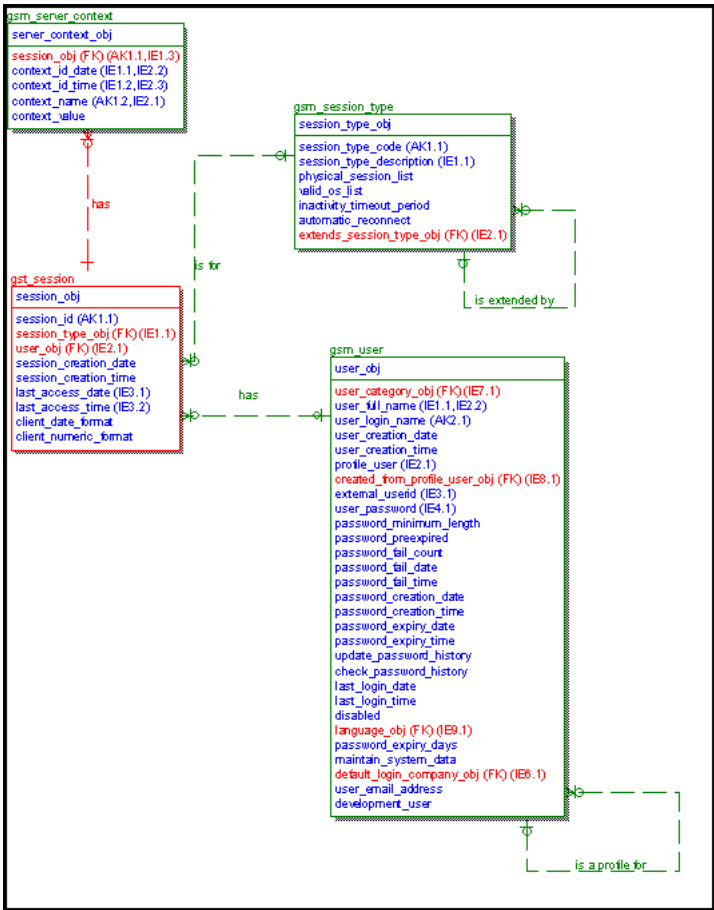


Figure 6–4: Context database table definitions

As you can see from the table definition, all context records are identified by a `session_obj`, which joins to a session record in a separate `gsm_session` table. This is based on the **session:server-connection-id** which provides a unique code for the current client connection. The ID is maintained from the time of connection to the AppServer until a disconnect statement. The value is also available to the client via the **client-connection-id** attribute of the AppServer object handle if an AppServer is in use.

For code portability when not using AppServer, code always refers to the global shared variable **gscSessionId**, which you saw earlier in `globals.i`, rather than the `session:server-connection-id`. This is set during application start-up on the client. On the server it is set during session activation and reset on deactivation. The type of information that is stored by the framework itself in the context table includes user information, security information, etc.

The context table stores a date and time stamp for each value, indicating when it was last set and the name and value of the information itself. The `context_name` field is therefore just a property name.

Apart from a `deleteContext` call to destroy the context information for a user, the API that provides applications access to the context table is a function of property management.

6.3.2 Property management

The Session Manager on the client maintains a temp-table of property values representing context information needed by the client. For values stored in the temp-table, no server call is necessary to retrieve a value. This temp-table is initialized at session start-up with values for the standard client properties such as User ID and login company. These values are then passed to the server if they are also needed there. Only properties that are solely maintained on the client can be cached locally on the client in this way, even though their values can be read server-side. If it is possible for code to modify the property value on the server, then the client temp-table cannot have the property value, and the client must always redirect the request for the property to the server.

The Session Manager is also used for client-side properties that do not need to be reflected on the server. For these properties, the value is only managed on the client. This can be a useful and efficient mechanism for temporary storage of information between program calls, as the Session Manager is always initiated at the start of every session and runs in every client session.

If there is no AppServer connection, then the server version of the Session Manager is run on the client. In this case, all properties are cached in the temp-table and the context database table is not used.

In addition to the many standard properties already defined, application code can define and store any additional properties just by passing them to the Session Manager. There are two simple calls to do this:

- **getPropertyList (INPUT pcPropertyList AS CHARACTER, INPUT plSessionOnly AS LOGICAL) RETURNS CHARACTER** — This function retrieves the specified property values from the local temp-table if they're available. If they are not available in the temp-table, then the function makes a call to the AppServer to check the context database table. If the session only flag is set to YES and the request is made on the client side, then the context database table is not checked. The input property list can also be a comma-separated list of property names. The return value is a CHR(3)-delimited list of values. It's advisable to get (and set) as many values together as possible if they need to be retrieved from the server, to minimize AppServer traffic:

```
DYNAMIC-FUNCTION("getPropertyList":U IN gshSessionManager
  ( INPUT pcPropertyList,
    INPUT plSessionOnly ).
```

- **SetPropertyList (INPUT pcPropertyList AS CHARACTER, INPUT pcPropertyValues AS CHARACTER, INPUT plSessionOnly AS LOGICAL) RETURNS LOGICAL** — This function sets the specified property values in the local temp-table if it is run on the client side. If it is invoked on the client and the Session-Only flag is set to NO, then the property is also set in the context database table for use by the remote session:

```
DYNAMIC-FUNCTION("setPropertyList":U IN gshSessionManager
  ( INPUT pcPropertyList,
    INPUT pcPropertyValues,
    INPUT plSessionOnly ).
```

Here are a few things to keep in mind when you use these functions:

- As noted, you can define a new property just by using it in a call to setPropertyList. For this reason, it is important to check for spelling errors in calls to the function, because an attempt to set a property using the wrong name silently creates a new property with the wrong name and set its value instead.

- Always consider which property values are needed on the client and which are needed on the server. You can use the property functions and the context table to store any information that may be needed elsewhere in the application. If it is needed only by other client-side code, then always set the Session-Only flag to YES. In this way, the value is stored only in the client temp-table. If business logic on the server could use or set the value, then you must set the Session-Only flag to NO and consider that every time you set the property, or retrieve it from the client, you incur an AppServer hit.
- To set a property value to null, use an empty string (""), not the Progress unknown value (?).

6.3.3 Managing procedures and containers

The central purpose of the Session Manager is to control all the running objects in the system, including the other Managers, application components, and supporting business logic procedures. Much of this work happens automatically, based on information defined in many places, such as:

- The Configuration File and Connection Managers, which define which Managers are started for each Session Type, what the startup options are, etc.
- Menus and Toolbars, which launch other objects, both static and dynamic, on Choose of a button or menu item, as defined in the Toolbar and Menu Designer.
- Definitions of custom super procedures for objects in the object Property sheets or the Repository Maintenance tool, which cause the Session Manager to start these supporting objects and make them super procedures.
- Application code that starts and makes requests of business logic procedures.

This section describes the procedures in the Session Manager's API that you can use in your applications to control the running of other application objects, both static and dynamic.

Using **launchProcedure** and **launch.i**

Call **launchProcedure** to start any persistent procedure your application needs:

```
RUN launchProcedure IN gshSessionManager
( INPUT pcPhysicalName,
  INPUT p1OnceOnly,
  INPUT pcOnAppserver,
  INPUT pcAppserverPartition
  INPUT p1RunPermanent,
  OUTPUT phProcedureHandle )
```

See the [Progress Dynamics Managers API Reference](#) manual for more information on this call. Your code should normally not run **launchProcedure** directly. Instead, the **launch.i** and **dynlaunch.i** include files act as a wrapper for calls to **launchProcedure**, allowing you to name both an external procedure to run and also the name of an entry point inside it, along with any **INPUT** and **OUTPUT** parameters to the internal procedure and other details that help encapsulate the calls. **dynlaunch.i** is the generally preferred include file to use if you do not require access to the server-side procedure after your request completes. It deals with the entire request in a single **AppServer** call.

Using **launchContainer**

You have already seen the **launchContainer** call in action many times. This is what the Dynamic Launcher uses, to start a dynamic container that you need to test. It is what the **AppBuilder** runs if you press the runner icon for an open container. It is also used from the **Toolbar** and **Menu** support code in **toolbar.p** to start containers that are run on **Choose of a menu item** or **toolbar button**.

You can use **launchContainer** within your own application code as well. One important use is to help you integrate existing menus or windows in an older application with new modules that you're developing in Progress Dynamics. Buttons or menu items added to your current application can launch dynamic containers built in Progress Dynamics, so that both can become part of a single integrated application. This section shows an example of how to do this.

First look at the launchContainer call itself:

```

RUN launchContainer IN gshSessionManager
( INPUT pcObjectFileName,
  INPUT pcPhysicalName,
  INPUT pcLogicalName,
  INPUT plOnceOnly,
  INPUT pcInstanceAttributes,
  INPUT pcChildDataKey,
  INPUT pcRunAttribute,
  INPUT pcContainerMode,
  INPUT phParentWindow,
  INPUT phParentProcedure,
  INPUT phObjectProcedure,
  OUTPUT phProcedureHandle,
  OUTPUT pcProcedureType).

```

The launchContainer call takes the following parameters:

- **INPUT pcObjectFileName (CHARACTER)** — The object filename used when you do not know the physical and logical names. Use this parameter to launch a physical object (static procedure) rather than a dynamic object.
- **INPUT pcPhysicalName(CHARACTER)** — The physical object name (with path and extension) if known. The difference between this and the Object File Name can be confusing. This is not an individual static procedure to support a single window. Rather it is the single driver procedure that instantiates all windows at runtime. For standard dynamic windows, always set this parameter to the procedure **ry/uiib/rydyncontw.w**, which is the standard framework procedure used to create any dynamic container window. For Progress Dynamics Treeview windows, specify the file **ry/uiib/rydyntreew.w**.
- **INPUT pcLogicalName (CHARACTER)** — The logical object name of the container. This is the name you gave to the container window when you created it.
- **INPUT plOnceOnly (LOGICAL)** — This logical flag indicates whether there may be more than one instance of this procedure running concurrently. If it is FALSE, then the Session manager checks whether there is already a running instance of the object in memory, and if so, reuses that. There is a Progress Dynamics ADM2 container property called **MultiInstanceActivated** that is set to reflect the user preference for multiple windows. Users can set this in the **File→Preferences** window from the Progress Dynamics Development or Administration windows. This choice is overridden if the object does not support multiple instances as identified by the **MultiInstanceSupported** container property. If this is FALSE, then only one instance will ever be run, regardless of the OnceOnly value passed in. This would be the general case for Object Controllers and filter windows.

- **INPUT pcInstanceAttributes (CHARACTER)** — The SmartObject instance attributes and values to pass to container, if any. If an instance attribute list is passed in, the list must be in the same standard ADM2 format as returned to the function **instancePropertyList**, with CHR(3) between entries and CHR(4) between the property name and its value within each entry. These attributes are then set in the container prior to running initializeobject in the container. The Session Manager procedure **setAttributesInObject** is used to set the attributes.
- **INPUT pcChildDataKey (CHARACTER)** — The child data key if applicable. When you invoke a Maintenance window from an Object Controller browse window, for example, the browse window passes in the key value for the selected row the Maintenance window should start on. This is the child data key. If a child data key is passed in and the OnceOnly flag is NO to enable multiple instances, then a check is made for an existing running instance with the same data key. If one is found, then regardless of the OnceOnly flag, this instance is just brought to the top, since it is invalid to have multiple instances of a container for the same data key. Also, if the object itself does not support multiple instances, the data key passed in is set to blank. **ChildDataKey** is a SmartObject property you can query in order to set this parameter correctly, when this is appropriate.
- **INPUT pcRunAttribute (CHARACTER)** — The run attribute if this is required to pass into the container. This could be specified in the Toolbar and Menu Designer, for example, for the action on a menu item. This run attribute is different from the SmartObject attributes and is not normally used.
- **INPUT pcContainerMode (CHARACTER)** — This is the initial operating mode for the container: view, or update, for example. Pass this in as blank to get the default.
- **INPUT phParentWindow (HANDLE)** — The parent (caller) window handle if known. This could be the handle of the window invoking the new container, if you want that to be its parent.
- **INPUT phParentProcedure (HANDLE)** — The parent (caller) procedure handle if known.
- **INPUT phObjectProcedure(HANDLE)** — The parent (caller) object handle if known. The handle of the object linked to the toolbar that contained the menu may also be passed in, e.g. the handle of a browser or viewer. If you use launchContainer to invoke new windows from old ones, where there is no meaningful parent object, this parameter can be passed in as the unknown value.

- **OUTPUT `phProcedureHandle (HANDLE)`** — The procedure handle of the container being run is returned in this Output parameter.
- **OUTPUT `pcProcedureType (CHARACTER)`** — The procedure type of the container being run is returned in this Output parameter. This is normally the string ICF.

Integrating old and new applications

This section contains an example of how to use the `launchContainer` procedure and the Session Manager to combine existing application modules and new Progress Dynamics modules in a single application. Figure 6–5 shows a simple application window that represents any window or menu in your current application. This one is a static SmartWindow, but any procedure window or other application procedure will do.

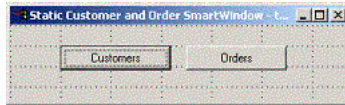


Figure 6–5: Sample application window

In the example, you launch Progress Dynamics containers from this window, and then later launch the window itself from a Progress Dynamics menu.

Creating a sample application window

To create the example, follow these steps:

- 1 ♦ Build a new Static SmartWindow in the AppBuilder.
- 2 ♦ Drop two buttons onto it and label them **Customers** and **Orders**. The example uses some of the objects from the tutorial application in the [Progress Dynamics Installation Guide](#) and in the [Getting Started with Progress Dynamics](#) book. You can use those or build any similar objects of your own to complete the example.
- 3 ♦ If you use the Progress Dynamics version of the AppBuilder to create a new Static SmartWindow, it will refer to the include file `globals.i` already in the Definition section, as every Progress Dynamics procedure does. Otherwise you need to add it to your procedure. Also define two variables at the scope of the procedure to hold the handles of the Customer and Order windows to launch:

```
{src/adm2/globals.i}
```

```
DEFINE VARIABLE ghCustomerWin AS HANDLE NO-UNDO.
DEFINE VARIABLE ghOrderWin AS HANDLE NO-UNDO.
```

- 4 ♦ Create an internal procedure called **launchDynWindow**. This takes as an Input parameter the logical name of the window to launch, and returns its procedure handle as an Output parameter. The parameters passed to launchContainer should be clear from the previous explanation of the procedure. As noted, the first argument (ObjectFileName) is not used; instead the name of the dynamic container procedure is passed in, along with the logical name of your container. The ParentWindow argument is the handle of the window in the sample procedure, which in our case is **wiWin**. The ParentProcedure parameter is its procedure handle:

```

PROCEDURE launchDynWindow:
/*-----
Purpose:      Uses the launchContainer procedure in the Session Manager
               to run a dynamic window from this static window.
Parameters:   INPUT pcWindowName AS CHARACTER -- the logical window name,
               OUTPUT phProcedureHandle AS HANDLE -- the new window handle
Notes:
-----*/
DEFINE INPUT  PARAMETER pcWindowName      AS CHARACTER  NO-UNDO.
DEFINE OUTPUT PARAMETER phProcedureHandle AS HANDLE     NO-UNDO.

DEFINE VARIABLE cProcedureType           AS CHARACTER  NO-UNDO.

RUN launchContainer IN gshSessionManager
( INPUT /* pcObjectFileName */      "",
  INPUT /* pcPhysicalFileName */    "ry/uib/rydyncontw.w",
  INPUT /* pcLogicalName */         pcWindowName,
  INPUT /* plOnceOnly */            YES,
  INPUT /* pcInstanceAttributes */  "",
  INPUT /* pcChildDataKey */        "",
  INPUT /* pcRunAttribute */        "",
  INPUT /* pcContainerMode */       "view",
  INPUT /* phParentWindow */        wiWin,
  INPUT /* phParentProcedure */     THIS-PROCEDURE,
  INPUT /* phObjectProcedure */     ?,
  OUTPUT phProcedureHandle,
  OUTPUT cProcedureType             /* ICF */ ) .
END PROCEDURE.

```

- 5 ♦ Create two trigger blocks. The first is for the Choose trigger on the Customer button. Use as the INPUT parameter the name of one of the logical container objects from the tutorial application or any others that you have built:

```

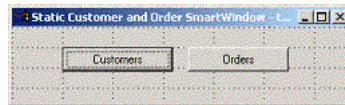
DO:
    RUN launchDynWindow (INPUT 'custbrowsewin',
                        OUTPUT ghCustomerWin).
END.

```

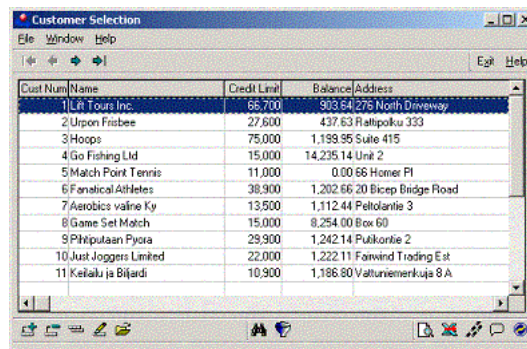

This second block of code is for the Choose trigger on the Order button:

```
DO:
  RUN launchDynWindow (INPUT 'orderbrowsewin',
                      OUTPUT ghOrderWin).
END.
```

- 6 ♦ This is all you need to do to integrate your static window in with the dynamic containers. Run it to see:



- 7 ♦ Press the Customers button to launch the dynamic Customer browse window:

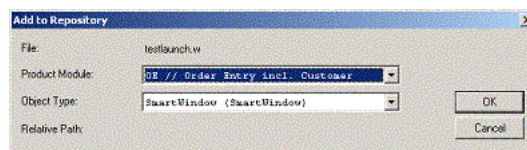


- 8 ♦ Save your window as **testlaunch.w**.

Registering a static window in the Progress Dynamics repository

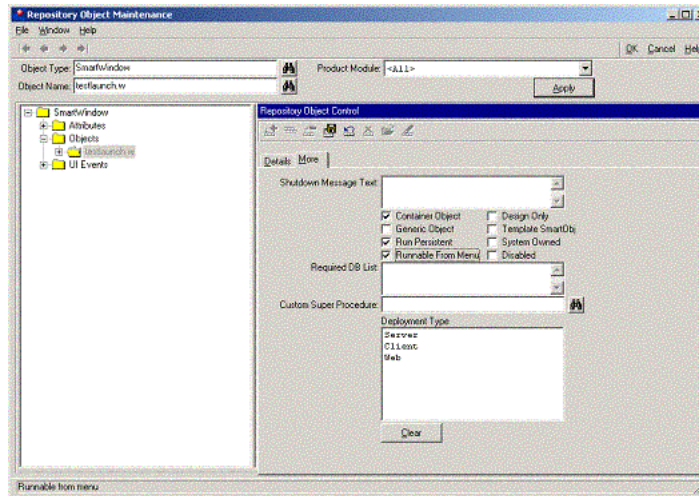
The next example integrates the applications in the opposite direction, invoking this static window from a Progress Dynamics Menu Controller window. To register your static window in the Progress Dynamics repository, follow these steps:

- 1 ♦ From the AppBuilder's File menu, select **Register in Repository**. The Register in Repository window appears:



- 2 ♦ Select a Product Module, and an Object Type of SmartWindow.
- 3 ♦ Click OK to register your window. Now you can reference it within the Progress Dynamics development tools.

There are two object attributes that you may need to set differently from the default settings you get when you add the object to the repository. Specifically, in order for the Toolbar and Menu Designer to know that this is a container object, and that it can be run from a menu, you need to set these attributes: Container Object, and Runnable From Menu:

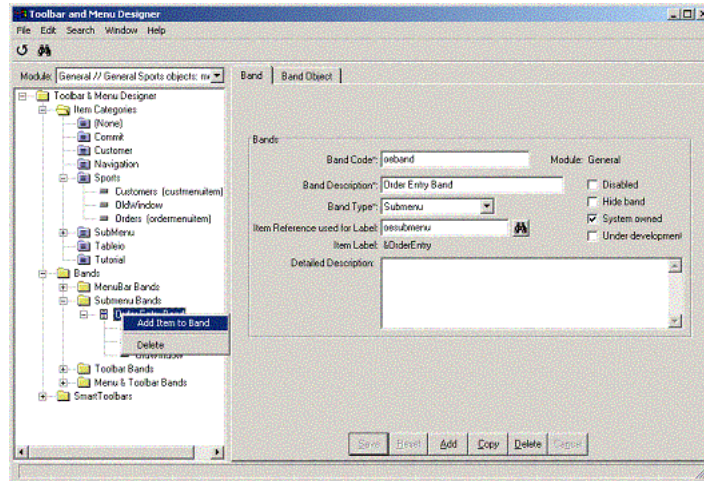


- 4 ♦ To do this, open the Repository Maintenance Tool, enter **testlaunch.w** as the Object Name and click Apply
- 5 ♦ Expand the SmartWindow node and the Objects node to show **testlaunch.w**.
- 6 ♦ Select the More tab in the maintenance folder and check Container Object and Runnable From Menu on. Verify that the Run Persistent attribute is also checked on and save your changes.

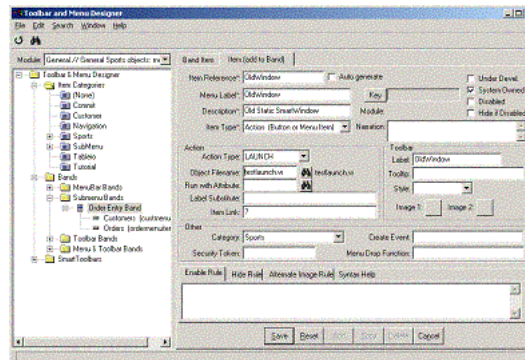
Adding a static window to the Dynamic Menu Controller window

Here, you can use the **oemenuwin** menu that is part of the tutorial application, but, any menu will do. To add the static window to a dynamic Menu Controller window, follow these steps:

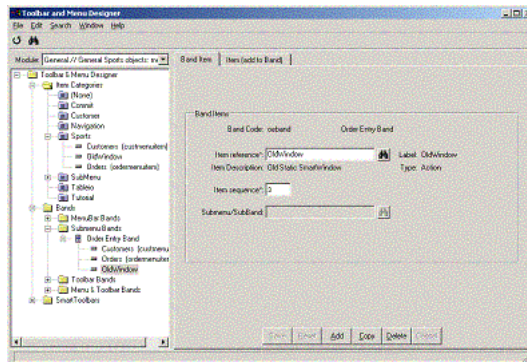
- 1 ♦ Open the Toolbar and Menu Designer:



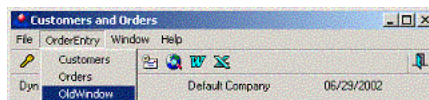
- 2 ♦ Enter **General** as the Module. This filters down the list of Items and Bands.
- 3 ♦ Expand the **Item Categories** node, and the **Sports** node. There are two existing items: for Customers and Orders. Add another item here called **OldWindow**. Alternatively, you can create the item and add it to the proper band in a single operation, by expanding the Bands node, then the Submenu bands node, and right clicking on the Order Entry Band to bring up the pop-up menu, from which you select Add Item to Band.
- 4 ♦ Either way, you define the item as shown below:



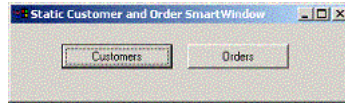
- 5 ♦ Enter **OldWindow** as the Item Reference and Menu Label, and enter a Description.
- 6 ♦ Select **Launch as the Action Type**, and **testlaunch.w** as the Object Filename. If you create the item under the Sports Item Category before adding the item to the band, then the Item Category will be filled in for you. Otherwise, if you created the new item directly under the band, enter Sports for the Category. Save this new item.
- 7 ♦ If you created the Item under the Sports Category, you must now expand the Bands and Submenu bands and add the OldWindow Item to the Order Entry Band. Otherwise you simply select the Band Item tab and set the Item sequence to 3, to add it to the menu after the Customers and Orders items:



- 8 ♦ Exit the Toolbar and Menu Designer. Because the oemenuwin Menu Controller uses the Order Entry band, the item you just added to it appears automatically when you run that menu window. Test your change by running oemenuwin from the Dynamic Launcher. Click the **Destroy ADM Super Procedures** toggle on to clear the client cache of menu items. Your new item appears in the Order Entry menu:



- 9 ♦ Select OldWindow, and the static SmartWindow appears:



So now you see how to make changes to both an existing static application, with or without SmartObjects, and to a Progress Dynamics application, to be able to integrate old and new modules in a single session.

Using launchExternalProcess

In addition to launching dynamic objects and 4GL windows using the Session Manager, you can also launch external Windows processes using the **launchExternalProcess** call:

```
RUN launchExternalProcess IN gshSessionManager
  ( INPUT pcCommandLine,
    INPUT pcCurrentDirectory,
    INPUT piShowWindow,
    OUTPUT piResult ).
```

The call takes these parameters:

- **INPUT pcCommandLine (CHARACTER)** — The command line to use, e.g. **notepad.exe <filename>**.
- **INPUT pcCurrentDirectory (CHARACTER)** — The default directory for the process.
- **INPUT piShowWindow (INTEGER)** — The show window flag, which can have one of these values:
 - 0) — Hidden
 - 1) — Normal
 - 2) — Minimized
 - 3) — Maximized
- **OUTPUT piResult (INTEGER)** — The result of the attempted launch action; can be either 0, which indicates failure, or a non-zero value, which is the Windows handle of new process.

The launchExternalProcess call uses the CreateProcessA API function in Windows NT or 2000.

To see an example of how this works, follow these steps:

- 1 ♦ Add another button to your **testlaunch.w** window, labeled **Notepad**.
- 2 ♦ Define this Choose trigger for it:

```

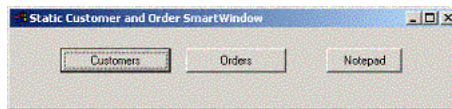
DO:
  DEFINE VARIABLE lResult AS LOGICAL      NO-UNDO.

  RUN launchExternalProcess IN gshSessionManager
    (INPUT "Notepad.exe " + THIS-PROCEDURE:FILE-NAME,
     INPUT "",                /* default directory */
     INPUT 1,                 /* window state -- normal */
     OUTPUT lResult).

END.

```

- 3 ♦ Re-run your window:



- 4 ♦ Choose the Notepad button to bring up the MS Windows Notepad program to edit the current procedure. (In this case, because it's being invoked from the AppBuilder, you see the temporary file the AppBuilder creates for the procedure you're working on:

```

DO: SP: form: bno
$SCOPE object-name testlaunch.w
DEFINE VARIABLE tv_this_object_name AS CHARACTER INITIAL "{&object-name}"!U NO-UNDO.
$SCOPE object-version 000000

/* Parameters definitions --- */
/* Local Variable Definitions --- */
/* object identifying preprocessor */
$gloob start2-staticSmartwindow yes

{src/adk2/globals.t}
  DEFINE VARIABLE ghCustomerwin AS HANDLE NO-UNDO.
  DEFINE VARIABLE ghOrderwin AS HANDLE NO-UNDO.

/* _UIB-CODE-BLOCK-END */
$ANALYZE-RESUME

$ANALYZE-SUSPEND _UIB-PREPROCESSOR-BLOCK
/* ***** preprocessor definitions ***** */
$SCOPE-define PROCEDURE-TYPE SmartWindow
$SCOPE-define SE-NAME PO
$SCOPE-define ADM-CONTAINER WINDOW
$SCOPE-define ADM-SUPPORTED-LINKS
Data-Target,Data-Source,Page-Target,Update-Source,Update-Target,Filter-target,Filter-Source
/* Name of first frame and/or browse and/or first query */
$SCOPE-define FRAME-NAME frmMain
/* standard list definitions
$SCOPE-define ENABLED-OBJECTS buCustomer buOrders buNotepad

```

6.4 Using the Configuration File Manager

The Configuration File Manager is different in its construction from the other Progress Dynamics Managers. Because it starts every other Manager in the framework, it is the first procedure run by Progress Dynamics on startup. Its principal procedure file is `af/app/afxm1cfgp.p`, and it does not use the techniques described in the “[Manager architecture](#)” section of this chapter. Because the Configuration File Manager is so fundamental to the framework, its handle is not stored as a global variable as with the other basic Managers. Instead, it makes itself a super procedure of the Progress session it is run in, so that any procedure in an application can run one of its entry points as if it was in the application itself (IN THIS-PROCEDURE, in other words), and it can be found in the session super procedure.

The “[Overview of the managers](#)” section of this chapter summarizes the utilities that you normally use to define and maintain manager types, session types, session properties, and other data in the domain of the Configuration File Manager. More information on using those utilities is in the *Progress Dynamics Administration Guide*. This section notes the API calls in this Manager that you are most likely to want to use in your code.

6.4.1 Using the `isICFRunning` function

At some points in your code, especially if your application is a mix of new and old code and needs to operate differently depending on whether Progress Dynamics is running, there is a function to tell you that called **`isICFRunning`**. Remember that ICF is the generic name for the Progress Dynamics product.

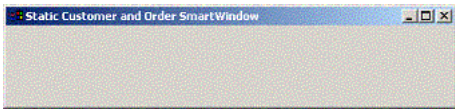
For example, in the `testlaunch.w` file you built earlier in this chapter, you might want to leave out the buttons that run Progress Dynamics-dependent components if the framework itself is not running. You can do this by adding this code to your startup in the Main Block of `testlaunch.w` in this example:

```
DEFINE VARIABLE lDynamics AS LOGICAL    NO-UNDO.

lDynamics = DYNAMIC-FUNCTION('isICFRunning') NO-ERROR.
IF lDynamics NE TRUE THEN
    ASSIGN buCustomer:HIDDEN = YES
           buOrders:HIDDEN = YES
           buNotepad:HIDDEN = YES.
```

Note that it is important to invoke the function `NO-ERROR`, since if the Configuration File Manager is not running, then the `isICFRunning` function is not defined, resulting in a runtime error. You also need to make sure that your code does not have any other Progress Dynamics dependencies if it needs to work properly both with and without the framework.

Admittedly this is a fairly silly example, since the test window **only** has buttons for Progress Dynamics-dependent functions, but if you run `testlaunch.w` from a Progress session that does not start Progress Dynamics, you can confirm that the buttons do not appear:



6.4.2 Using the `getManagerHandle` function

The handles of the basic Managers are available as global variables defined in `src/adm2/globals.i`. New Managers and other useful handles should not be defined in this way, to avoid the unnecessary proliferation of global variables and the need to recompile every procedure in the framework, and also in your application, when one is added.

For all Managers, the Configuration File Manager supports a function that returns the handle based on the Manager’s name, or more precisely, the Manager Type Code from the Manage Type maintenance utility under the Session menu. If you look at all the existing Managers, you can see the Manager Type Codes for them, and also note the value of the Static Handle field.

The first thing to note is that the Configuration File Manager itself is not on the list. This is because it’s always the first thing started and always starts everything else, so only other Managers are listed in the configuration information for a session.

Look at the Static Handle field in [Figure 6–6](#) and note that all the basic Managers except for the Connection Manager have a two- or three-letter symbol for the handle name. This indicates that there is a static handle for the Manager. The Connection Manager has a Static Handle code of **NON**, short for none, meaning **No static handle available**.

Manager Type Code	Manager Type Name	System Owned	Write to Config	Static Handle
ConnectionManager	Connection Manager	YES	YES	NON
CustomizationManager	Customization Manager	YES	YES	NON
GeneralManager	General Manager	YES	YES	GM
LocalizationManager	Localization Manager	YES	YES	TM
ProfileManager	Profile Manager	YES	YES	PM
RepositoryDesignManager	Repository Design Manager	YES	YES	NON
RepositoryManager	Repository Manager	YES	YES	RM
RequestManager	Request Manager	YES	YES	NON
RIManager	Referential Integrity Manager	YES	YES	RI
SecurityManager	Security Manager	YES	YES	SEM
SessionManager	Session Manager	YES	YES	SM
UserInterfaceManager	User Interface Manager	YES	YES	NON

Figure 6–6: Static Handle field of the Manager Type Control window

If you edit the Manager Type for one of the other Managers, a Static Handle name such as **gshSessionManager** is shown. [Figure 6-7](#) shows that there is no static handle for the Connection Manager.

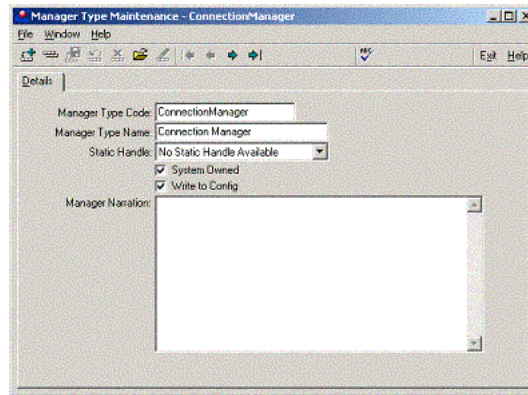


Figure 6-7: Manager Type Maintenance - Connection Manager

This is because the Connection Manager was created after the other basic Progress Dynamics Managers, and observes the policy of not creating new handles for new Managers.

To obtain its handle, use the **getManagerHandle** call:

```
hManager = DYNAMIC-FUNCTION('getManagerHandle', pcManagerTypeCode).
```

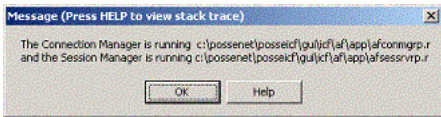
You can reference the API calls in the Configuration File Manager as if they were in your application procedure, but because no function prototypes for the functions are included, you need to use the DYNAMIC-FUNCTION syntax to avoid a compile-time error message that the compiler does not recognize the function name.

Here is an example of using this call. Note that because you do not use the global handle variables in `globals.i`, you do not need to reference the include file unless your code elsewhere references one of the other Manager handles directly:

```
/* testManagerHandle.p -- test the getManagerHandle function. */

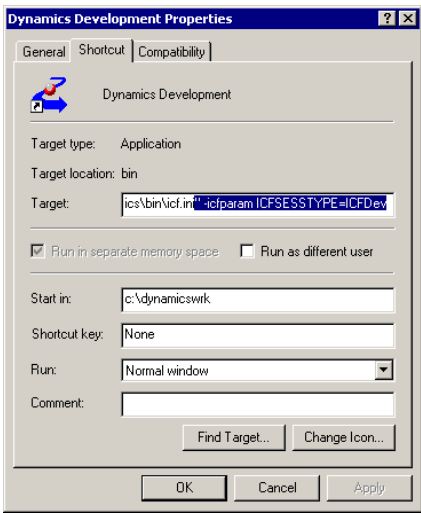
DEFINE VARIABLE hConn AS HANDLE      NO-UNDO.
DEFINE VARIABLE hSess AS HANDLE      NO-UNDO.
hConn = DYNAMIC-FUNCTION('getManagerHandle', 'ConnectionManager').
hSess = DYNAMIC-FUNCTION('getManagerHandle', 'SessionManager').
MESSAGE "The Connection Manager is running " hConn:FILE-NAME SKIP
        "and the Session Manager is running" hSess:FILE-NAME.
```

Run this code to see the Message window:

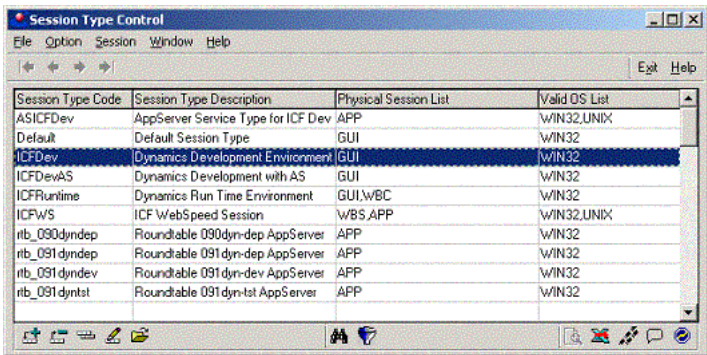


6.4.3 Checking session types and session parameters

Each Progress Dynamics session is started up based on a session type. The session type is the one parameter normally passed in to the startup command using the ICFSESSTYPE parameter of the icfstartup Progress startup option:



In this example, the ICFSESSTYPE used at startup, which is really a **logical** session type that can have a number of properties of its own, is **ICFDEV**. Look at the Session Types to see that the **physical** session type for ICFDEV is **GUI**:



Use the **getPhysicalSessionType** function to query the physical session type:

```
CType = DYNAMIC-FUNCTION('getPhysicalSessionType').
```

These are the possible physical types:

- **APP** — If the SESSION:CLIENT-TYPE is APPSERVER, it is an AppServer session.
- **BTC** — If the SESSION:CLIENT-TYPE is 4GLCLIENT, and SESSION:BATCH-MODE is yes, it is a batch client.
- **CUI** — If the SESSION:CLIENT-TYPE is 4GLCLIENT, and the SESSION:DISPLAY-TYPE is “TTY”, it is a character client.
- **GUI** — If the SESSION:CLIENT-TYPE is 4GLCLIENT, and neither of the previous 2 conditions were met, it is a GUI client.
- **WBC** — If the SESSION:CLIENT-TYPE is WEBCLIENT, it is a WebClient session.
- **WBS** — If the SESSION:CLIENT-TYPE is WEBSPEED, it is a WebSpeed Transaction Agent.
- If none of the previous conditions have been met, the physical session type is left blank, indicating that the physical session type is unrecognized.

You can also access the values of any of the session properties, or even define new properties of your own, using the getSessionParam and setSessionParam functions:

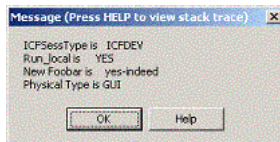
```
cParamValue = DYNAMIC-FUNCTION('getSessionParam', pcParam).
```

```
lResult = DYNAMIC-FUNCTION('setSessionParam', pcParam, pcValue).
```

Here is an example of using these. If you set a parameter that was not defined before, it is defined for you and you can retrieve its value later:

```
/* testSessionParam.p -- tests get/setSessionParam and
   getPhysicalSessionType in the Configuration File Manager). */
DEFINE VARIABLE cSessType AS CHARACTER NO-UNDO.
DEFINE VARIABLE cLocal   AS CHARACTER NO-UNDO.
DEFINE VARIABLE cFoobar  AS CHARACTER NO-UNDO.
DEFINE VARIABLE cPhysical AS CHARACTER NO-UNDO.
cSessType = DYNAMIC-FUNCTION('getsessionparam','icfsesstype').
cLocal = DYNAMIC-FUNCTION('getsessionparam','run_local').
DYNAMIC-FUNCTION('setsessionparam','foobar','yes-indeed').
cFoobar = DYNAMIC-FUNCTION('getsessionparam','foobar').
cPhysical = DYNAMIC-FUNCTION('getphysicalsessiontype').
MESSAGE "ICFSessType is " cSessType SKIP
        "Run_local is   " cLocal   SKIP
        "New Foobar is  " cFoobar  SKIP
        "Physical Type is" cPhysical.
```

Running this code results in this Message window:



For more information about other API calls see the Configuration File Manager section in the [Progress Dynamics Managers API Reference](#).

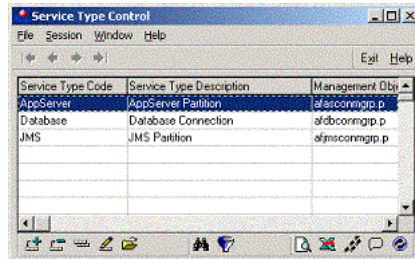
6.5 Using the Connection Manager

The Configuration File Manager starts the Connection Manager during the startup process. In fact, the Configuration File Manager assumes that the Connection Manager is the first manager specified in the Managers section of the ICFCONFIG.XML file.

The Connection Manager is a universal API that wraps the different types of services so that developers can connect to the services using a standard API.

To provide the specific communication to each type of service, there must be a Service Type Manager procedure for the service. The framework provides a Database Manager, an AppServer Manager, and a JMS Manager for you to take care of communicating with their respective Service Types. To make a connection to the service, the programmer simply makes a call to the Connection Manager with the name of the service to be connected.

The Service Types and service connection data are all stored in the database and those that are required at startup are written to the ICFCONFIG.XML file. The Configuration File Manager takes care of calling the Connection Manager to establish the initial connections. The Service Types are maintained in the Service Type Control window under the Session menu:



When the Session Manager itself starts up, it obtains a complete list of all the services and service types that the session has access to and provides these to the Connection Manager. The Connection Manager only connects to these services upon request.

The protocol required to communicate with a service is as follows:

1. Register the Service Type Manager.

To register the Service Type Manager of the service, the application must start the Service Type Manager as a persistent procedure, and then making a call to the procedure **setServiceTypeManager** in the Connection Manager. Remember that there is no global variable for the Connection manager, so you need to run `getManagerHandle` to obtain it:

```
hConnection = DYNAMIC-FUNCTION('getManagerHandle', 'ConnectionManager').
RUN MyService.p PERSISTENT SET hMyService.
RUN getServiceTypeManager IN hConnection (INPUT hMyService).
```

Once the Service Type Manager has been registered, any calls that are made to the Connection Manager are diverted to the appropriate Service Type Manager.

2. Register the service.

To register a service, the application needs to make a call to **registerService** in the Connection Manager, passing the handle to a buffer that contains the related data:

```
RUN registerService IN hConnection ( INPUT bMyService).
```

3. Connect to the service.

To connect to a service the application needs to call **connectService** in the Connection Manager passing the name of the service to be connected:

```
RUN connectService IN hConnection ('MyService').
```

4. Disconnect from the service.

To disconnect a service the application calls **disconnectService** in the Connection Manager passing the name of the service to be disconnected:

```
RUN disconnectService IN hConnection ('MyService').
```

5. Interact with the service.

Calls to interact with a service are summarized as follows:

```
lConnected = DYNAMIC-FUNCTION('isConnected' IN hConnection,  
pcServiceName AS CHARACTER).
```

```
cServiceHandle = DYNAMIC-FUNCTION('getServiceHandle' In hConnection,  
pcServiceName)  
/* Note that the handle is returned as character string */
```

```
cPhysicalService = DYNAMIC-FUNCTION('getPhysicalService' IN hConnection,  
pcMyService).
```

```
cConnectionString = DYNAMIC-FUNCTION('getConnectionString' IN  
hConnection, pcMyService).  
/* This returns the character string needed to connect to the service */
```

```
cConnectionParams = DYNAMIC-FUNCTION('getConnectionParams' IN
hConnection,
pcMyService).
/* This returns the connection parameter string as it is stored on the
database.*/
```

6.6 Using the Profile Manager

This section contains an example that enhances the user profile settings for dynamic Browsers, allowing column resizing, moving, and sorting to be saved to the repository as user preferences. Frank Beusenberg of Progress Software created the original code for this example. This behavior has been integrated into the framework as a standard feature in a slightly different way. It serves as an excellent example of extending the standard Objects, their attributes, and their behavior, as well as using the Managers to assist you in this.

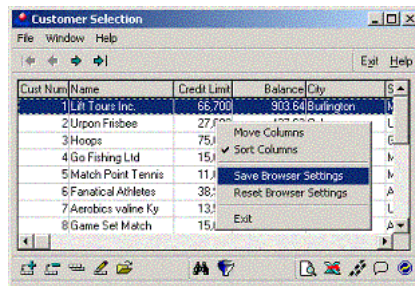
You already know that the framework saves window sizes and positions for each user. This example allows users to change column sizes, column order, and column sorting, and save these as permanent preferences as well.

This example uses the following Profile Manager calls:

- **checkProfileDataExists**
- **getProfileData**
- **setProfileData**

In addition, it contains some confirmation questions and messages to put up using the **{aferrortxt.i}** message include file, and the **askQuestion** and **showMessages** calls from the Session Manager API.

The final result is a popup menu, available on Browsers where users can select whether selecting a column header should be used to move columns or to sort on a column. They can also save their browser settings or reset to the browser's original settings:



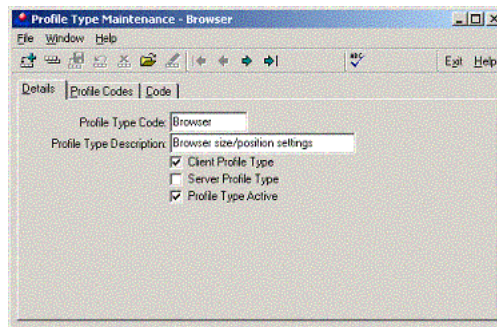
6.6.1 Creating user profile codes and types

To create user profile types and codes and to extend the set of User Profile Codes and Types that are supplied with the framework, follow these steps:

- 1 ♦ Choose the **User Profile Code Control** from the Progress Dynamics Administration Session menu:



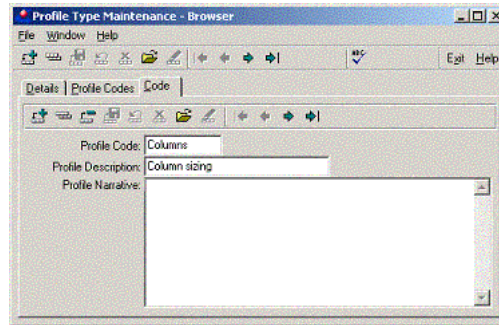
The **Profile Types Maintenance - Browser** window appears:



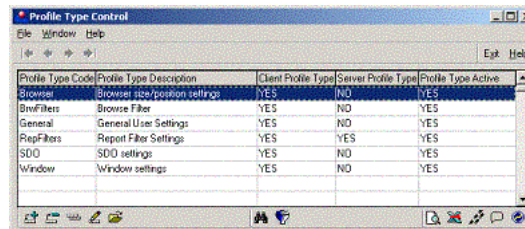
Here you see the built-in codes that allow you to save window sizes and other preferences to the profile tables.

- 2 ♦ Press **Add**.
- 3 ♦ Name the new code **Browser** and give it a Description.
- 4 ♦ Leave **Client Profile Type** checked on, but uncheck **Server Profile Type**. This combination tells the Profile Manager that this is a code that is maintained on the client side of the profile cache at runtime, but not on the server side. This saves the overhead of sending every client setting change back to the server as it is made. All changes are sent back to the server to be written into the database at the end of the session.
- 5 ♦ Leave the **Profile Type Active** toggle checked.
- 6 ♦ Save this new Profile Type.

- 7 ♦ Select the **Code** tab to add specific codes to the type. Press **Add** in the Toolbar inside the tab folder and add a code called **Columns** and a Profile Description:



- 8 ♦ Save this and press **Add** again.
- 9 ♦ Add a second code called **Sorting** with its own description, **Column sorting**, and save.
- 10 ♦ Select the **Profile Codes** tab to see both of your new codes.
- 11 ♦ Exit the Profile Type Maintenance window and see your new Type added to the list:



6.6.2 Creating new browser properties

Now you can start to create the code to support your new feature. This is an extension to the standard Browser behavior, so you use the custom files for your changes.

First you need to define two new properties that the Browser code will need. These are:

- **BrowsePopupActive** — This logical property could be set to disable the Popup menu so that users can't change the appearance of a Browser.
- **BrowseColumnsMovable** — This logical property indicates whether selecting a column header allows the column to be moved (if the property is TRUE) or allows sorting on the column (if the property is FALSE).

If you were doing this for a standard ADM2 application that is **not** using Progress Dynamics, you would define these new properties in a custom include file that is part of the class definition. For the Browser this is **src/adm2/custom/brspropcustom.i**. You would copy this file to a local directory with the same relative path, and edit it to add your properties, adding the lines shown in bold below to the template code in the Main Block of the include file:

```
/* PROPERTY CODE FOR ADM2 NON-DYNAMICS CUSTOMIZATION */
&IF "{&ADMSuper}":U EQ "":U &THEN
    {src/adm2/custom/brsprtocustom.i}
&ENDIF

&GLOBAL-DEFINE xpBrowsePopupActive /* Is the Browse POPUP Menu active? */
&GLOBAL-DEFINE xpBrowseColumnsMovable /* Are the Browse Columns Movable? */

&IF "{&ADMSuper}":U = "":U &THEN
    ghADMProps:ADD-NEW-FIELD( "BrowsePopupActive":U, "LOGICAL":U, 0, ?, YES ).
    ghADMProps:ADD-NEW-FIELD( "BrowseColumnsMovable":U, "LOGICAL":U, 0, ?, NO
).
&ENDIF
```

The xp preprocessors tell the framework that references to the properties can be made directly, by reading the property values out of the temp-table record where they are stored, which is an optimization over running a function to access them. The ADD-NEW-FIELD statements add fields to hold the properties to this temp-table, with a data-type of LOGICAL, no extent, no specific format, and an initial value of YES and NO respectively.

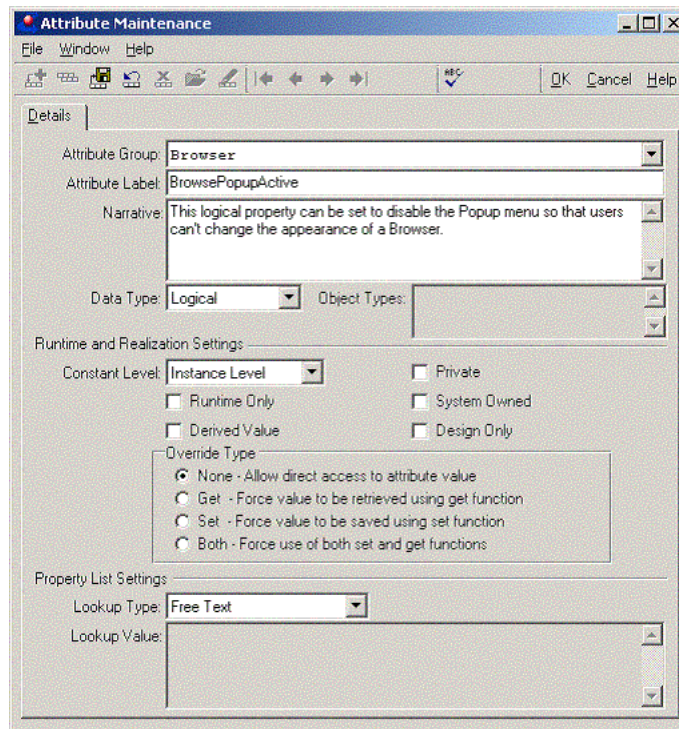
When you re-compile the dynamic Browser procedure later, these two custom properties would be added to its definition.

However, as explained elsewhere, Progress Dynamics does not use these property include files, neither the standard ones for SmartObject classes nor the custom ones for extended classes. All attributes are defined in the repository, and it is here that the Progress Dynamics runtime engine looks to assemble the list of all attributes for an Object.

For this reason, you do not use the custom property include file to define extended properties for a class. Instead, you simply define them in the repository. To do this, you need to define the attributes and then associate them with the dynamic Browser.

To define the attributes, follow these steps:

- 1 ♦ Choose **Attributes**→**Attribute Control** from the Progress Dynamics Development menu.
- 2 ♦ Press **Add** to define a new attribute in the Attribute Maintenance window:

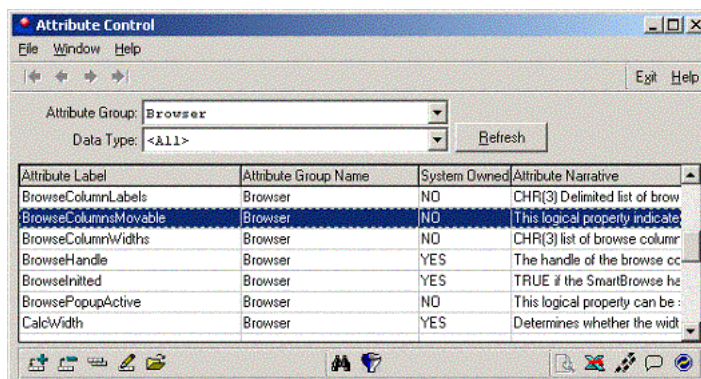


- 3 ♦ Define each of the two new attributes in turn as shown in the previous figure.
- 4 ♦ Put them in the Browser Attribute Group, and give them a Data Type of **Logical**.
- 5 ♦ Leave the Constant Level at its default value of **Instance Level**, which means that the attribute values can be defined for a Browser instance. None of the toggles apply.
- 6 ♦ Leave the Override Type at its default value of None.
- 7 ♦ Leave the Lookup Type at its default value of Free Text. In fact, the dynamic property sheet will recognize the data type of Logical and automatically present True and False and the two valid choices for a user setting the property values for an Object.

Note that the Object Types list in this window is not enabled for input. This displays the Object Types the attribute has been added to, but is not directly updatable. After you complete the next step of adding the attributes to the DynBrow class, then you can return to this window and verify that the DynBrow class is shown as the Object Type for the attributes.

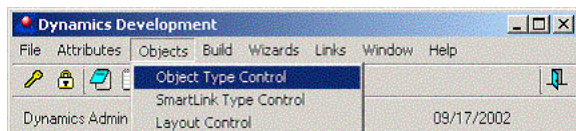
- 8 ♦ Save each of the two new attributes in turn.

When you return to the Attribute Control browse, press Refresh to see the two new Browser attributes along with all the standard ones:



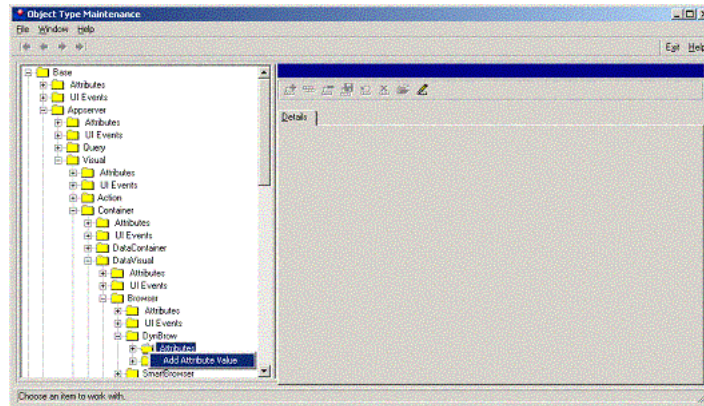
Although you have put the new attributes into the Browser group, this is only an informal grouping. Next you must follow these steps to add them explicitly to the DynBrow class for the dynamic Browser, so that they are added to the Object attributes at runtime.

- 9 ♦ Go into the Object Type Control from the Progress Dynamics Development menu:

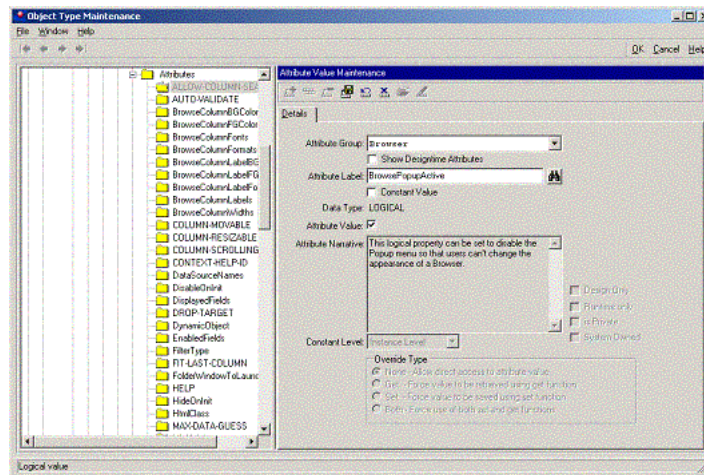


- 10 ♦ Here you must expand the Base node down through AppServer, Visual, Container, DataVisual, and Browser to locate the DynBrow class. As noted elsewhere, the hierarchy of this Object Type treeview does not fully represent the hierarchy of classes for every Object Type, as some classes (such as AppServer) are only used optionally or only used by a subset of the Objects at that level.

- 11 ♦ Expand the DynBrow node and then right click on the Attributes node to see the popup menu with the Add Attribute Value option. Select that option:

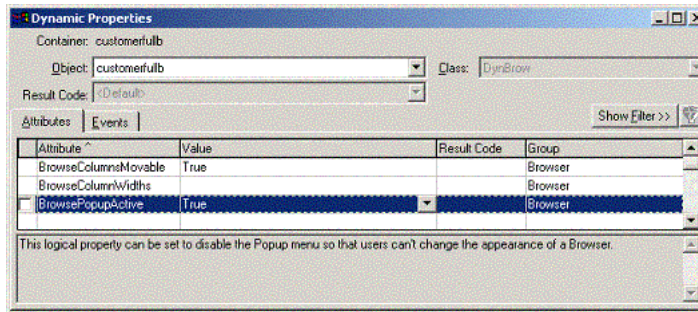


- 12 ♦ In the Attribute Value Maintenance frame to the right, select the Browser Attribute Group and the BrowsePopupActive Attribute Label:



- 13 ♦ Because this is a Logical attribute its value is represented as a toggle box. Check the Attribute Value toggle on to set the initial value of the attribute to True.
- 14 ♦ Save this and then press the Add button to add a second reference to associate the BrowseColumnsMovable attribute with the DynBrow class.

Now your two new attributes will be added to every new dynamic Browser as it is created at runtime. You can verify this by clearing the repository cache, opening a dynamic Browser, and looking at its properties in the dynamic property sheet:



6.6.3 Creating a custom super procedure for the browser

Next you need to activate the custom super procedure that extends the Browser class. Copy the file **src/adm2/custom/browsercustom.i** to a local directory with the same relative pathname, and edit it to remove the comments around the start-super-proc statement that starts **browsercustom.p**:

```
/* ***** Main Block ***** */
/* Starts here the custom super procedure
   Uncomment to run it */

RUN start-super-proc ("adm2/custom/browsercustom.p":U).
```

Add the supporting code for the popup menu. Copy the file **src/adm2/custom/browsercustom.p** to a local directory with the same relative path. The remainder of this section highlights key parts of the code that needs to be added to the super procedure. A complete version of the code is available in the **browsercustom_popup.p** file that accompanies this documentation. You can refer to this procedure for more code details, and run it (after renaming it of course) to try out the feature.

The first requirement is that there must be functions that support getting and setting the two new properties you just defined. As we noted, the xp preprocessors you defined allow code within the super procedure itself, or in other super procedures in the Browser support code, to access the property directly in the property temp-table record for the object. This is done using the {get} and {set} include files that you've seen elsewhere in this documentation. The include files access the property without using a function call, which is a small optimization.

But other objects cannot use this convention because they do not have access to the temp-table record directly. So you need to define **get** and **set** functions for every property that should be retrievable or settable from outside the object itself and its supporting code.

The form of these functions is simple. Here's the **get** function for the **BrowsePopupActive** property:

```
FUNCTION getBrowsePopupActive RETURNS LOGICAL
  ( ) :
/*-----
  Purpose:  Returns the value of the BrowsePopupActive property.
-----*/

  DEFINE VARIABLE lActive AS LOGICAL    NO-UNDO.

  {get BrowsePopupActive lActive}.
  RETURN lActive.

END FUNCTION.
```

As you can see, it simply turns around and accesses the property via the {get} include file, which is available to it because it is within the class hierarchy for the object, where the include files can be used.

The **set** property function is similar:

```
FUNCTION setBrowsePopupActive RETURNS LOGICAL
  ( INPUT pActive AS LOGICAL ) :
/*-----
  Purpose:  Sets the BrowsePopupActive property.
-----*/

  {set BrowsePopupActive pActive}.

  RETURN TRUE.
END FUNCTION.
```

You need equivalent functions for the **BrowseColumnsMovable** property as well.

6.6.4 Customizing the initializeObject procedure

The next piece of code you need is a localization of the initializeObject procedure. This should first invoke the standard behavior with a RUN SUPER statement:

```
/* Run default behavior first. */  
  
RUN SUPER.
```

Next it retrieves the BrowsePopupActive property value and creates the popup menu if the property hasn't been set to FALSE:

```
{get BrowsePopupActive lPopupActive}.  
IF lPopupActive THEN  
    RUN createBrowsePopupMenu ( INPUT TARGET-PROCEDURE ).
```

NOTE: The code excerpts here are not complete. All variable references, for example, are defined in the procedure or function where they are used. See the accompanying procedure file for the complete code.

Before looking at the createBrowsePopupMenu procedure the code needs to identify the key under which the profile values are stored. In order to allow you to customize a Browser's appearance differently in different container windows, the container's name and the logical Browser name are combined to form the key. If the container does not have a LogicalObjectName property value, then it is presumably a static container, so the procedure handle's FILE-NAME is used instead:

```
/* Create the ProfileKey. */  
{get ContainerSource hContainerSource}.  
IF VALID-HANDLE( hContainerSource ) THEN  
    {get LogicalObjectName cContainerName hContainerSource} NO-ERROR.  
ELSE  
    cContainerName = "".  
  
IF ( cContainerName = "" ) AND VALID-HANDLE( hContainerSource ) THEN  
    cContainerName = hContainerSource:FILE-NAME.  
  
cProfileKey = cContainerName + cBrowserName.
```


Using checkProfileDataExists and getProfileData

Now you are ready to access the Profile Manager itself. Use a call to see whether profile data is defined for the profile code and key:

```
/* Try to restore column settings. Check the Profile Manager if the profile
   data we're looking for actually exists... */

RUN checkProfileDataExists IN gshProfileManager (
    INPUT "Browser",
    INPUT "Columns",
    INPUT cProfileKey,
    INPUT YES,
    INPUT NO,
    OUTPUT lProfileExists ).
```

The syntax for the **checkProfileDataExists** call is:

```
RUN checkProfileDataExists IN gshProfileManager
( INPUT pcProfileTypeCode,
  INPUT pcProfileCode,
  INPUT pcProfileDataKey,
  INPUT plCheckPermanentOnly,
  INPUT plCheckCacheOnly,
  OUTPUT plExists ).
```

These are its parameters:

- **INPUT pcProfileTypeCode (CHARACTER)** — This is the code for the profile information type that you defined in the User Profile Code maintenance utility, Browsers in the case of this example.
- **INPUT pcProfileCode (CHARACTER)** — This is the sub-code for the type that you also defined in the User Profile Code utility. You created profile codes for Columns and Sorting for the example.
- **INPUT pcProfileDataKey (CHARACTER)** — This is whatever key you wish to define for storing and retrieving data for a particular code. In the case of the example, it's the container name plus the browser name.
- **INPUT plCheckPermanentOnly (LOGICAL)** — This flag tells the Profile Manager whether it should look only for settings marked as being Permanent, as opposed to settings that are being saved for the Session only. This is normally a settable user preference, although in our somewhat simplified example, Browser settings are always saved as Permanent, as you will see later.

- **INPUT pcCheckCacheOnly (LOGICAL)** — If this logical flag is set to TRUE, then only the currently cached data will be searched for the settings. If it's FALSE, then the database table will also be checked if the setting is not found in the cache.
- **OUTPUT plExists (LOGICAL)** — This returned value tells you whether the profile data you want is defined or not. In the example, there should be profile data for the key if this user had previously saved Browser settings for this Browser in this container, otherwise not.

If there is already profile data for the combination of user, container, and Browser, then the next call retrieves it:

```
IF !ProfileExists THEN
DO:
    ASSIGN rRowID = ?.

    RUN getProfileData IN gshProfileManager (
        INPUT "Browser",
        INPUT "Columns",
        INPUT cProfileKey,
        INPUT NO,
        INPUT-OUTPUT rRowID,
        OUTPUT cProfileData ).
```

The syntax for the **getProfileData** call is:

```
RUN getProfileData IN gshProfileManager
( INPUT pcProfileTypeCode,
  INPUT pcProfileCode,
  INPUT pcProfileDataKey,
  INPUT plNextRecordFlag,
  INPUT-OUTPUT prRowID,
  OUTPUT pcProfileData).
```

The **getProfileData** procedure has these parameters:

- **INPUT pcProfileTypeCode (CHARACTER)** — As above.
- **INPUT pcProfileCode (CHARACTER)** — As above.
- **INPUT pcProfileDataKey (CHARACTER)** — As above.

- **INPUT `plNextRecordFlag` (LOGICAL)** — The `getProfileData` procedure can be used in some cases to walk through all profile records matching the combination of profile type code, profile code, and data key passed in. You can omit one or more of these parameters if you want to retrieve all records matching a partial set of keys. In such a case, `getProfileData` returns the profile data from one record at a time. In order to retrieve all the records, you must call `getProfileData` with the `NextRecordFlag` set to `NO` on the first call, and then continue to call it with the `NextRecordFlag` set to `YES` until an unknown value in the `RowID` parameter described next indicates that there are no more matching records. To retrieve a single data value, set this parameter to `NO`.
- **INPUT-OUTPUT `prRowID` (ROWID)** — `getProfileData` always returns the `RowID` of the record that supplied the profile data, using this INPUT-OUTPUT parameter. If you are walking through a set of records, then you must pass the current `RowID` back in with each call after the first one, along with a `NextRecordFlag` of `YES`. If you only want to retrieve a single profile data value matching the three key values for Type Code, Profile Code, and Profile Key, then you **must** pass in the unknown value for this parameter; otherwise the Profile Manager will ignore the first three key value parameters and use the `RowID` as the basis for the request. If this parameter comes back as the unknown value, then there was no matching record (or no **next** record if `NextRecordFlag` was `YES`).
- **OUTPUT `pcProfileData` (CHARACTER)** — You can store any meaningful character string you want as the data associated with a profile key. This parameter returns it to you.

In its search for your profile data, `getProfileData` first looks in the client cache if it's running client side. If the data is not found and this is not a cached profile type, it looks in the database. The procedure always checks for Session data first, then Permanent data. If a `RowID` is passed in, then this is used to find the record. The `RowID` is the `RowID` of the temp-table if this is a client-only profile type; otherwise it is the `RowID` of the database record.

Profile types are either maintained completely using the client cache, or always read from the database. When trying to get profile data, the procedure first fully caches the entire profile type if it is not already cached **and** this is a client-only profile type. Normally the full cache is built at application start-up, but profile types can be cached as used.

The initializeObject procedure now has the profile data for this user, container, and Browser, if it has been previously saved. If there is data, then the code manipulates the order and size of each Browse column accordingly. Because this code is not pertinent to our study of the Profile Manager, just a couple of key lines are reproduced here:

```
...  
hBrowse:MOVE-COLUMN( iColumnPosition, iLoop ).  
hColumn:WIDTH-PIXELS = INTEGER( ENTRY( 2, cColumnEntry, CHR(4) ) ).  
...
```

Refer to the procedure file for the complete code example.

Next, the code makes similar calls to checkProfileDataExists and getProfileData for data under the Sorting code. If a sort setting comes back, then it resets the query's sort sequence accordingly, and reopens the SDO's query. Note that the SDO is the Data-Source for the Browser, so following that link identifies the procedure where the query must be reset and re-opened. The REFRESHABLE attribute setting keeps the Browse from flashing as it is being manipulated:

```
hBrowse:REFRESHABLE = NO.  
DYNAMIC-FUNCTION( "setQuerySort":U IN hDataSource, cProfileData ).  
DYNAMIC-FUNCTION( "openQuery":U IN hDataSource ).  
hBrowse:REFRESHABLE = YES.
```

6.6.5 Creating the Popup menu

The initializeObject procedure calls **createBrowsePopupMenu** unless the menu has been disabled. This custom procedure creates the menu and its menu items as dynamic 4GL objects.

The first thing to examine is why it is necessary to pass in the procedure handle of the Browser object as an INPUT parameter. The call to createBrowsePopupMenu in initializeObject looks like this:

```
RUN createBrowsePopupMenu ( INPUT TARGET-PROCEDURE ).
```

The called procedure defines it as an INPUT parameter to be used later in the code:

```
DEFINE INPUT    PARAMETER hTarget    AS HANDLE    NO-UNDO.
```

So why can't createBrowsePopupMenu just reference TARGET-PROCEDURE directly, as you have seen in many other cases? For the answer, refer back to the discussion and diagrams in [Chapter 1, "Writing Super Procedures for Progress Dynamics Objects."](#) There we explained that in order for any internal procedure to be able to refer to TARGET-PROCEDURE successfully, it must itself be run IN TARGET-PROCEDURE. In this case, because createBrowsePopupMenu is in the same procedure file as the initializeObject procedure that calls it, the RUN statement just makes the call directly. For this reason, a reference to TARGET-PROCEDURE within createBrowsePopupMenu evaluates to the handle of the super procedure browsercustom.p, not to the Browser SmartObject instance itself, and this is not what the code needs. Thus either the TARGET-PROCEDURE handle needs to be passed in as an argument, or the RUN statement must bounce back to browsercustom.p by being formulated as RUN createBrowsePopupMenu IN TARGET-PROCEDURE. Either technique works.

The code first creates the menu object itself:

```
CREATE MENU hPopupMenu
      ASSIGN POPUP-ONLY = TRUE
             TITLE      = "Browser Menu".
```

It then creates each of the menu items:

```
CREATE MENU-ITEM hColsMovableItem
      ASSIGN PARENT      = hPopupMenu
             LABEL       = "&Move Columns"
             NAME        = "ColsMovable"
             TOGGLE-BOX  = TRUE
             CHECKED     = FALSE.

Etc.
```

Again, see the accompanying procedure file for the complete code.

Next, the procedure attaches the menu to the browse widget handle, which it has retrieved using the BrowseHandle property of the Browser object. Just to clarify, the BrowseHandle is the widget handle of the browse control itself. The hTarget parameter passed in to the procedure is the **procedure** handle of the Browser SmartObject instance:

```
/* Add POPUP Menu to the Browse Widget. */

hBrowse:POPUP-MENU = hPopupMenu.
```

Finally, the code defines trigger procedures for the MENU-DROP event of the menu itself, the VALUE-CHANGED event of the Movable Item and Sortable Item, which control the check marks on those items, and the CHOOSE event for the Save and Reset options. The PERSISTENT RUN syntax is required to associate an internal procedure such as eventMenuItemValueChanged with a dynamic menu item:

```
ON MENU-DROP OF hPopupMenu
    PERSISTENT RUN eventPopupMenuMenuDrop IN hTarget ( INPUT hPopupMenu ).
ON VALUE-CHANGED OF hColsMovableItem
    PERSISTENT RUN eventMenuItemValueChanged IN hTarget
        ( INPUT hColsMovableItem ).
ON VALUE-CHANGED OF hColsSortableItem
    PERSISTENT RUN eventMenuItemValueChanged IN hTarget
        ( INPUT hColsSortableItem ).
ON CHOOSE OF hSaveItem
    PERSISTENT RUN eventMenuItemChoose IN hTarget ( INPUT hSaveItem ).
ON CHOOSE OF hResetItem
    PERSISTENT RUN eventMenuItemChoose IN hTarget ( INPUT hResetItem ).
```

Note that it is important for the trigger definitions to specify that the event procedure is run IN TARGET-PROCEDURE. Because TARGET-PROCEDURE itself is not available here, as noted, they do this using the procedure handle hTarget that was passed in as a parameter. In this way the individual trigger procedures can correctly reference TARGET-PROCEDURE themselves.

6.6.6 Defining the Popup menu event procedures

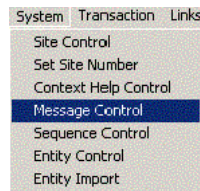
There are several procedures that respond to menu events, including:

- [The Menu Item Choose procedure and using messages in the Message table](#)
- [Using the Session Manager's askQuestion procedure](#)
- [Using the setProfileData procedure](#)
- [Using the SessionManager's showMessages procedure](#)
- [Changing the Browser's COLUMN-MOVABLE attribute](#)
- [Setting the popup menu's state on menu drop](#)
- [Testing the extended browser profile data](#)

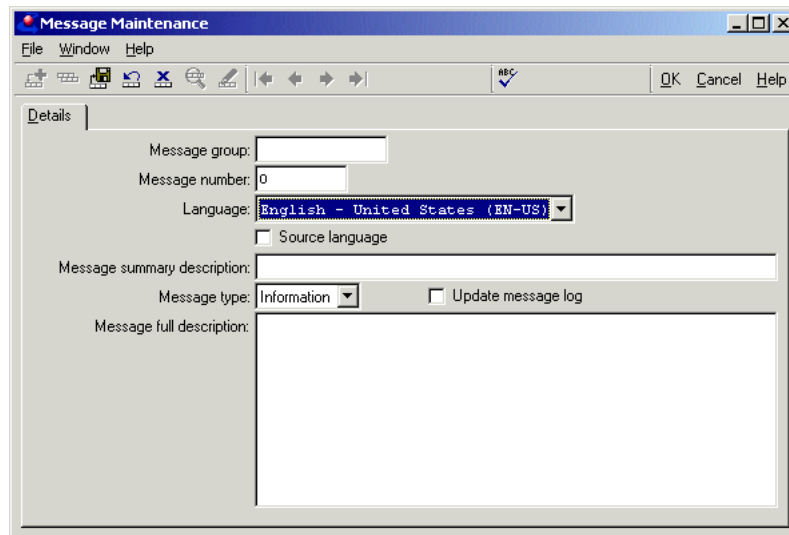
The Menu Item Choose procedure and using messages in the Message table

The **eventMenuItemChoose** procedure is run when the Save or Reset menu item is chosen. As part of this process, four different messages are used to interact with the user. Two are questions confirming the intent to Save or Reset settings, and the other two are confirmation messages that the Save or Reset succeeded. It is questionable whether either the questions or the confirmation are really appropriate to the user interface, since saving column size and position preferences is not such a major operation that the user should necessarily be forced to OK two message boxes to complete the request. The messages are here more to demonstrate how to use the Session Manager's message support in your own application code. In order to make these messages reusable and translatable, you must create them by following these steps:

- 1 ♦ Choose **System**→**Message Control** in the Progress Dynamics Administration window's System menu:



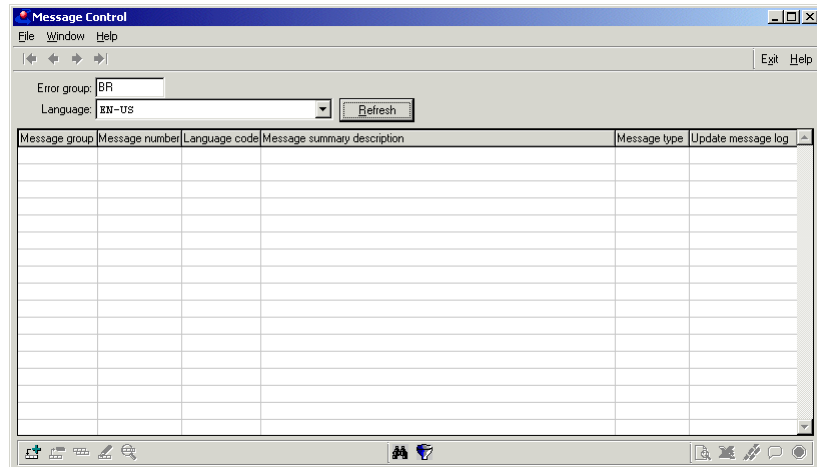
- 2 ♦ Press **Add** to bring up Message Maintenance for a new message:



Each message has a two-part key, consisting of a message group and a message number within the group. To keep the new messages distinct from standard system messages, you can put them in a new group, such as **BR**.

- 3 ♦ Press the Add button in the Message Control to add the first of your new messages.
- 4 ♦ Give it a Group of **BR** and a Number of **1**.
- 5 ♦ Set the default language and check Source Language. At least one message in a group must indicate the source language for that group. For subsequent messages, you need only check Source Language.
- 6 ♦ Enter a short message text as the Message Summary Description and a longer message in the editor box.
- 7 ♦ Select an Message Type of Question for the first two messages and Information for the next two.
- 8 ♦ Define a total of four new messages:
 - **BR-1 — Save Browser settings?**
 - **BR-2 — Reset Browser settings?**
 - **BR-3 — Saving Browser settings succeeded.**
 - **BR-4 — Resetting Browser settings succeeded.**

- 9 ♦ Return to the Message Control window and filter on the Error Group BR to see all your new messages:



Now you can use the messages in your application. The code retrieves the browse settings question, if the Menu Item chosen was SaveBrowseSettings, using the {aferrortxt.i} include file. For more information see the [Progress Dynamics Developer's Guide](#). If the Menu Item is Reset, then message 2 is retrieved instead:

```
IF ( hMenuItem:NAME = "SaveBrowseSettings" ) THEN
DO:
    ASSIGN cMessage = {aferrortxt.i 'BR' '1'} /* "Save browser settings?" */
    lDeleteProfileEntry = FALSE.
...etc.
```

Using the Session Manager's askQuestion procedure

This message text is passed to the Session Manager's **askQuestion** procedure to present the question to the user. Note that the message text returned by aferrortxt.i is specially formatted to be used by either the **askQuestion** and **showMessage** procedures, or the **checkerr.i** include file (which is also described in the [Progress Dynamics Developer's Guide](#)).

Its format is not appropriate to display directly to the user:

```
/* Ask user for save or reset confirmation. */
RUN askQuestion IN gshSessionManager (
    INPUT cMessage,
    INPUT "OK,Cancel",
    INPUT "OK",
    INPUT "Cancel",
    INPUT "",
    INPUT "",
    INPUT "",
    INPUT-OUTPUT cAnswer,
    OUTPUT cButtonPressed).

/* Only continue this if user pressed the "OK" button. */
IF ( cButtonPressed <> "OK" ) THEN
    RETURN.
```

This call puts up a message box with OK and Cancel buttons, where OK is the default on Return and Cancel is the default on Escape, and gets back the name of the button the user pressed. If this isn't the OK button then the procedure just returns.

The **askQuestion** call takes these parameters:

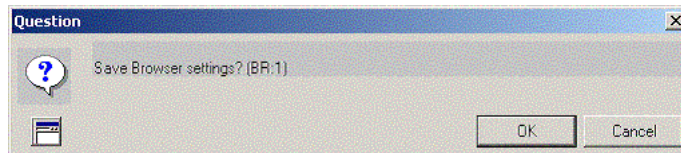
```
RUN askQuestion IN gshSessionManager
( INPUT pcMessageList,
  INPUT pcButtonList,
  INPUT pcDefaultButton,
  INPUT pcCancelButton,
  INPUT pcMessageTitle,
  INPUT pcDataType,
  INPUT pcFormat,
  INPUT-OUTPUT pcAnswer,
  OUTPUT pcButtonPressed ).
```

- **INPUT pcMessageList (CHARACTER)** — This is the specially formatted message to display.
- **INPUT pcButtonList (CHARACTER)** — This is a list of the buttons that should appear in the message dialog.
- **INPUT pcDefaultButton (CHARACTER)** — This is the default button to fire if the user presses Return instead of selecting a button.
- **INPUT pcCancelButton (CHARACTER)** — This is the button to fire if the user presses Escape instead of selecting a button.

- **INPUT pcMessageTitle (CHARACTER)** — This optional argument is the title that should appear in the message dialog. The default is Question.
- **INPUT pcDataType (CHARACTER)** — This optional argument is the data type of the answer to the question, if one is requested.
- **INPUT pcFormat (CHARACTER)** — This optional argument is the format of the answer to the question.
- **INPUT-OUTPUT pcAnswer (CHARACTER)** — The question dialog can, if needed, prompt the user for an answer to the question it poses. This parameter returns the answer to the question being asked. If a non-blank value is passed in, it will be used as an initial answer value. If the data type and format are blank, then no response will be requested and only the button pressed with a blank answer will be returned.
- **OUTPUT pcButtonPressed (CHARACTER)** — This is the label of the button that was pressed (explicitly or by default if the user pressed Return or Escape). If there is no explicit answer to the question, as defined in the pcAnswer, pcDataType, and pcFormat arguments, then the button pressed is effectively the user's answer. If the button labels are passed in with an ampersand (&), then the button pressed will also contain the ampersand when returned, so you will need to check for this.

If the session is running on the server-side, messages cannot be displayed. Instead they are written to the message log. On the server side there is no user interface, so the default button label and answer are always returned. On the client side the messages are displayed in a dialog window. The procedure checks the `suppressDisplay` property in the Session Manager and if this is set to YES, it does not display the message but simply passes the message to the log as would be the case for a server-side message. This is useful when your application is running take-on or bulk load procedures client-side.

The `askQuestion` call in the example looks like this at runtime:



Using the `setProfileData` procedure

The code then gathers up the browse settings for column order and column widths and passes it to the Profile Manager using the procedure **`setProfileData`**:

```
/* Save column settings to user profile. */
ASSIGN rRowID = ?.

RUN setProfileData IN gshProfileManager (
    INPUT "Browser",
    INPUT "Columns",
    INPUT cProfileKey,
    INPUT rRowID,
    INPUT cColumnData,
    INPUT lDeleteProfileEntry,
    INPUT "PER" ).
```

The `setProfileData` procedure has the following syntax:

```
RUN setProfileData IN gshProfileManager
( INPUT pcProfileTypeCode,
  INPUT pcPProfileCode,
  INPUT pcProfileDataKey,
  INPUT prRowid,
  INPUT pcProfileDataValue,
  INPUT plDeleteFlag,
  INPUT pcSaveFlag ).
```

The `setProfileData` procedure takes these parameters:

- **INPUT `pcProfileTypeCode` (CHARACTER)** — As above.
- **INPUT `pcPProfileCode` (CHARACTER)** — As above.
- **INPUT `pcProfileDataKey` (CHARACTER)** — As above.
- **INPUT `prRowid` (ROWID)** — As above; if this is passed in the first three arguments do not apply (and they will be ignored if they are specified).
- **INPUT `pcProfileDataValue` (CHARACTER)** — This is the character string to assign as the data for this profile entry. The format and content is up to the application that defines and uses the profile code.

- **INPUT pIDeleteFlag (LOGICAL)** — If this is TRUE, then the profile data for the matching code will be deleted rather than set. In the case of the example this value corresponds to the variable lDeleteProfileEntry, which is TRUE if the user selected Reset, and FALSE if the user selected Save.
- **INPUT pcSaveFlag (CHARACTER)** — This parameter should be set to either PER, to mark the settings as Permanent, or SES, to save them for the duration of the session only. If the save flag is set to Session, then the user's Session ID is stored in the profile table's context id field, otherwise the field is left blank to indicate a permanent value.

Note that setProfileData can be used to either set or clear profile data for multiple records that match a partial key passed in. If either one or two of the first three parameters are passed in, but not all three, then all matching profile records are set to the specified profile data value; or they are deleted if the Delete Flag is set.

Back in the example code, the same setProfileData operation is then done for the Sorting profile code and the current sort sequence.

Using the SessionManager's showMessages procedure

Finally, the code retrieves the appropriate confirmation message that the operation succeeded, and displays it to the user using the Session Manager's **showMessages** procedure:

```
/* Show message to user indicating action has been taken. */
IF NOT lDeleteProfileEntry THEN
    /* "Saving browser settings succeeded!" */
    ASSIGN cMessage = {aferrortxt.i 'BR' '3'}.
ELSE
    /* "Resetting browser settings succeeded!" */
    ASSIGN cMessage = {aferrortxt.i 'BR' '4'}.

RUN showMessages IN gshSessionManager (
    INPUT cMessage,
    INPUT "INF",
    INPUT "OK",
    INPUT "OK",
    INPUT "",
    INPUT "Browser Settings",
    INPUT NO,
    INPUT ?,
    OUTPUT cButtonPressed ).
```

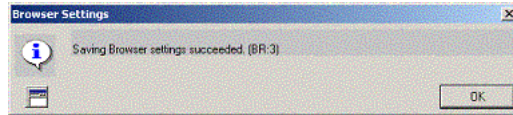
The syntax for **showMessages** is as follows:

```
RUN showMessages IN gshSessionManager
( INPUT  pcMessageList
  INPUT  pcMessageType
  INPUT  pcButtonList
  INPUT  pcDefaultButton
  INPUT  pcCancelButton
  INPUT  pcMessageTitle
  INPUT  plDisplayEmpty
  INPUT  phContainer
  OUTPUT pcButtonPressed).
```

These are the parameters for showMessages:

- **INPUT pcMessageList (CHARACTER)** — This is the specially-formatted message to display. This can also be a CHR(3)-delimited list of messages.
- **INPUT pcMessageType (CHARACTER)** — This can be any of the various message types except for the special Question type, including Message (MES), Information (INF), Warnings (WAR), Errors (ERR), Serious Halt Errors (HAL) and About Window (ABO).
- **INPUT pcButtonList (CHARACTER)** — This is a list of the buttons that should appear in the message dialog.
- **INPUT pcDefaultButton (CHARACTER)** — This is the default button to fire if the user presses Return instead of selecting a button.
- **INPUT pcCancelButton (CHARACTER)** — This is the button to fire if the user presses Escape instead of selecting a button.
- **INPUT pcMessageTitle (CHARACTER)** — This optional argument is the title that should appear in the message dialog.
- **INPUT plDisplayEmpty (LOGICAL)** — This flag indicates whether the message dialog should be displayed even if the message text is empty.
- **INPUT phContainer (HANDLE)** — This is the handle of the container the message dialog is running from.
- **OUTPUT pcButtonPressed (CHARACTER)** — This returns the label of the button the user pressed.

Our sample code results in the following message dialog box:



Changing the Browser's COLUMN-MOVABLE attribute

The procedure **eventMenuItemValueChanged** is called if the user changes one of the two check-marked menu items (Movable and Sortable). It sets the browse widget's COLUMN-MOVABLE attribute accordingly. Setting COLUMN-MOVABLE to TRUE allows you to move browse columns by selecting the header, and disables automatic column sorting:

```
/* Set COLUMN-MOVABLE off if the Menu Item was "sortable" */
  lValue = hMenuItem:CHECKED.
  IF ( hMenuItem:NAME = "ColsSortable" ) THEN
    lValue = NOT lValue.
    {set BrowseColumnsMovable lValue}.
    hBrowse:COLUMN-MOVABLE = lValue.
```

Setting the popup menu's state on menu drop

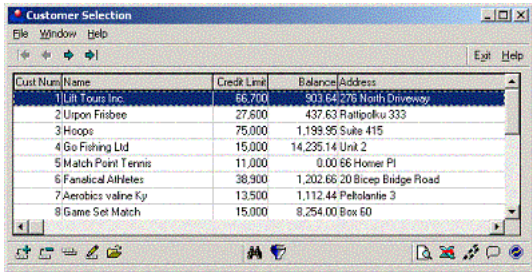
The **eventPopupMenuMenuDrop** procedure is called whenever the user brings up the popup menu by right clicking on the browse. It checks the value of the **BrowseColumnsMovable** property and sets the two check-marked menu items accordingly:

```
{get BrowseColumnsMovable lMovable}.
IF ( hMenuItem:NAME = "ColsMovable" ) THEN
  hMenuItem:CHECKED = lMovable.
ELSE IF ( hMenuItem:NAME = "ColsSortable" ) THEN
  hMenuItem:CHECKED = NOT lMovable.
```

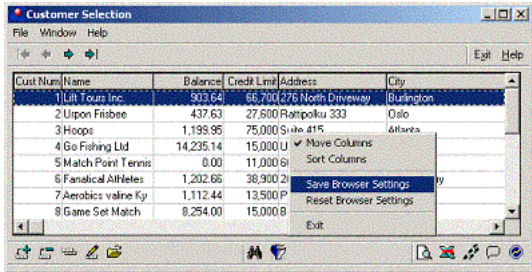
Testing the extended browser profile data

Follow these steps to see what the effect of this custom code is:

- 1 ♦ Recompile the procedure **ry/obj/rydynbrowb.w**, the framework’s dynamic Browser. Run any container with a dynamic Browser in it, for instance the Customer browse window:



- 2 ♦ Bring up the popup menu. You can sort on any column by selecting its header. You can resize columns and the new widths will be saved. You can select the Move Columns menu item, grab a column by its header, and move it to a new position:



- 3 ♦ Select Save Browser Settings from the menu to save your changes to the repository so the container looks the same the next time you run it.

6.7 Using the Localization Manager

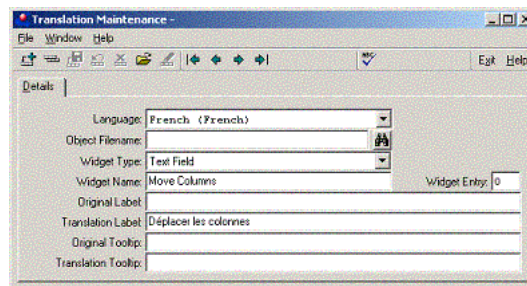
The Localization Manager handles text translations and other localization support. [Progress Dynamics Installation Guide](#) has a section on translating the labels on a window, which introduces the User Interface that makes use of the Localization Manager. In addition, the message maintenance utility lets you translate messages such as the ones you just created into other languages.

This section introduces you to a few of the API calls you can use to adapt the Localization Manager for more specialized needs. Specifically, our popup menu example has one weakness where localization is concerned: the popup menu is not a standard Progress Dynamics object, and its labels are hard-coded in the `browsercustom.p` file, which prevents it from being translated. So in this section you define translations for the popup menu item labels and then access them through the Localization Manager API.

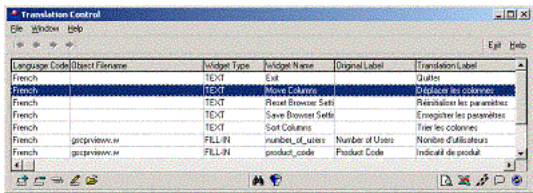
6.7.1 Storing translated text strings in the Repository

Follow these steps to create the translations:

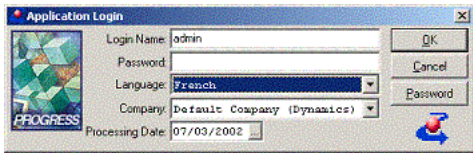
- 1 ♦ Select **Application→Language Control** in the Progress Dynamics Administration's Application menu to create a new language. If you have been through the tutorial or have defined multiple languages for any other purpose, you already have languages other than English defined in the repository.
- 2 ♦ Select **Application→Translation Control** from the same Application menu.
- 3 ♦ Press **Add** in the Translation Control browse window and add translations for each of the text strings in the popup menu.
- 4 ♦ To translate text strings not associated with a specific Progress Dynamics widget type, select the language you are translating into.
- 5 ♦ Select **Text Field** as the Widget Type, and enter the base string (in the original language) in the Widget Name field.
- 6 ♦ Enter the translated text in the Translation Label field:



- 7 ♦ Save this and enter four more text translations for Sort Columns, Save Browser Settings, Reset Browser Settings, and Exit. When you are done, exit the maintenance window to see your translations in the Translation Control browse:



- 8 ♦ Now select **File→Re-Login** from the AppBuilder menu and log back in using your Login Name, but with the Language set to French:



6.7.2 Using the API calls `translatePhrase` and `getTranslation`

Use the **`translatePhrase`** function call to access a text translation. This function returns translations for base text strings one at a time:

```
DYNAMIC-FUNCTION("translatePhrase":U IN gshTranslationManager,  
    INPUT pcText,  
    INPUT pdLanguageObj )
```

The `translatePhrase` function takes these parameters:

- **INPUT `pcText` (CHARACTER)** — The base text string (in the original language) that you want to have translated.
- **INPUT `pdLanguageObj` (DECIMAL)** — The Object ID key of the Language table record for the language to be translated into. You can retrieve this Object ID, if you need it, using the **`currentLanguageObj`** property, which the Session Manager maintains and can return to you with the **`getPropertyList`** API call. However, if you want the text translated into whatever the current language is, you can simply pass in 0 for this parameter.
- **RETURNS CHARACTER** — The function returns the translated string.

The `translatePhrase` function is basically a simplified version of the **getTranslation** call, which you can use when you need more than text strings translated:

```
RUN getTranslation IN gshTranslationManager
( INPUT pdLanguageObj,
  INPUT pcObjectName,
  INPUT pcWidgetType,
  INPUT pcWidgetName,
  INPUT pdWidgetEntry,
  OUTPUT pcOriginalLabel,
  OUTPUT pcTranslatedLabel,
  OUTPUT pcOriginalTooltip,
  OUTPUT pcTranslatedTooltip )
```

The `getTranslation` procedure takes these parameters:

- **INPUT pdLanguageObj (DECIMAL)** — The Object ID of the Language table record identifying the target language. The default is 0 for the current login language.
- **INPUT pcObjectName (CHARACTER)** — The name of the application object being translated. This can be an actual object name or it can be blank, in which case the translation that applies to *all* objects of the widget type (in effect the default value) is returned. This field should also be blank if you're retrieving a text translation.
- **INPUT pcWidgetType (CHARACTER)** — This can be one of the following strings: `title`, `browse`, `fill-in`, `radio-set`, `text`, `button`, `toggle-box`, `combo-box`, `slider`, or `editor`. Setting the widget type and widget name both to `title` is a special case for window title translation.
- **INPUT pcWidgetName (CHARACTER)** — This should be a window title, a text value, or a widget label.
- **INPUT pdWidgetEntry (INTEGER)** — The widget entry number is used for radio set items.
- **OUTPUT pcOriginalLabel (CHARACTER)** — The widget label in the original language is returned in this OUTPUT parameter.
- **OUTPUT pcTranslatedLabel (CHARACTER)** — The translated widget label is returned in this OUTPUT parameter. Text translations are also returned in this parameter.

- **OUTPUT pcOriginalTooltip (CHARACTER)** — If this is a widget that supports Tooltips, the Tooltip in the original language is returned in this OUTPUT parameter.
- **OUTPUT pcTranslatedTooltip (CHARACTER)** — The Tooltip translation, if any, is returned in this OUTPUT parameter.

6.7.3 Using the multiTranslation call

If you have a large number of strings that you want to translate in a single call, use the **multiTranslation** API call:

```
RUN multiTranslation IN gshTranslationManager
( INPUT plAllLanguages,
  INPUT-OUTPUT TABLE ttTranslate ).
```

The multiTranslation call takes two parameters:

- **INPUT plAllLanguages (LOGICAL)** — TRUE if records should be returned for every language an input string has been translated into, otherwise FALSE. If TRUE, then extra records are created in the temp-table for each language available. The temp-table must in that case initially contain entries with a language Object ID of 0. These are deleted in the temp-table that is returned, which instead contains a record for the translation into each language.
- **INPUT-OUTPUT TABLE ttTranslate** — The **ttTranslate** temp-table is defined in the include file **af/app/afitttranslate.i**. This temp-table definition is used by the Localization Manager itself, and you can define a temp-table based on it as well for calls such as multiTranslation.

These are the fields in the table:

```

/* ***** Main Block ***** */
&IF DEFINED(ttTranslate) = 0 &THEN
  DEFINE TEMP-TABLE ttTranslate NO-UNDO RCODE-INFORMATION
  FIELD dLanguageObj      AS DECIMAL
    /* language object or 0 for login language */
  FIELD cLanguageName     AS CHARACTER FORMAT "X(20)":U LABEL "Language":U
    /* language name if known */
  FIELD cObjectName       AS CHARACTER FORMAT "X(40)":U LABEL "Object Name":U
    /* object name or blank for all */
  FIELD lGlobal           AS LOGICAL  FORMAT "YES/NO":U LABEL "Global":U
    /* yes = global translation, no = specific object (if not blank) */
  FIELD cWidgetType       AS CHARACTER FORMAT "X(20)":U LABEL "Widget Type":U
    /* widget type, e.g. text, button, etc. */
  FIELD cWidgetName       AS CHARACTER FORMAT "X(40)":U LABEL "Widget Name":U
    /* widget name or if type is text, text to translate */
  FIELD hWidgetHandle     AS HANDLE
    /* handle of widget if known / required */
  FIELD iWidgetEntry      AS INTEGER  FORMAT "9":U  LABEL "Element":U
    /* widget entry, used for radio-sets, etc. */
  FIELD lDelete           AS LOGICAL  FORMAT "YES/NO":U LABEL "Delete":U
    /* yes = global translation, no = specific object (if not blank) */
  FIELD cTranslatedLabel  AS CHARACTER FORMAT "X(30)":U
    LABEL "Translated Label":U /* translated label */
  FIELD cOriginalLabel    AS CHARACTER FORMAT "X(30)":U LABEL "Original
Label":U
    /* original untranslated label */
  FIELD cTranslatedTooltip AS CHARACTER FORMAT "X(40)":U
    LABEL "Translated Tooltip":U /* translated tooltip */
  FIELD cOriginalTooltip  AS CHARACTER FORMAT "X(40)":U
    LABEL "Original Tooltip":U /* original untranslated tooltip */
  INDEX key1 AS UNIQUE PRIMARY dLanguageObj cObjectName cWidgetType
cWidgetName
    hWidgetHandle iWidgetEntry
  INDEX key2 cLanguageName cObjectName cWidgetType cWidgetName iWidgetEntry
  INDEX key3 cObjectName cWidgetType cWidgetName iWidgetEntry
  INDEX key4 cWidgetName iWidgetEntry cObjectName cWidgetType
  .
  &GLOBAL-DEFINE ttTranslate
&ENDIF

```

These fields should be familiar to you if you have seen the parameter list for getTranslation.

To use the Translation temp-table, first define it in the custom super procedure and then populate it. Follow these steps:

- 1 ♦ Include the definition in the Definitions section of **browsercustom.p**:

```
{af/app/afttttranslate.i}
```

- 2 ♦ Populate the table in **createBrowsePopupMenu**, to pass it in to **multiTranslation**. Define a variable with the text strings you need, and a counter variable to walk through them:

```
DEFINE VARIABLE cStrings          AS CHARACTER NO-UNDO  
    INIT "Move Columns,Sort Columns,Save Browser Settings,Reset  
Browser Settings,Exit".  
DEFINE VARIABLE iString           AS INTEGER    NO-UNDO.
```

- 3 ♦ Populate the temp-table with records for each of the strings needing translation:

```
DO iString = 1 TO NUM-ENTRIES(cStrings):  
    CREATE ttTranslate.  
    ASSIGN ttTranslate.dLanguageObj = 0    /* Use the login language */  
        ttTranslate.cObjectName = ""  
        ttTranslate.lGlobal = NO  
        ttTranslate.cWidgetType = "TEXT"  
        ttTranslate.cWidgetName = ENTRY(iString, cStrings).  
END.
```

- 4 ♦ Pass this to the API call:

```
RUN multiTranslation IN gshTranslationManager  
    (INPUT NO,    /* Not for all languages */  
     INPUT-OUTPUT TABLE ttTranslate).
```

The same temp-table is returned, with the `cTranslatedLabel` field filled with the translation. If the current language is the original language or there is no translation for an item, then nothing is returned in the `cTranslatedLabel` field. Therefore you need to check for this and use the original text string in that case:

```
/* If the current language is the original language or one for which
   there are no translations, then cTranslatedLabel is not set, so
   fall back on the original text. */
FOR EACH ttTranslate:
    IF ttTranslate.cTranslatedLabel = "" THEN
        ttTranslate.cTranslatedLabel = cWidgetName.
    END.
```

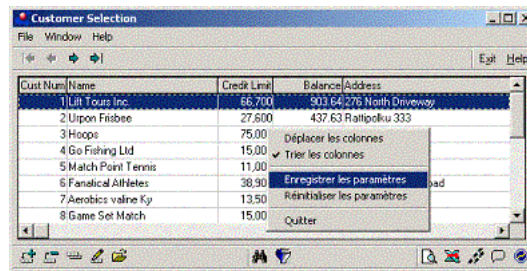
- 5 ♦ Retrieve the appropriate temp-table record for each of the menu items and use the translation for the label. Note that they will not (likely) come back in the order you created them in, because there's no sequence assigned:

```
FIND ttTranslate WHERE ttTranslate.cWidgetName = "Move Columns".
CREATE MENU-ITEM hColsMovableItem
    ASSIGN PARENT      = hPopupMenu
    LABEL      = ttTranslate.cTranslatedLabel /* "&Move Columns" */
    NAME       = "ColsMovable"
    TOGGLE-BOX = TRUE
    CHECKED    = FALSE.
```

- 6 ♦ After you are done using the temp-table, empty it:

```
EMPTY TEMP-TABLE ttTranslate.
```

- 7 ♦ Save and compile the modifications to `browsercustom.p`, delete the current running instance of the super procedure in the Procedures ProTool, and return your test window. Now you see the translated strings, because you are logged in as a French user:



The Localization Manager itself uses `translatePhrase` and the other API calls here to translate titles, labels, and substitution arguments in messages. If you define text translations for these types of strings in your application, they will be translated for you in all the standard places where the framework handles translations. You can use the same calls as you did here to translate other text that the framework does not take care of for you.

6.8 Using the Security Manager

The Security Manager supports the tools under the Security menu in the Administration menu window, and applies security restrictions to an application based on the definitions you create in those tools. See the *Progress Dynamics Developer's Guide* chapter on “Using the Toolbar and Menu Designer” for more information on using tools.

This section briefly describes how you can use the Manager's API in your own application code.

The API includes calls to change a user's password (**`changePassword`**), validate a user for a given login ID, password, and company (**`checkUser`**), and other useful calls that you can make from your own applications. In addition, there is a set of calls for the various types of security allocations that we described in the *Progress Dynamics Developer's Guide*: field-level, data-level, access-token, data-range, etc. You can use these calls from any existing application to take advantage of both the login and user validation mechanism in the framework, and also to define security checks in existing application code based on the data structure and supporting tools in Progress Dynamics. You can also use these calls to make security checks in places where the framework does not automatically do it for you.

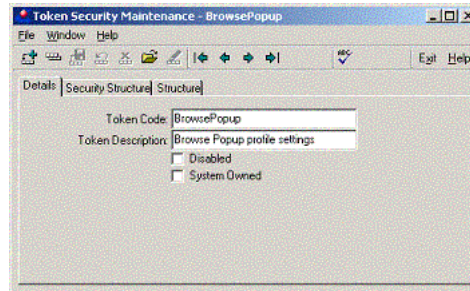
To see how this is done, you can add a security check to the browser custom super procedure you have been working on. This check allows the system administrator to disable the use and setting of browse profile settings on a per-user basis.

To add a security check, follow these steps:

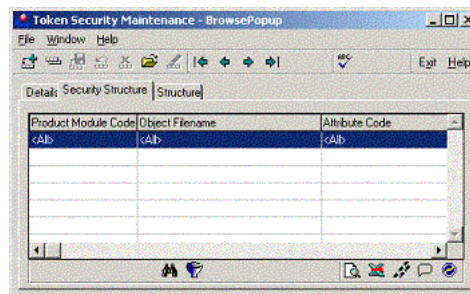
- 1 ♦ Define a security token to represent the use of profiles and the browse popup menu, and then apply a restriction to a user based on that token.
- 2 ♦ Select **Token Security Control** from the Security menu:



- 3 ♦ Press the Add button, define a Token Code of BrowsePopup, and give it a Token Description. When you define the Description, keep in mind that this is the field that the Security Allocation tool sorts on to display all the tokens for you, so give it text beginning with Browse Popup, so that you can locate it easily:



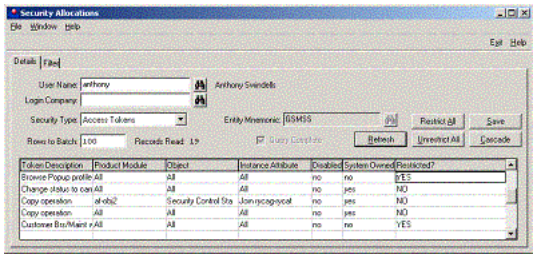
- 4 ♦ Save this new token. Select the Security Structure tab just to remind yourself that, by default, a security token is defined for all Product Modules, all Objects, and for any possible Attribute Code that might be passed in to the object:



NOTE: You can create more specific security structures to apply restrictions to in the Structure tab. For example, you can restrict the use of the popup menu for only certain windows or certain products.

- 5 ♦ Define a restriction for a user based on this token.

6 ♦ Open the Security Allocation window from the Security menu:



7 ♦ Enter a user name.

8 ♦ Select Access Tokens as the Security Type, and press Refresh. All access tokens are displayed. Locate the one for the BrowsePopup token, and double-click on the “Restricted?” field to set it to YES.

9 ♦ Save and exit.

10 ♦ Add code to the browser custom procedure to check for this code. The call you use is **tokenSecurityCheck**:

```
RUN tokenSecurityCheck IN gshSecurityManager
  (INPUT pcObjectName,
   INPUT pcAttributeCode,
   OUTPUT pcSecurityOptions).
```

It takes these parameters:

- **INPUT pcObjectName (CHARACTER)** — The current container name (with no path, if it’s a physical file name).
- **INPUT pcAttributeCode (CHARACTER)** — The runtime attribute code, if any, passed in to the container.
- **OUTPUT pcSecurityOptions (CHARACTER)** — A comma-separated list of security tokens for which the current user is restricted.

In a case such as this one, where the token is defined for all objects, it would make sense to pass in a blank value for the pcObjectName parameter. However, the procedure will not return anything if you do this, so the application code uses the container name that is already available.

- 11 ♦ In `browsercustom.p`, go into the `initializeObject` procedure and define a variable called `cSecurityObjects`. The existing code already retrieves the `LogicalObjectName` of the Browser's container, so we can use that value for our call as well:

```
/* Get the name of the container we're in, which is needed by both
   the tokenSecurityCheck and the profile data key. */
{get ContainerSource hContainerSource}.
IF VALID-HANDLE( hContainerSource ) THEN
  {get LogicalObjectName cContainerName hContainerSource} NO-ERROR.
ELSE
  cContainerName = "".
```

- 12 ♦ Add a call to **tokenSecurityCheck**, passing in the container name, and getting the token list back. Check for the "BrowsePopup" token in the list, and if it is there, just RETURN:

```
/* Example of using the Security Manager. If the token BrowsePopup
   comes back for this user, then skip all the profile code. */
RUN tokenSecurityCheck IN gshSecurityManager
  (INPUT cContainerName, /* pcObjectName */
   INPUT "", /* pcAttributeCode */
   OUTPUT cSecurityObjects).
IF LOOKUP("BrowsePopup", cSecurityObjects) NE 0 THEN
  RETURN.
```

Note that the way this is coded, not only does the popup menu not get created, but any existing profile settings are not read out of the repository either. You could of course make this code work any way you wanted.

- 13 ♦ Save and compile this change, delete any current running instance of the `browsercustom.p` procedure, re-logon as the user you defined the restriction for, rerun your test window, and the popup menu should not be active. Also, any previously defined profile settings for the browse will not be applied.

6.9 Using the General Manager

As its name implies, the General Manager has API calls to support a number of different kinds of general-purpose requirements. These are summarized in the overview section at the top of this chapter.

Use the **getEntityDescription** procedure to retrieve a field value from any record of any table, given its key. This is a valuable way to get needed values from the repository database without embedding hard-coded references to the repository table structures into your code. It also handles the client-server division for you transparently. Refer to the [Progress Dynamics Managers API Reference](#) for details on every call available in the General Manager.

There are other useful API calls provided by the General Manager. The first is **getRecordDetail**. Use this call to pass any query to the Manager, and get back a list of field names and values for the first record satisfying the query. The call is useful for queries that return just a single record, the equivalent of a unique FIND:

```
RUN getRecordDetail IN gshGenManager
    (INPUT pcQuery,
     OUTPUT pcFieldList ).
```

The procedure takes these parameters:

- **INPUT pcQuery** (CHARACTER) — The query string, beginning with FOR EACH followed by the database table name.
- **OUTPUT pcFieldList** (CHARACTER) — A specially formatted list of field names and values for the matching record. What is returned is a CHR(3)-delimited list of <table.field> name and value pairs. In other words, each field name (qualified by the table name) is followed by CHR(3), followed by the field's formatted string value, followed by another CHR(3), and so on. The record's database ROWID is also returned, at the end of the list, in the format ROWID(<table>) CHR(3) <table ROWID>.

Here is a test procedure for getRecordDetail. It includes globals.i, so that the Manager handle is available, defines a query to retrieve the first Customer record, and displays the results:

```
/* testRecordDetail.p -- test procedure for the getRecordDetail procedure
   in the General Manager API. */

{src/adm2/globals.i}

DEFINE VARIABLE pcQuery    AS CHARACTER NO-UNDO.
DEFINE VARIABLE cFieldList AS CHARACTER NO-UNDO.
DEFINE VARIABLE iField     AS INTEGER   NO-UNDO.

    pcQuery = 'FOR EACH Customer where CustNum = 1'.

    RUN getRecordDetail IN gshGenManager
        (INPUT pcQuery,
         OUTPUT cFieldList ).

    DO iField = 1 TO NUM-ENTRIES(cFieldList, CHR(3)) BY 2:
        DISPLAY ENTRY(iField, cFieldList, CHR(3))    FORMAT "x(30)"
            ENTRY(iField + 1, cFieldList, CHR(3))    FORMAT "x(40)"
            WITH FRAME F 21 DOWN.
        DOWN WITH FRAME F.
    END.
```

Figure 6–8 shows what the procedure's output looks like.

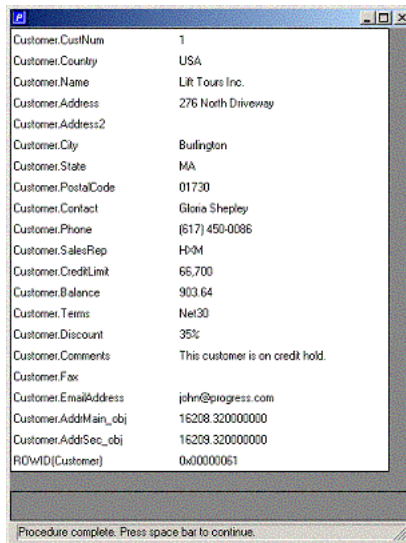


Figure 6–8: Output of testRecordDetail.p

Note the ROWID value at the end of the list. You can apply the Progress TO-ROWID function to this string to turn it back into a proper RowID datatype.

As another example of how this call could be useful, consider the translatePhrase function described earlier. One of the parameters is the language Object ID for the language to translate into. If what you want is the current language, you just pass in 0. But if you want some other language, you can use getRecordDetail, to retrieve the language_obj field for the language if you know the Where clause to construct to retrieve the record you want. In this case:

```
FOR EACH gsc_language Where language_code = "French"
```

This query yields the results shown in [Figure 6–9](#). The first field value is the language Object ID, which you can convert to native DECIMAL form and pass in to another call that needs it.

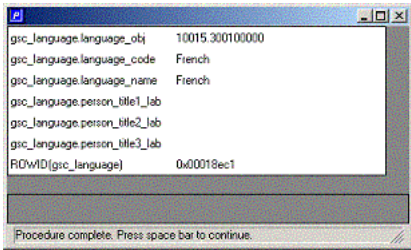


Figure 6–9: FOR EACH query results

Keep in mind that because this is a general-purpose call, no caching of information is possible. Every query will result in an AppServer call in a distributed application. But also keep in mind that the call shields you from having to worry about the code needed to access the data on the AppServer, as long as the database you need is connected on the default AppServer whose handle is in gshAstraAppServer.

This series of examples should give you a good idea of how to take advantage of the Progress Dynamics Managers to extend their default behavior to satisfy the needs of you application, and to help you integrate existing applications into a new Progress Dynamics application.

Creating a New Manager In Progress Dynamics

A Progress Dynamics Manager is a structured business logic procedure (or PLIP) that has both a client and a server part. The term Manager here refers to a procedure with a few special characteristics:

- It operates on both sides of the AppServer connection, and has both a client part and a server part.
- It is normally registered as part of the session type, and pre-started so that it is available to other procedures in the session when needed.
- It has an API that can be called from elsewhere in the application.

Beyond this, a Manager can do just about anything at all.

The server side of the Manager controls database access, and the client side coordinates access to the Manager's API. The client Manager or Client Proxy cannot access database statements directly, and acts as a gateway to calls that get information from the server. You can cache data on one or both sides of the AppServer connection, depending on the nature of the Manager.

Progress Dynamics comes with a number of built-in Managers. You can find more information on these Managers in [Chapter 6, "Using the Progress Dynamics Managers,"](#) which briefly describes the architecture of the existing Managers. However, before Version 2 there was no specific template for a new Manager. The task of building a Manager has been streamlined to make creating a new Manager easier since the existing Version 1.1 Managers were built, so the details of construction are different. This is why the description of the existing Managers does not exactly match the use of the new Version 2 Manager template.

This chapter describes the new template procedures and how you can use them to create Managers of your own for your application. It covers the following topics:

- [The manager templates](#)
- [Building a new manager in the AppBuilder](#)
- [Registering your manager](#)
- [Creating a new Field Edit manager](#)
- [Customizing an existing manager](#)

7.1 The manager templates

The Progress Dynamics Manager has one basic characteristic that distinguishes it from other business logic procedures: it runs on both sides of the AppServer connection. In this way it shares one key characteristic of the SmartDataObject, which enables you to write code for it in the AppBuilder just as you do for SDOs or SDO logic procedures in a dynamic application. The template for new Managers has the DB-AWARE flag set on, to tell the AppBuilder to allow you to write code into this single procedure file and to designate whether that code should be compiled and executed for the client, for the server, or for both.

The DB-AWARE flag is first of all a preprocessor value in the template:

```
/* ***** Preprocessor Definitions ***** */
&Scoped-define PROCEDURE-TYPE Procedure
&Scoped-define DB-AWARE YES
```

You can also set the DB-AWARE flag in special Procedure Settings comments that the AppBuilder parses when it reads in a file for editing. The preprocessor allows your code to reference the DB-AWARE flag, and this internal setting prevents the value from being changed, which would disable the Manager:

```
/* ***** Procedure Settings ***** */
&ANALYZE-SUSPEND _PROCEDURE-SETTINGS
/* Settings for THIS-PROCEDURE
  Type: Procedure
  Allow:
  Frames: 0
  Add Fields to: Neither
  Other Settings: CODE-ONLY APPSERVER DB-AWARE NO-PROXY
*/
&ANALYZE-RESUME _END-PROCEDURE-SETTINGS
```

In addition to the DB-AWARE setting, there is a setting for the AppBuilder's information called NO-PROXY. This setting tells the AppBuilder that, unlike for an SDO, the AppBuilder should not actually generate the client proxy when you save the Manager procedure.

There are several reasons for this:

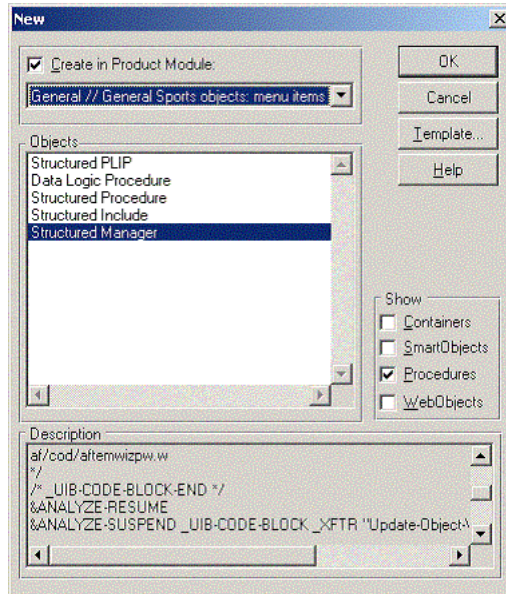
- First, in the case of the SDO (or the similarly constructed SBO), the client proxy is always given a filename that is the same as the base procedure, but with the extension `_cl`. The external request runs the base procedure name, and the internal logic of the SDO (or, more precisely, the `constructObject` call in its Container code) determines whether in fact it is that server-side procedure or the client proxy that should be run, depending on the session configuration. The client proxy is never run explicitly. This naming convention is not appropriate for Managers, where either the client procedure or the server procedure is explicitly registered with the Session Type as the proper procedure to run when the session starts up. Therefore you can give the client procedure any name you want. For the sake of consistency Progress recommends that you follow the existing Progress Dynamics convention of using a common prefix followed by `srvrp.p` for the server-side procedure and `clntp.p` for the client side, but there is no requirement that you follow this pattern.
- Second, all the parts of an SDO, including the base procedure (customerfollow, for example), its companion include file that defines the SDO temp-table, the client proxy procedure, and the logic procedure where you write validation logic, are considered a single unit in Progress Dynamics. Only the base procedure is registered in the repository when you save the SDO. This is not the case for Managers, because either the client or the server version of the Manager is run for a given session type, and therefore both the client procedure and the server procedure need to be individually registered.
- Third, if you do not want to save the client procedure to the same directory as the server procedure, the AppBuilder is not set up to save different procedures to different directories as part of a single operation. Progress Dynamics uses a convention of saving only procedures in the framework that will run on the AppServer in the app directory.
- Fourth, the client procedure normally never changes once it has been generated, because it is really just two lines of code: one line sets a flag called DB-REQUIRED that tells the compiler to compile out database references, and a second line includes the server procedure. All the real code for both sides is in the server procedure; only the DB-REQUIRED flag tells the compiler which code to compile for which version.

So for all these reasons, the AppBuilder does not generate or manage the client procedure for you. You can create the client procedure yourself by copying the template for it and naming the server file name that it should include.

The two template procedures that you use to build new Managers are `ry/app/rytemsrvrp.p`, and `ry/obj/rytemclntp.p`. The first is opened for you when you create a new Manager in the AppBuilder, and the second you copy and edit yourself.

7.2 Building a new manager in the AppBuilder

To create a new Manager, select **New→Structured Manager** in the AppBuilder:



In the New Manager Wizard, set the desired information to identify the procedure, and you are ready to code.

7.2.1 Separating client code from server code

Programming a Manager is like programming any other business logic procedure, except that you write code for both sides of the server connection.

When you create a new procedure in the Section Editor, shown in [Figure 7-1](#), you see a DB-Required toggle in the header section that you can use to indicate whether the code you write goes into the client or server side when the procedure is compiled.

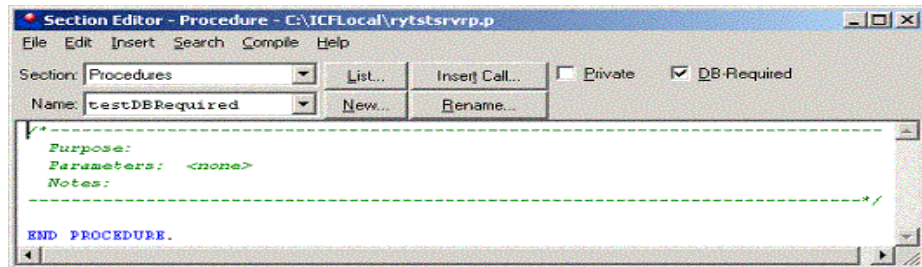


Figure 7-1: Section Editor

By default, the DB-Required toggle is checked on. When it is on, the entire internal procedure or function you are writing compiles into only the server side .r file for the Manager. If you turn it off, it goes into both the client and server files. The description below describes how it works.

When you save your procedure, the AppBuilder writes these additional preprocessor definitions into your source procedure:

```
/* Db-Required definitions. */
&IF DEFINED(DB-REQUIRED) = 0 &THEN
    &GLOBAL-DEFINE DB-REQUIRED TRUE
&ENDIF
&GLOBAL-DEFINE DB-REQUIRED-START    &IF {&DB-REQUIRED} &THEN
&GLOBAL-DEFINE DB-REQUIRED-END      &ENDIF
```

The first of these definitions says that if the DB-REQUIRED preprocessor has not already been defined, then define it to be TRUE. The other two definitions provide for a wrapper around each entry point in the file. Around every function and internal procedure definition you create, the AppBuilder puts the DB-REQUIRED-START and DB-REQUIRED-END preprocessors that compile the entry point in or out:

```
{&DB-REQUIRED-START}

&IF DEFINED(EXCLUDE-testDBRequired) = 0 &THEN

&ANALYZE-SUSPEND _UIB-CODE-BLOCK _PROCEDURE testDBRequired Procedure
_DB-REQUIRED
PROCEDURE testDBRequired :
/*-----
-
  Purpose:
  Parameters:  <none>
  Notes:
-----*/

END PROCEDURE.

/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME

&ENDIF

{&DB-REQUIRED-END}
```

If you look at the template for the client side procedure, you can see why they compile differently:

```
/* rytstclntp.p - non-db proxy for rytstsrvp.p */  
  
&GLOBAL-DEFINE DB-REQUIRED FALSE  
  
{"rytstsrvp.p"}
```

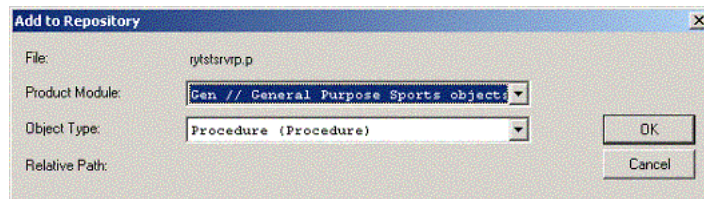
This file sets the DB-REQUIRED flag to FALSE, before the AppBuilder-generated statement in the main procedure has a chance to set it to TRUE. Therefore, all DB-REQUIRED blocks are left out of the compilation that winds up on the client.

7.3 Registering your manager

Once you have created your Manager, you need to register it so that it starts properly and is available for use at run time. First, you need to register the procedure in the Progress Dynamics repository, then create a Manager Type for it, and finally associate that type with one or more Session Types.

To register the Manager procedure in the repository, follow these steps:

- 1 ♦ Open the procedure in the AppBuilder, and select Save To Repository from the File menu:

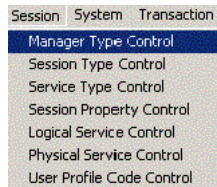


- 2 ♦ Enter a Product Module for the procedure, and make sure its Object Type is set to Procedure.
- 3 ♦ Choose OK to save this information in the repository. Now you can refer to your procedure in the Session utilities.

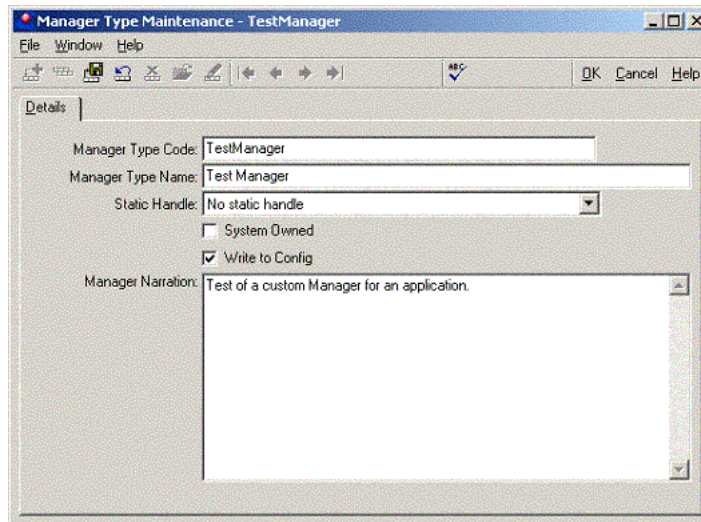
7.3.1 Creating a new manager type

To create a new Manager Type, follow these steps:

- 1 ♦ Select **Manager Type Control** from the Progress Dynamics Administration's Session Menu:



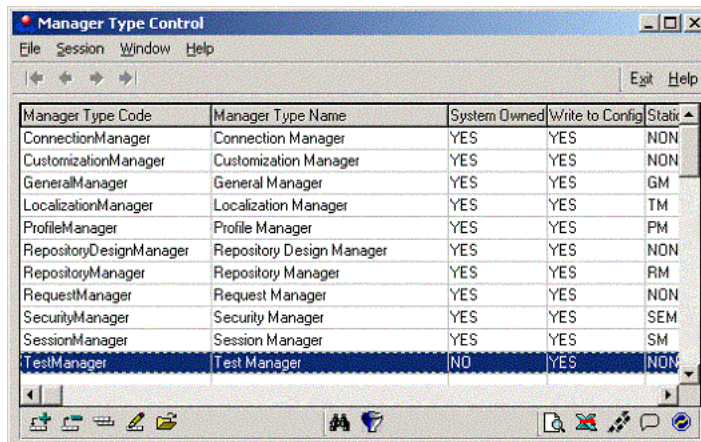
- 2 ♦ Choose **Add** to create a new control. The Manager Type Maintenance window appears:



The Manager Type Maintenance window has the following fields:

- **Manager Type Code** — A single-word identifier used as a key for retrieving the Manager's handle, and to identify it to the Session Manager.
- **Manager Type Name** — Any meaningful descriptive string for the Manager, which appears in messages referring to the Manager. By convention, this is just the Manager Type Code with spaces as appropriate.
- **Static Handle** — The built-in Progress Dynamics Managers have global variables defined for them in the file `src/adm2/globals.i`. Unless your new Manager is going to be accessed with great frequency at different places within your application, which might possibly make it noticeably faster to have immediate access to the handle from anywhere, you should not add a new global handle for it. This would require you to edit `globals.i`, and then recompile every procedure in the entire framework and your application to provide access to it. The handles that are already in `globals.i` are there for efficiency for the basic framework Managers (and not even all of those). Instead, you can use the **getManagerHandle** function in the Session Manager to retrieve the handle of your Manager from anywhere in your application. In this case you set the Static Handle entry to No Static Handle.
- **System Owned** — This toggle restricts modifications to the Manager to those users with System privilege.
- **Write to Config** — If you want the Manager information to be written out to the XML configuration file that holds all the session information, then check the Write to Config toggle on. If you want to be able to associate the Manager with one or more Session Types and have it automatically started for you when those sessions are started, then you must check Write to Config on. If you intend to access the Manager only programmatically once your application session has already started, then you can leave it off.
- **Manager Narration** — This field is for any helpful further explanation of the Manager's purpose.

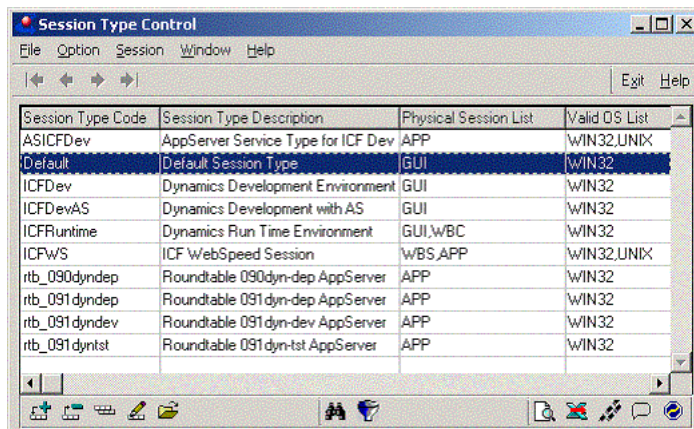
- 3 ♦ Choose Save when you have entered your information, and return to the Manager Type Control window to see the new Manager in the list with all the other Managers:



7.3.2 Adding your manager to the session types

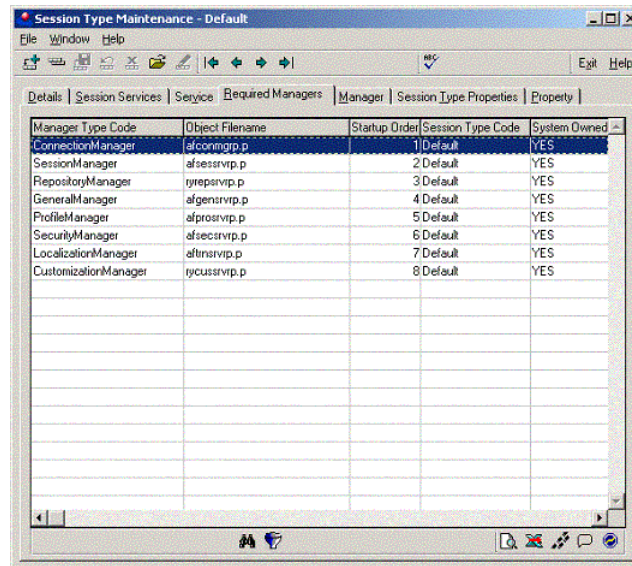
To start your Manager with one of the framework's Session Types, add it to the list of that Session Type's Required Managers by following these steps:

- 1 ♦ Open the **Session Type Control** window from the Session Menu:



2 ♦ You can either:

- Add your Manager to more existing Session Types. To add the Test Manager to the Default Session Type, select the Default Session Type, press the Edit button, and select the Required Managers tab. The list of session Manager procedures appears:



The screenshot shows a window titled "Session Type Maintenance - Default" with a menu bar (File, Window, Help) and a toolbar. Below the toolbar is a tabbed interface with tabs: Details, Session Services, Service, Required Managers, Manager, Session Type Properties, and Property. The "Required Managers" tab is selected, displaying a table with the following data:

Manager Type Code	Object Filename	Startup Order	Session Type Code	System Owned
ConnectionManager	afconnmgr.p	1	Default	YES
SessionManager	afsessrv.p	2	Default	YES
RepositoryManager	tyrepssrv.p	3	Default	YES
GeneralManager	afgenssrv.p	4	Default	YES
ProfileManager	afprossrv.p	5	Default	YES
SecurityManager	afsecssrv.p	6	Default	YES
LocalizationManager	aflocaessrv.p	7	Default	YES
CustomizationManager	tycusssrv.p	8	Default	YES

- Create new Session Types to define different configurations for the client or server.

There are some special things to note about the list of Managers.

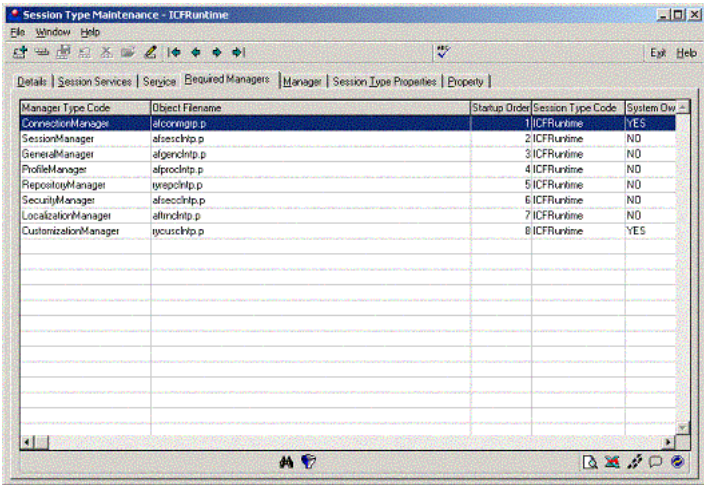
- **Progress Dynamics Session Startup** — The standard startup procedure for any Progress Dynamics session is `icfstart.p`. This procedure runs the procedure `af/app/afxmlcfgp.p`, the Configuration File Manager, which opens and parses the XML configuration file containing all the rest of the information about the session. The Configuration File Manager starts the rest of the Managers. In some cases it may not make a difference which order a Manager is started in, except that the Connection Manager must always be first in the list, because it is responsible for starting any databases and AppServers, and then getting the rest of the Managers started. Ordinarily you simply add a new Manager to the end of the list, so make sure you note the number of Managers that are already associated with your Session Type, and assign yours the next higher sequence number.

- **Manager Procedure Name** — Note the name of the Manager procedure that appears in the list. There is just one version of the Connection Manager, so it will always appear as `afconmgr.p`. The other Managers, however, have both a client part and a server part, which conform to the naming convention `afxxxsvrp.p` and `afxxxclntp.p`, where `xxx` represents a three-letter code for the Manager. A given Session Type runs either on the client or on the server, and must start the appropriate Manager procedures for its side of the connection. A standalone session runs the server-side Managers. Note that this is

different from the way that SDOs and other DB-AWARE objects work. In an SDO you always specify the base name of the object, which runs on the server, and the internals of the `constructObject` code determine whether to run that or the client proxy. The client proxy in turn is responsible for running the server-side object, and maintaining a connection to it.

Because Managers run explicitly on client or server, you need to specify the correct name of the client-side or server-side Manager procedure when you add it to the session. You can see from the list of Managers for the Default session, for example, that because it runs as a standalone session, without AppServer (and therefore without a separate client session), it runs the server versions of the Managers.

A client-side Session Type, by contrast, such as the `ICFRuntime` session, runs the client-side procedures:

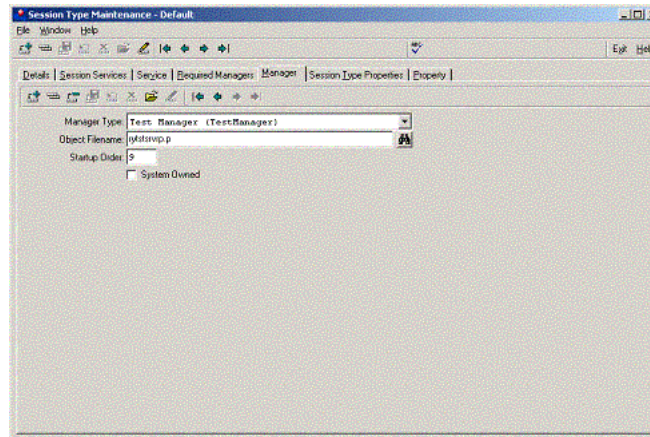


The screenshot shows a window titled "Session Type Maintenance - ICFRuntime". It has a menu bar (File, Window, Help) and a toolbar. Below the toolbar is a tabbed interface with tabs for "Details", "Session Services", "Service", "Required Managers", "Manager", "Session Type Properties", and "Property". The "Manager" tab is selected, displaying a table with the following data:

Manager Type Code	Object Filename	Startup Order	Session Type Code	System Ow
ConnectionManager	afconmgr.p	1	ICFRuntime	YES
SessionManager	afsessclntp.p	2	ICFRuntime	NO
GeneralManager	afgenclntp.p	3	ICFRuntime	NO
ProfileManager	afproclntp.p	4	ICFRuntime	NO
RepositoryManager	afrepsclntp.p	5	ICFRuntime	NO
SecurityManager	afseccclntp.p	6	ICFRuntime	NO
LocalizationManager	afloclntp.p	7	ICFRuntime	NO
CustomizationManager	afcusclntp.p	8	ICFRuntime	YES

Once you have selected the Session Type to add your Manager to, follow these steps to edit it:

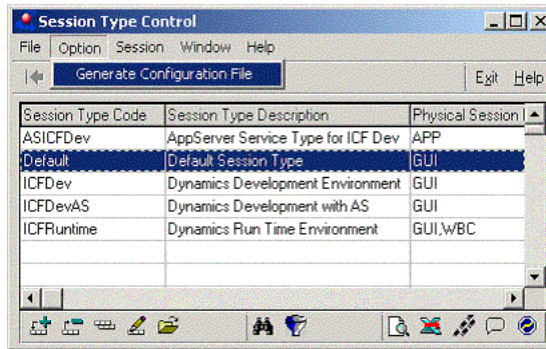
- 1 ♦ Select the **Manager** tab from the Session Type Maintenance folder.
- 2 ♦ Press the Add button **inside the tab folder** to add your Manager to the Session:



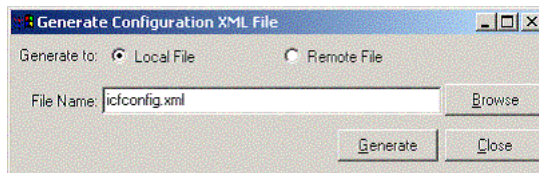
- 3 ♦ Select **Test Manager** from the pre-populated list of Manager Types.
- 4 ♦ Enter the **Object Filename**, which is either the client procedure or the server procedure. Note that you do not have to specify a path to the procedure, because it has already been registered in the repository, so the Configuration File Manager will be able to locate it from its description as stored in the repository.
- 5 ♦ Enter the next available Startup Order sequence for the manager.
- 6 ♦ Check the System Owned toggle **on** if you want only users with System authorization to be able to modify its information.
- 7 ♦ Choose **File→Save** to add your Manager to the list, and close the Session Type Maintenance window.

The description of your Manager is now in the repository, but in order for it to be found when the session starts up, you must add the same information to the XML configuration file that all sessions read in when they start up.

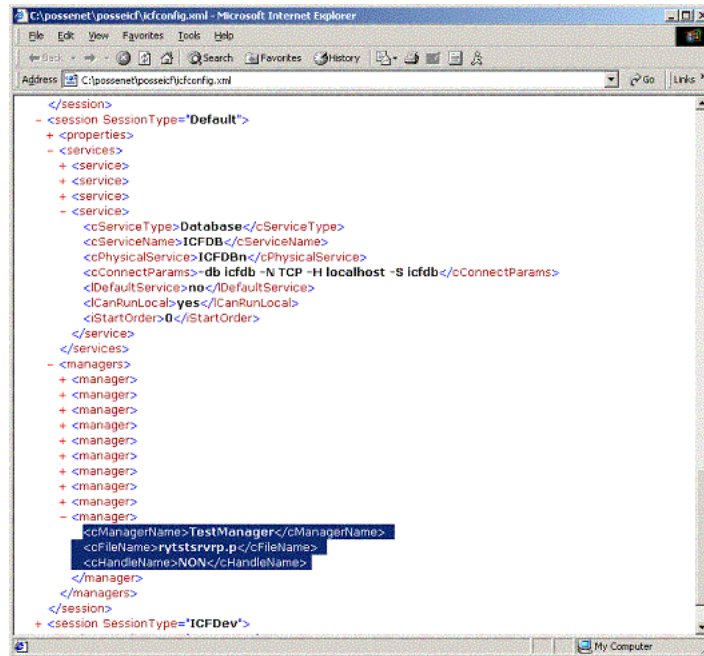
- 8 ♦ Return to the Session Type Control window and select **Option**→**Generate Configuration File** to rebuild the XML file from the repository data:



- 9 ♦ At the prompt, enter the name of the configuration file, and choose either **Local File** or **Remote File**, depending on whether the file is local to your machine or remote. The default file is `icfconfig.xml`, in the `src` directory where your Progress Dynamics code was installed. Move the file to another location in your application's ProPath if you are going to modify it.
- 10 ♦ After setting the filename, choose **Generate**:



The generated configuration file appears:



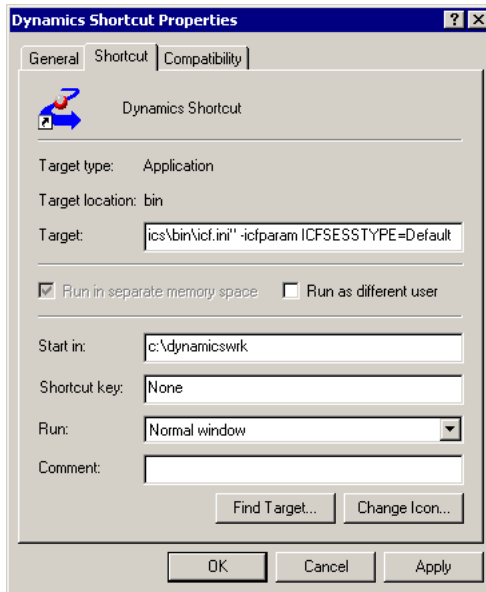
This file contains the information for the Test Manager added to the Default session (a number of nodes in the XML information are not expanded here, so that you only see what has been added).

7.3.3 Invoking the session type in your startup command

To start the Session Type with your Manager in it, specify the Session Type in the startup command line. For instance, you can look at the command line for the Progress Dynamics Default icon:



The Dynamics Default Properties sheet allows you to specify your Session Type parameters:



The startup procedure is `icfstart.p`, and the `-icfparam` startup option must specify the `ICFSESSTYPE` parameter and give it a value of `Default`, or whatever your Session Type is. Then when you open the session, your Manager is started along with others.

In this simple test case, the standard `plipSetup` procedure in the Test Manager has a `MESSAGE` in it to show you that it got started properly:

Procedure `plipSetup`:

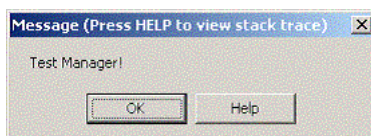
```

/*-----
Purpose:   Run by main-block of PLIP at startup of PLIP
Parameters: <none>
Notes:
-----*/

{ry/app/ryplipsetu.i}
MESSAGE "Test Manager!".
END PROCEDURE.

```

When you start the Default session, the message appears:



7.4 Creating a new Field Edit manager

This section contains an example of how to design, build, and register a new Field Edit Manager.

The Field Edit Manager supports the definition of special kinds of editing and validation for fields in the application's database. It also supports the visualization of these characteristics in the client User Interface. For example, the developer can define fields that are mandatory, independent of the Progress schema definitions, provide a visual cue for those mandatory fields in the Viewers in the application's maintenance windows, and then ensure that those fields have all been entered before an update is returned to the server. You can define a field as requiring upper or lower case letters only, and you can adjust the screen value for the field accordingly before updated values are returned to the server. This example illustrates a number of ways in which you can extend the framework, each of which is described in its own sections:

- [Adding a new table to the Repository](#)
- [Defining the manager itself](#)
- [Registering the new manager](#)
- [Accessing the manager from SDOs](#)
- [Accessing the manager from viewers](#)
- [Invoking the field edits from an SDO](#)
- [Testing the new manager](#)

7.4.1 Adding a new table to the Repository

If your application requires data that cannot easily be represented in the repository database tables, you can add an additional table or tables to store it. Whether you actually add these tables to the repository database itself (ICFDB), or to your application database, is up to you. In either case, you can join the new tables to other repository tables to provide a connection between standard repository data and your extensions.

In this case, you need a new table to store the field edit definitions for the Manager to use. This is an extension to the repository definition of entities (tables) and their fields. When you import entity definitions into the repository as the first step in defining a Progress Dynamics application for your database, the framework creates an Entity Mnemonic record for each table, and an Entity Display Field record for each field in the table, with some of its display characteristics, such as its label and format. The Object Generator uses this data to create the default SDOs, Browsers, and Viewers that make up the starter set of objects in the application.

Figure 7–2 shows these two standard repository tables.

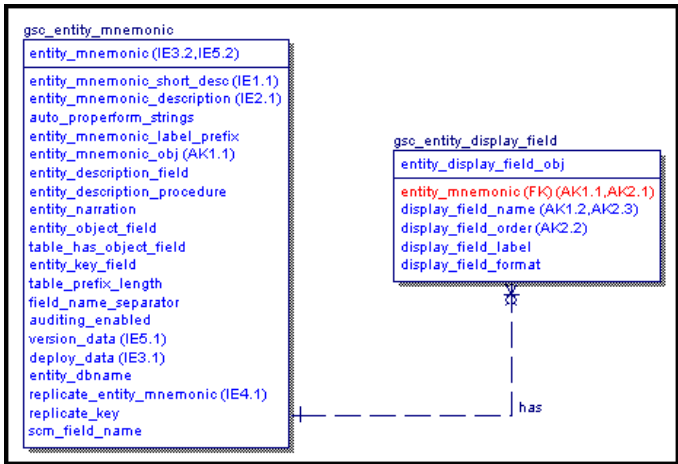


Figure 7–2: Standard Repository tables

To support the Field Edit Manager, you must add an additional table to store field edit information for each field. The table is called `gsc_entity_field_edit`. You can create it in the Progress Data Dictionary. The table contains the fields described in [Table 7–1](#).

Table 7–1: The `gsc_entity_field_edit` Table

Field	Description
<code>entity_field_edit_obj</code>	This DECIMAL field is the Object ID for the record itself.
<code>entity_display_field_obj</code>	This DECIMAL field is the Object ID of the associated <code>gsc_entity_display_field</code> record to join to.
<code>edit_type</code>	This CHARACTER field stores the code for the special edit type, such as Required or Case.
<code>edit_value</code>	This CHARACTER field stores the value for the edit type, such as Yes for Required, or Upper or Lower for Case.

You must have a unique primary index on the `entity_field_edit_obj` field, and a second index on this field plus the `edit_type` field.

When creating the new table, provide the appropriate format for the Object ID fields, with nine decimals to assure that the site ID for the database is properly encoded, as shown in [Figure 7-3](#).

Field Properties

Field Name: Member of an Index: no
 Data Type: decimal Member of a View: no

Optional

Format: Examples...

Label:

Column Label:

Initial Value:

Order #: Decimal: Position:

Description:

Help Text:

☒ Mandatory ☐ Case-Sensitive ☐ Extent 0

Triggers... Validation... View-As... String Attrs... Data Server...

OK Save Cancel < > Help

Figure 7-3: Field Properties window

You need a CREATE trigger procedure to define the code that automatically assigns the next available Object ID to each new record in the table. The trigger procedure must include the standard framework code that defines the getNextObj function where Object IDs are assigned, and then invoke it to assign the ID. The Table Triggers window is shown in [Figure 7-4](#).

Table Triggers

Event: Procedure: Options: ☐ Check CRC ☐ Overridable

TRIGGER PROCEDURE FOR CREATE OF gsc_entity_field_edit.
 (af/sup/aftrigproc.i)

ASSIGN gsc_entity_field_edit.entity_field_edit_obj =
 getNextObj();

Cut Copy Paste Find... < > Replace... Insert File...

Trigger: Check Syntax: ☐ On Save

OK Save Cancel Help

Figure 7-4: Table Triggers dialog box

We won't go into how the field edit data gets into the new table. Presumably, you would create a new tool where developers can define the special edits, but that is beyond the scope of what we are describing here. You can populate the table using something as simple as a procedure to define some test values, such as the following example, which marks the CustNum, Name, City, State, and Country fields in the Sports2000 database Customer table as required fields, and the State field as requiring upper case:

```
FOR EACH gsc_entity_display_field WHERE entity_mnemonic = 'customer':
  IF LOOKUP(display_field_name, 'custnum,name,city,state,country') NE 0 THEN
    DO:
      CREATE gsc_entity_field_edit.
      ASSIGN gsc_entity_field_edit.entity_display_field_obj =
        gsc_entity_display_field.entity_display_field_obj
        gsc_entity_field_edit.edit_type = "Required"
        gsc_entity_field_edit.edit_value = "YES".
    END.
  IF display_field_name = 'State' THEN
    DO:
      CREATE gsc_entity_field_edit.
      ASSIGN gsc_entity_field_edit.entity_display_field_obj =
        gsc_entity_display_field.entity_display_field_obj
        gsc_entity_field_edit.edit_type = "Case"
        gsc_entity_field_edit.edit_value = "Upper".
    END.
  END.
END.
```

7.4.2 Defining the manager itself

In the AppBuilder, select **New**→**Structured Manager** to create the new Manager procedure from its template. Enter the standard file documentation information in the wizard. Name the procedure `fldedsvrp.p`. Note that this server-side Manager procedure in fact contains all the code for both the client and server versions of the Manager. As you write code for the Manager, the DB-REQUIRED flag determines which code gets compiled into which version of the Manager.

To create the client-side Manager procedure, simply copy the client Manager template, which is the file `af/sup2/aftemclntp.p`, to the working directory where you want your Manager. Name your version `fldedclntp.p`. Edit the file to include the server Manager procedure `fldedsvrp.p`, including its relative pathname, depending on your own directory structure:

```
/* fldedclntp.p - non-db proxy for fldedsvrp.p */

&GLOBAL-DEFINE DB-REQUIRED FALSE

{"fldedsvrp.p"}
```

As you can see, the client procedure simply defines the DB-REQUIRED flag to be false and then includes the principal (server) Manager procedure. In this way, the references to DB-REQUIRED inside `fldedsvrp.p` determine which code gets compiled out of the client procedure. Save the client Manager procedure, and compile it when you have finished writing the Manager. All your code now goes into `fldedsvrp.p`.

Data caching considerations

The first procedure to define in the Manager, `cacheFieldEdits`, builds a client-side cache of data from the Entity Field Edit table.

Different kinds of Managers may need to cache different data on either the client or server or both. Keep a few things in mind when you're designing the caching mechanism for a Manager:

- If code that is executed on the server needs quick access to data, it may be worthwhile to cache that data in temp-tables on the server side. This is especially true if the data needs to be massaged in some way as it is read out of whatever underlying database tables or other data sources provide the raw information, and if that data processing is relatively expensive. Caching the data once it has been read and processed makes it available for later access without incurring that overhead again.
- If there is a benefit to avoiding the caching overhead when a request comes in from a client, then the server start-up code may want to cache data in advance. Whether this is **all** the data derived from the underlying database tables (or any other source) or some part of the data is up to you to decide. You need to balance the up-front startup cost of caching the data against the immediate cost of caching the data when the first request comes in that needs that particular data. If the server-side Manager is pre-started as part of an AppServer session, which will be the case if the Manager is made one of the required Managers in the Session Type, then this startup cost may occur when the overall system is started, and the AppServer sessions may continue to run for a long time. In this case, an end user starting a client session or making a request from a client may never see the overhead of the server caching, making the pre-caching an appropriate action.
- If code that is executed on the **client** needs access to data that originates in database tables, the data **must** be passed from the server to the client so that the client does not need a database connection. If the data is likely to be needed again on the client, it makes sense to cache it in temp-tables there so that it's available when needed.
- There may be a benefit on the client as well as the server to pre-caching data. Because the user will normally experience a wait when starting the application due to the overhead of pre-caching data, you need to balance the benefit of having the data immediately available later on against the cost of loading it on startup.

- If data cached on either the client side or the server side may become stale or out-of-date, you need to provide a way to determine when that data needs to be refreshed, and a mechanism to clear the old data from the client cache so that it is re-retrieved from the server, or to clear old data from the server cache so that it can be re-loaded from the database.
- In a stateless environment, any server-side temp-table cache is local to a particular AppServer session, and if the data isn't pre-cached in a uniform way, the data in any AppServer session is simply a by-product of the client requests that happen to come in to that session. Since a client request may be handled by any available AppServer session, you can't expect that data cached by some earlier request is in the cache of the server session that handles the next request. If server-side data is not pre-cached, but it makes sense to cache data on the server at all, then you must simply allow for the fact that each AppServer session gradually builds up a cache reflecting the requests that it has handled. This may make sense if pre-caching is not appropriate, but if the server sessions are gradually able to handle requests more efficiently because of the data cache they build up.

In the case of the example Field Edit Manager, data caching makes sense only on the client. Each time an SDO starts up, it requests the field edit data for its enabled table or tables from the Field Edit Manager. If that data isn't already available, then the client Manager requests it from the server Manager and adds it to the client cache.

The cache temp-table has this definition:

```
DEFINE TEMP-TABLE ttFieldEditData NO-UNDO
  FIELD cEntityName      AS CHARACTER
  FIELD cFieldName       AS CHARACTER
  FIELD cEditType        AS CHARACTER
  FIELD cEditValue       AS CHARACTER
  INDEX key1 AS UNIQUE PRIMARY cEntityName cFieldName cEditType.
```

The Entity Name comes from the Entity Mnemonic table, and the Field Name from the Display Field table. The Edit Type and Edit Value come from the new Field Edit table.

The `cacheFieldEdits` procedure takes a list of one or more tables as input, and checks to see whether the field edit data for the table(s) is already in the cache temp-table. Note that because the procedure will normally be called on the client, the section editor's DB-Required toggle must be checked off, so that the code is compiled into the client version of the Manager, as shown in Figure 7-5.

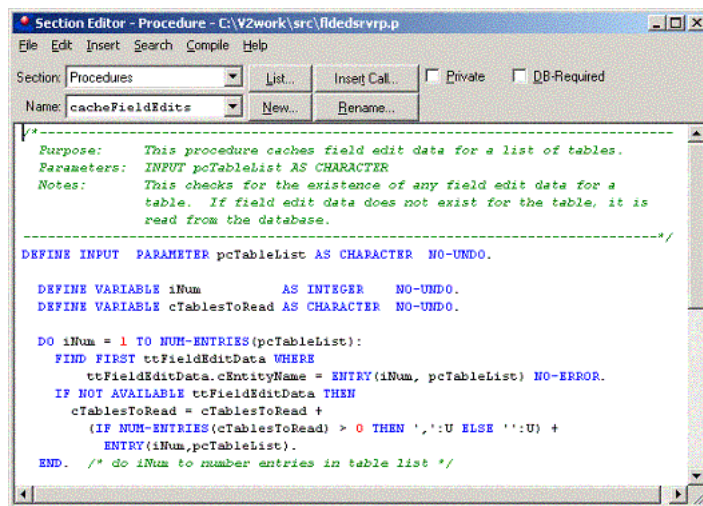


Figure 7-5: Section Editor

If any of the tables are not yet cached, then the code calls a separate procedure that has the DB-Required flag set to TRUE to load them from the repository database. The `cacheFieldEdits` procedure needs to check whether the procedure that reads the database is available to run locally or must be run remotely, so that it knows where to run the database-dependent code. Because the DB-Required procedure is completely compiled out of the client Manager when the Manager is split between client and server, the code can check for the existence of the DB-Required procedure in the Manager's internal entries. If it is not there, then it must be run remotely. This code fragment uses the include file `dynlaunch.i` to make the call, which is explained in the following example:

```
IF cTablesToRead NE ':':U THEN DO:
  IF LOOKUP('fetchFieldEdits', THIS-PROCEDURE:INTERNAL-ENTRIES) NE 0 THEN
    RUN fetchFieldEdits (INPUT pcTableList,
                        OUTPUT TABLE ttNewData).
  ELSE {dynlaunch.i
    &PLIP = 'FieldEditManager'
    &iProc = 'fetchFieldEdits'
    &mode1 = INPUT &parm1 = pcTableList &dataType1 = CHARACTER
    &mode2 = OUTPUT &parm2 = ttNewData &dataType2 = TABLE
  }
```

An alternative way to make this check is to use the statement:

```
IF CONNECTED('ICFDB') THEN
```

This has the disadvantage of hard-coding the logical database name where the data resides into the procedure, which may not be a good idea. If you later move the data (to your application database, for example) then you need to remember to change this statement as well.

Using dynlaunch.i to make server calls in your manager

The include file `dynlaunch.i` supports making a call to an internal procedure inside a server-side procedure, which may or may not already be running. It uses the Progress 4GL dynamic Call object, which is new to Progress Version 9.1D, to package the parameters to the procedure call. It handles all these steps in a single AppServer call:

- Identifying whether the external procedure is already running on the server, and starting it as a persistent procedure if it isn't.
- Running the internal procedure inside the server-side persistent procedure.
- Getting back the OUTPUT parameters from the internal procedure call.
- Deleting the server-side procedure if it was started just for this call.

As a result, using `dynlaunch.i` is generally the preferred way to make calls to internal procedures across the AppServer connection in Progress Dynamics Version 2, as long as the external procedure on the server does need to be started and then left running after the call is complete.

These are the basic include file arguments for `dynlaunch.i`:

- The include file takes an **&PLIP** named argument that is the name of the external procedure needed on the server. Alternatively, as in this example, it can be the logical name (the Manager Type name) of any registered Progress Dynamics Manager, including a newly created one such as this.
- The **&Iproc** named argument is the name of the internal procedure to run.

There must be three named arguments for each parameter in the internal procedure's calling sequence. For each of these, the letter *n* represents the order of the parameters in the call:

- The **&moden** named argument is INPUT, OUTPUT, or INPUT-OUTPUT.
- The **&parmn** named argument is the name of the variable or table field storing the parameter.
- The **&datatype_n** named argument holds the datatype of the parameter.

As with all include file references, quoted strings must be inside single quotes. If a string argument includes spaces or other word breaks, then the single-quoted string must then be inside double quotes.

In the case of this example, the code is running the internal procedure `fetchFieldEdits` in the server-side `FieldEditManager`, and passing as an INPUT parameter a list of the tables to retrieve edit information for, and returning as an OUTPUT parameter a temp-table containing those edits.

Writing the server-side caching procedure

The procedure that reads database data into the temp-table is separate from `cacheFieldEdits` because it needs to have the `Db-Required` toggle set to `TRUE`, so that it will be compiled only on the server-side, or when the database is otherwise available.

This `fetchFieldEdits` procedure uses a copy of the `ttFieldEditData` temp-table that must also be defined in the Definitions section of the Manager:

```
DEFINE TEMP-TABLE ttNewData NO-UNDO LIKE ttFieldEditData.
```

This is so that only the data for the current request is returned in the call. The client-side cache will gradually be built up as more requests are made. Because no data is maintained in memory on the server, `fetchFieldEdits` first needs to empty out any leftover data from an earlier request:

Procedure `fetchFieldEdits`:

```
/*-----  
Purpose:      Server-side procedure to load field edit data from the database  
              into a temp-table, to be returned to the client.  
Parameters:  INPUT  PARAMETER pcTableList AS CHARACTER NO-UNDO --  
              list of tables to cache.  
              OUTPUT PARAMETER TABLE FOR ttNewData --  
              returned temp-table.  
Notes:       This DB-REQUIRED procedure is executed on the server, from the  
              client, if the Manager is divided between client and server.  
              It only loads newly requested data into the NewData temp-table,  
              and returns that to be appended to the client cache.  
-----*/  
DEFINE INPUT  PARAMETER pcTableList AS CHARACTER NO-UNDO.  
DEFINE OUTPUT PARAMETER TABLE FOR ttNewData.  
DEFINE VARIABLE iNum AS INTEGER NO-UNDO.  
  
/* Remove any leftover data from a previous request. The ttNewData table  
   just holds the data for the current request and returns that to the  
   client. */  
EMPTY TEMP-TABLE ttNewData.
```


The procedure then goes through the list of requested tables, locates any FieldEdit records for their fields, and creates temp-table records for them:

```

/* Read field edit data from database into the temp-table to return
to the caller. */
DO iNum = 1 TO NUM-ENTRIES(pcTableList):
  FIND FIRST gsc_entity_mnemonic WHERE
    gsc_entity_mnemonic.entity_mnemonic_description =
      ENTRY(iNum, pcTableList) NO-LOCK NO-ERROR.
  IF AVAILABLE gsc_entity_mnemonic THEN DO:
    FOR EACH gsc_entity_display_field OF gsc_entity_mnemonic,
      EACH gsc_entity_field_edit OF gsc_entity_display_field NO-LOCK:

      CREATE ttNewData.
      ASSIGN
        ttNewData.cEntityName =
          gsc_entity_mnemonic.entity_mnemonic_description
        ttNewData.cFieldName =
          gsc_entity_display_field.DISPLAY_field_name
        ttNewData.cEditType = gsc_entity_field_edit.edit_type
        ttNewData.cEditValue = gsc_entity_field_edit.edit_value.
      END. /* for each entity field edit of each display field */
    END. /* if available entity */
  END. /* do to number of tables */

RETURN.
END PROCEDURE.

```

Back in the calling procedure, cacheFieldEdits, this data is copied into the client cache:

```

FOR EACH ttNewData:
  CREATE ttFieldEditData.
  BUFFER-COPY ttNewData TO ttFieldEditData.
END.

```

Note that it would be slightly more efficient to use the APPEND keyword with the OUTPUT parameter and have the new data added directly to the existing client cache. However, the underlying mechanism that supports dyn!launch.i does not allow this, so this is a penalty to be paid when using it. Keep in mind that in more complex caching examples, it is likely that there would be some overlap between data returned and data already on the client that couldn't be checked in advance. For example, if you were to use the Repository Manager API to retrieve object definitions to cache on the client, then even when the parent object was uniquely identifiable, a single request could return a whole host of data for various object instances contained in that parent object, some of which might already be present in the client cache. In such a case you would have to receive new data in a separate table or set of tables anyway, and then do the necessary work on the client side to determine whether any of the data was already present, to avoid potential clashes of unique Object IDs or other keys.

Clearing the client cache

In order to clear the cache on the client side if it needs to be refreshed, you can define a clearClientCache procedure. This checks whether it is being invoked on the server or not, and empties the temp-table if it is on the client. The DB-Required toggle must be checked off for this procedure:

```
Procedure clearClientCache:
/*-----
Purpose:      This procedure clears the cache of field edit data.
Parameters:   <none>
Notes:
-----*/

IF NOT (SESSION:REMOTE OR SESSION:CLIENT-TYPE = "WEBSPEED":U) THEN
DO:
    EMPTY TEMP-TABLE ttFieldEditData.
END.

END PROCEDURE.
```

In this way, any further requests of the cache on the client side will force a request to the server, since no cached data will be found on the client.

Note the use of the syntax:

```
IF [NOT] (SESSION:REMOTE OR SESSION:CLIENT-TYPE = "WEBSPEED") THEN...
```

This expression tells the code whether it is executing on the server-side of a connection that has a separate client. Only if this is **not** the case do you want to empty the cache temp-table, because it is not maintained on the server.

NOTE: In this example, you could safely empty the temp-table on the server as well, because nothing is maintained there anyway. However, in cases where data is cached separately on client and server, it could make a big difference which side you empty the cache on.

The NOT keyword is included in the statement to make it TRUE for the client side (including a stand-alone client with a local database connection), or omitted to make it TRUE for the server side. The SESSION:REMOTE attribute is true if this is an AppServer session. Otherwise, the CLIENT-TYPE attribute will be equal to WEBSPEED if the session is a WebSpeed Agent. In either of these cases, the code is executing remotely relative to the user interface, and presumes to have a database connection to the repository data.

Note that this expression is a little different in its effect from the earlier check of the INTERNAL-ENTRIES, or for that matter making a CONNECTED database check. In the case where the client and server are effectively combined, that is, when the client session is running the full server-side Manager with a database connection, none the INTERNAL-ENTRIES or CONNECTED checks will return TRUE, because the full Manager is running and the database is connected. But the NOT SESSION:REMOTE check will also return TRUE, because it is a not a remote session. This is as it should be, since a single session is doing the work of both client and server. Think about which of these types of check you want to use in a particular situation, depending on the logic of the code being bracketed by the expression.

Retrieving field edit data on the client

Now that you've taken care of getting field edit data cached on the client, you need a procedure to do lookups in it for client-side objects that need the information. You can do this with a procedure called getFieldEditData. This takes input parameters for the table, field, and one or more types of Edit Types the caller needs values for, and returns a delimited list of the values. On the chance that the value for some new Edit Type might itself contain a comma, the Edit Values OUTPUT parameter uses CHR(3) as a delimiter between the values for the requested Edit Types.

The procedure simply looks up the requested record(s) in the temp-table and returns their values, returning blank for edit types not defined for the fields in the temp-table. Note that the code presumes that the data is already available in the client cache, because the SDO for the table will have requested it. If this isn't reliably the case (perhaps because clearClientCache might have been called after the SDO was initialized), this procedure could run cacheFieldEdits itself if the data needed isn't available.

Remember to check the DB-Required toggle off for this procedure:

Procedure fieldEditData:

```

/*-----
Purpose:      Returns field edit data for a database field.  It can return
edit values for one or more edit types.
Parameters:   INPUT  pcTable      AS CHARACTER - Table name
              INPUT  pcField      AS CHARACTER - Field name
              INPUT  pcEditTypes  AS CHARACTER - Edit Types
                                   can be a comma-delimited list
              OUTPUT pcEditValues AS CHARACTER - Edit Values
                                   will be a CHR(3)-delimited list
                                   for multiple edit types

Notes:
-----*/
DEFINE INPUT  PARAMETER pcTable      AS CHARACTER NO-UNDO.
DEFINE INPUT  PARAMETER pcField      AS CHARACTER NO-UNDO.
DEFINE INPUT  PARAMETER pcEditTypes  AS CHARACTER NO-UNDO.
DEFINE OUTPUT PARAMETER pcEditValues AS CHARACTER NO-UNDO.

DEFINE VARIABLE iTypes AS INTEGER NO-UNDO.
DO iTypes = 1 TO NUM-ENTRIES(pcEditTypes):
    FIND ttFieldEditData WHERE
        ttFieldEditData.cEntityName = pcTable AND
        ttFieldEditData.cFieldName  = pcField AND
        ttFieldEditData.cEditType   = ENTRY(iTypes, pcEditTypes) NO-LOCK
NO-ERROR.
    pcEditValues = pcEditValues +
        (IF iTypes > 1 THEN CHR(3) ELSE '':U) +
        IF AVAILABLE ttFieldEditData THEN
            ttFieldEditData.cEditValue ELSE '':U.
END. /* DO 1 to number of edit type entries */

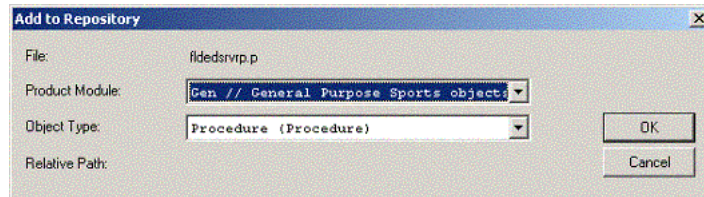
RETURN.
END PROCEDURE.

```

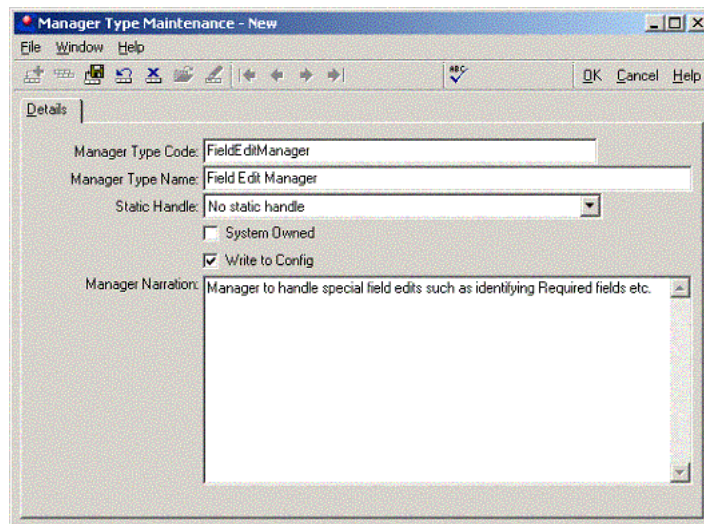
7.4.3 Registering the new manager

Now that you have completed the Manager procedure, you need to make it available to the Session Types that need it. Remember that you have both a client version and a server version. The server version, `fldedsvr.p`, must be specified for any session that expects to have the repository database connected. This would include both server-side AppServer Session Types and stand-alone client-server Session Types. The client version of the procedure, `fldedclnt.p`, should be added to a client Session Type that will not have the database connected. Follow these steps to register the new manager:

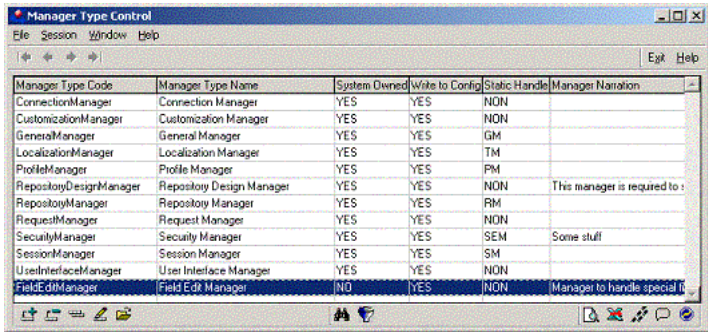
- 1 ♦ Register each version of the procedure in the repository by opening it in the AppBuilder and selecting **File→Save To Repository**.
- 2 ♦ Select the **Product Module** the procedure should be associated with, and the **Object Type Procedure**:



- 3 ♦ Select the **Manager Type Control** from the Administration Session menu, and press Add to create a record for the Field Edit Manager:



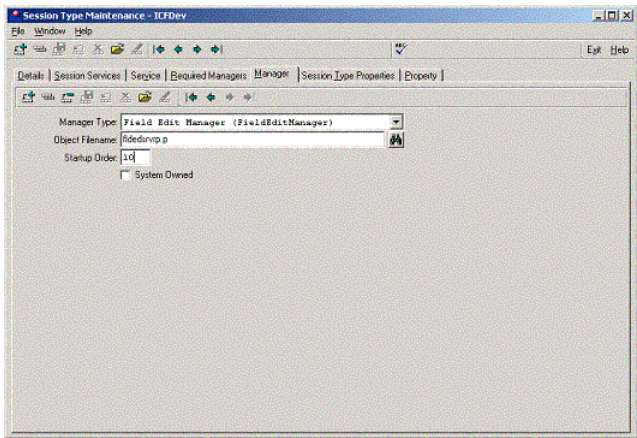
- 4 ♦ Give it a Manager Type Code of **FieldEditManager**. This is the logical name you can then use to invoke procedures inside the Manager, as you saw earlier in the `dynLaunch.i` call. Enter a more descriptive Manager Type Name, which will appear in messages and in the drop-down list of Managers displayed elsewhere, and select **No static handle** from the list of possible Static Handle values.
- 5 ♦ Check the **System Owned** toggle on if you want access to this definition to be restricted. Check the **Write to Config** toggle on to have this information written out to the configuration file. You must do this if you want to put the new Manager in the Required Managers list for any Session Type, and have it pre-started for you as part of the session.
- 6 ♦ Save this Manager Type information, and return to the manager Type Control:



The screenshot shows the 'Manager Type Control' window with a table listing various manager types. The 'FieldEditManager' is highlighted at the bottom.

Manager Type Code	Manager Type Name	System Owned	Write to Config	Static Handle	Manager Name
ConnectionManager	Connection Manager	YES	YES	NON	
CustomizationManager	Customization Manager	YES	YES	NON	
GeneralManager	General Manager	YES	YES	GM	
LocalizationManager	Localization Manager	YES	YES	TM	
ProfileManager	Profile Manager	YES	YES	PM	
RepositoryDesignManager	Repository Design Manager	YES	YES	NON	This manager is required to :
RepositoryManager	Repository Manager	YES	YES	RM	
RequestManager	Request Manager	YES	YES	NON	
SecurityManager	Security Manager	YES	YES	SEM	Some stuff
SessionManager	Session Manager	YES	YES	SM	
UserInterfaceManager	User Interface Manager	YES	YES	NON	
FieldEditManager	Field Edit Manager	NO	YES	NON	(Manager to handle special c...

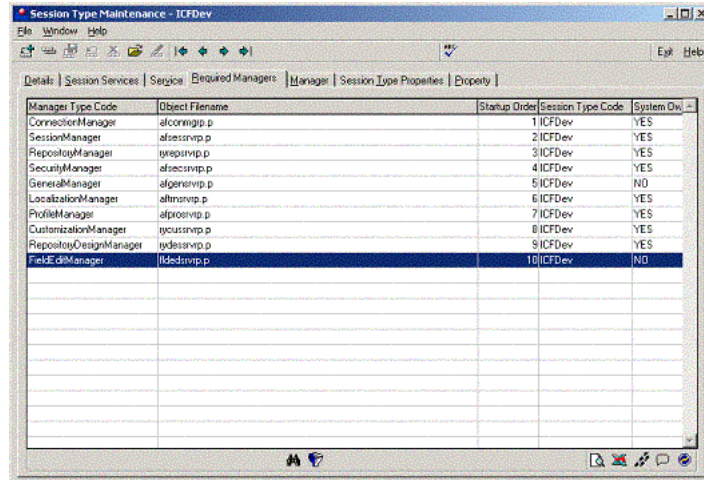
- 7 ♦ Back in the Session menu, select Session Type Control, and select the Session Type(s) you want to add either the client or server Manager to. Edit the Session Type, select the Manager tab, and press the Add button inside the folder to add the new Manager. Note that the new Manager appears in the drop-down list of Manager Types to choose from:



The screenshot shows the 'Session Type Maintenance - ICFDev' window. The 'Manager' tab is selected, and the 'Field Edit Manager (FieldEditManager)' is chosen from the 'Manager Type' dropdown. The 'Object Filename' is 'fidedurp.p' and the 'Startup Order' is '10'. The 'System Owned' checkbox is unchecked.

- 8 ♦ Specify either the client procedure or the server procedure as the Object Filename, whichever is appropriate, and pick the next available Startup Order for the Manager, so that it is the last one started.

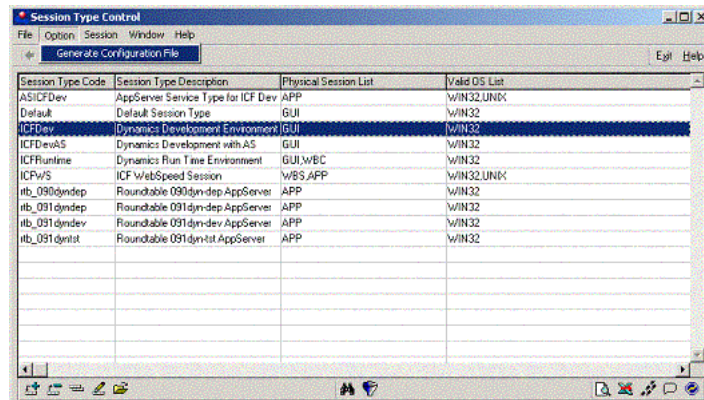
This example shows the list of Required Managers after adding the Field Edit Manager to the ICFDev Session Type:



The screenshot shows the 'Session Type Maintenance - ICFDev' window with the 'Required Managers' tab selected. The table lists various managers and their properties.

Manager Type Code	Object Filename	Startup Order	Session Type Code	System Dwg
ConnectorManager	alconnrv.p	1	ICFDev	YES
SessionManager	alssrvrv.p	2	ICFDev	YES
RepositoryManager	alrsrvrv.p	3	ICFDev	YES
SecurityManager	alsecrrv.p	4	ICFDev	YES
GeneralManager	algsrvrv.p	5	ICFDev	NO
LocalizationManager	altnsrv.p	6	ICFDev	YES
ProfileManager	alprsrv.p	7	ICFDev	YES
CustomizationManager	alcrsrv.p	8	ICFDev	YES
RepositoryDesignManager	aldrsrv.p	9	ICFDev	YES
FieldEditManager	alfedrv.p	10	ICFDev	NO

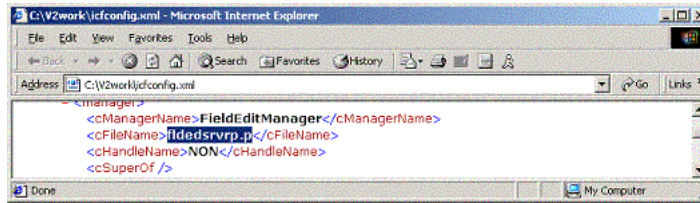
- 9 ♦ Back in the Session Type Control window, select Option→Generate Configuration File to regenerate the configuration XML file with the new information:



The screenshot shows the 'Session Type Control' window with the 'Generate Configuration File' button highlighted in the 'Option' menu. The table below lists session types and their properties.

Session Type Code	Session Type Description	Physical Session List	Valid OS List
AS/ICFDev	AppServer Service Type for ICF Dev	APP	WIN32,UNIX
Default	Default Session Type	GUI	WIN32
ICFDev	Dynamics Development Environment	GUI	WIN32
ICFDevAS	Dynamics Development with AS	GUI	WIN32
ICFRuntime	Dynamics Run Time Environment	GUI,WBC	WIN32
ICFWS	ICF WebSpeed Session	WBS,APP	WIN32,UNIX
rtb_090dyndep	Roundtable 090dyn-dep AppServer	APP	WIN32
rtb_091dyndep	Roundtable 091dyn-dep AppServer	APP	WIN32
rtb_091dyndev	Roundtable 091dyn-dev AppServer	APP	WIN32
rtb_091dynit	Roundtable 091dyn-it AppServer	APP	WIN32

Now if you check the contents of that configuration file, you see the new Manager listed for the sessions you added it to:



7.4.4 Accessing the manager from SDOs

When an SDO is initialized, it requests the field edit data for its enabled tables. To define the code to support this, you can create a custom override procedure for the data class. As with other custom super procedures used to subclass an Object, edit the *src/adm2/custom/datacustom.i* procedure to uncomment the line that starts the super procedure *adm2/custom/datacustom.p*. Then create a local *src/adm2/custom/datacustom.p* procedure and edit it to add a local version of *initializeObject*:

Procedure initializeObject:

```
/*-----
Purpose:      Override of initializeObject procedure.
Parameters:   <none>
Notes:        This gets the handle of the Field Edit Manager and passes it
               a list of enabled tables in this SDO to cache field edits for.
-----*/
DEFINE VARIABLE cEnabledTables AS CHARACTER NO-UNDO.
DEFINE VARIABLE hFieldEditManager AS HANDLE NO-UNDO.

RUN SUPER.

hFieldEditManager = DYNAMIC-FUNCTION('getManagerHandle':U IN
gshSessionManager,
                                INPUT 'FieldEditManager':U).

cEnabledTables = DYNAMIC-FUNCTION('getEnabledTables':U IN
TARGET-PROCEDURE).

RUN cacheFieldEdits IN hFieldEditManager (INPUT cEnabledTables).

END PROCEDURE.
```

The procedure first gets the handle of the new Manager from the Session Manager, using the Session Manager's global handle, **gshSessionManager**, and the logical name **FieldEditManager**. Then it gets the list of enabled tables from the SDO itself, and runs the **cacheFieldEdits** procedure in the Manager.

Next the SDO needs a procedure to do validations against the rules defined in the field edits. Procedures. If any errors in the detail are detected, this **fieldEditValidate** procedure creates a message string in the format expected by the standard validation support:

```

Procedure fieldEditValidate:
/*-----
Purpose:      Perform validation stored in the entity field edit table.
Parameters:   OUTPUT pcMessageList AS CHARACTER
Notes:        Invoked by SDO data logic procedure RowObjectValidate, if any
               errors occurs, they should be passed back to RowObjectValidate.
-----*/
DEFINE OUTPUT PARAMETER pcMessageList  AS CHARACTER  NO-UNDO.

DEFINE VARIABLE cEditValues           AS CHARACTER  NO-UNDO.
DEFINE VARIABLE cEnabledTables        AS CHARACTER  NO-UNDO.
DEFINE VARIABLE cTable                AS CHARACTER  NO-UNDO.
DEFINE VARIABLE cValueList            AS CHARACTER  NO-UNDO.
DEFINE VARIABLE hColumn               AS HANDLE     NO-UNDO.
DEFINE VARIABLE hFieldEditManager     AS HANDLE     NO-UNDO.
DEFINE VARIABLE hLogicBuffer          AS HANDLE     NO-UNDO.
DEFINE VARIABLE iCol                  AS INTEGER    NO-UNDO.
DEFINE VARIABLE iNumTables            AS INTEGER    NO-UNDO.

      hFieldEditManager= DYNAMIC-FUNCTION('getManagerHandle':U IN
gshSessionManager,
                                   INPUT 'FieldEditManager':U).

```

After retrieving the Manager handle, the procedure retrieves the LogicBuffer handle from the SDO. The Logic Buffer is in fact defined in the logic procedure of the SDO, and is the buffer for the record as it is accessed by the validation entry points in the logic procedure, such as rowObjectValidate:

```

RUN getLogicBuffer IN TARGET-PROCEDURE (OUTPUT hLogicBuffer).

```

For each column in the buffer, the code asks the Field Edit Manager whether there is a **Required** edit type or a **Case** edit type defined for that field, by running `getFieldEditData`:

```
DO iCol = 1 TO hLogicBuffer:NUM-FIELDS:
hColumn = hLogicBuffer:BUFFER-FIELD(iCol)
cTable = DYNAMIC-FUNCTION('columnTable':U IN TARGET-PROCEDURE,
                           INPUT hColumn:NAME).

IF VALID-HANDLE(hFieldEditManager) THEN
DO:
  RUN getFieldEditData IN hFieldEditManager
  (INPUT cTable,
   INPUT hColumn:NAME,
   INPUT 'Required,Case':U,
   OUTPUT cEditValues).
```

NOTE: This is not a particularly efficient way to determine which fields are required, since it involves making a separate call to the Manager for every field in the SDO's buffer, even though most of these probably do not have a record in the temp-table cache. If you are designing a new Manager, you should consider the most efficient way to get the information you need and structure the data and the calls accordingly.

If there is a Required type, and its value is Yes (which is normally the case if it is defined at all), and the STRING-VALUE of the field is blank, then the code uses the framework's `aferrortxt.i` include file to format a message using the built-in message with the key 'AF1':

```
/* Check the Required edit first. */
IF ENTRY(1,cEditValues,CHR(3)) = 'Yes':U AND
hColumn:STRING-VALUE = '':U THEN
  ASSIGN pcMessageList = pcMessageList +
  (IF NUM-ENTRIES (pcMessageList,CHR(3)) > 0 THEN CHR(3) ELSE '':U) +
  {aferrortxt.i 'AF' '1' cTable hColumn:NAME hColumn:LABEL}.
```

If you look this message up in the Administration System menu, under the Message Control, you see the display shown in [Figure 7-6](#).

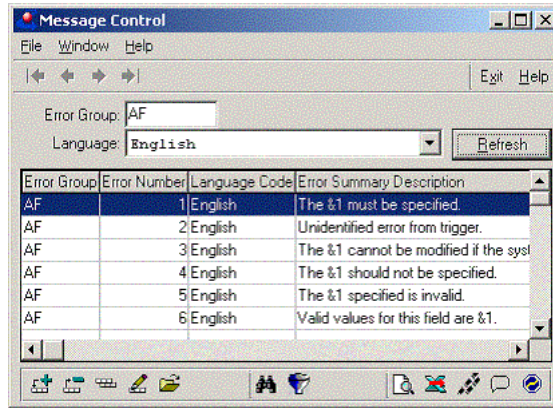


Figure 7-6: Message Control window

The hColumn:LABEL argument to aferrortxt.i is the value used for the substitution argument &1 in the Error Summary Description.

Next the code checks the other type of edit, which specifies that a character field should be upper case or lower case, and makes the appropriate change:

```
/* Then check the Case edit, and adjust the value of the field if there's
   an edit defined for it. */
CASE ENTRY(2,cEditValues,CHR(3)):
  WHEN 'Upper':U THEN
    ASSIGN hColumn:BUFFER-VALUE = CAPS(hColumn:BUFFER-VALUE).
  WHEN 'Lower':U THEN
    ASSIGN hColumn:BUFFER-VALUE = LC(hColumn:BUFFER-VALUE).
  OTHERWISE.
END CASE.
END. /* if valid field edit manager handle */
END. /* do to number of fields in logic buffer */
END PROCEDURE.
```

7.4.5 Accessing the manager from viewers

In order for Viewers to display a visual cue for required fields, you must override the standard Viewer behavior with a `customviewer.p` procedure. As for the data class, edit `src/adm2/custom/customviewer.i` to start the super procedure `adm2/custom/viewercustom.p`. Then create a local version of `src/adm2/custom/viewercustom.p` and define an `enableFields` procedure in it. Invoke the standard behavior with a `RUN SUPER` statement, and then retrieve the Field Edit Manager handle:

```

Procedure enableFields:
/*-----
  Purpose:      Override of enableFields to apply affordance for field edits
                 defined in the entity field edit table.
  Parameters:   <none>
  Notes:
  -----*/
DEFINE VARIABLE cEditValue      AS CHARACTER NO-UNDO.
DEFINE VARIABLE cFieldHandles   AS CHARACTER NO-UNDO.
DEFINE VARIABLE cTable          AS CHARACTER NO-UNDO.
DEFINE VARIABLE hDataSource     AS HANDLE NO-UNDO.
DEFINE VARIABLE hField          AS HANDLE NO-UNDO.
DEFINE VARIABLE hFieldEditManager AS HANDLE NO-UNDO.
DEFINE VARIABLE hSideLabel      AS HANDLE NO-UNDO.
DEFINE VARIABLE iNumField       AS INTEGER NO-UNDO.

  RUN SUPER.

  hFieldEditManager= DYNAMIC-FUNCTION('getManagerHandle':U IN
gshSessionManager,
                                     INPUT 'FieldEditManager':U).

```

Now you need to get the `DataSource` handle (the Viewer's SDO), which you use to identify the `columnTable` property for each of the Viewer's fields. This is the database table the column is derived from, and becomes one of the arguments to `getFieldEditData`.

You also need to retrieve the `FieldHandles` property of the Viewer. This is a list of the widget handles for the field representations in the Viewer.

For each of the fields, run `getFieldEditData` in the Manager to see if there is a Required edit for the field:

```
hDataSource = DYNAMIC-FUNCTION('getDataSource':U IN TARGET-PROCEDURE).

cFieldHandles = DYNAMIC-FUNCTION('getFieldHandles':U IN TARGET-PROCEDURE).

DO iNumField = 1 TO NUM-ENTRIES(cFieldHandles):
  hField = WIDGET-HANDLE(ENTRY(iNumField, cFieldHandles)).

  cTable = DYNAMIC-FUNCTION('columnTable':U IN hDataSource,
                           INPUT hField:NAME).

  IF VALID-HANDLE(hFieldEditManager) THEN
    DO:
      RUN getFieldEditData IN hFieldEditManager
        (INPUT cTable,
         INPUT hField:NAME,
         INPUT 'Required':U,
         OUTPUT cEditValue).
```

If there is a Required edit, then you modify the side label for the field to begin with an asterisk:

```
IF cEditValue = 'yes':U THEN
  DO:
    ASSIGN
      hSideLabel = hField:SIDE-LABEL-HANDLE
      hSideLabel:SCREEN-VALUE = '*' :U + hSideLabel:SCREEN-VALUE NO-ERROR.
  END. /* if required is "yes" */
END. /* if field edit manager valid */
END. /* do to number of data fields in viewer */

END PROCEDURE.
```

Save and compile the `viewercustom.p` procedure.

7.4.6 Invoking the field edits from an SDO

The `rowObjectValidate` procedure in an SDO's logic procedure has statements in it that are generated by the Object Generator, among other things to verify that mandatory fields (as designated in the Data Dictionary) are filled in. The code is in this form, for each mandatory field:

```
IF isFieldBlank(b_Customer.Comments) THEN
  ASSIGN
    cMessageList = cMessageList + (IF NUM-ENTRIES(cMessageList,CHR(3)) > 0
      THEN CHR(3) ELSE '':U) +
      {aferrortxt.i 'AF' '1' 'Customer' 'Comments' ''Comments''}.
```

The presumption of the example is that you want to replace this standard check with a check based on the data you have entered into the `field_edit` table. So you should edit the logic procedure of an SDO and replace these statements with a single call to `fieldEditValidate`. It will return a message string in the same format as that generated by the other validation checks:

```
Procedure rowObjectValidate:
/*-----
  Purpose:      Procedure used to validate RowObject record client-side
  Parameters:   <none>
  Notes:
  -----*/

  DEFINE VARIABLE cMessageList    AS CHARACTER    NO-UNDO.
  DEFINE VARIABLE cValueList      AS CHARACTER    NO-UNDO.

  RUN fieldEditValidate IN TARGET-PROCEDURE (OUTPUT cMessageList).

  ASSIGN ERROR-STATUS:ERROR = NO.
  RETURN cMessageList.

END PROCEDURE.
```

7.4.7 Testing the new manager

To test your new Manager, build a window using the SDO that has the new version of `rowObjectValidate`. Either compile a static Viewer for the SDO, or recompile the dynamic Viewer procedure `ry/obj/rydynvieww.w` to support your new behavior for all dynamic Viewers. Add some records to the `gsc_entity_field_edit` table to define some edits for fields in the SDO. (These screen shots reflect the sample data entered using the procedure shown earlier in this section.)

Start the Session Type that uses the Field Edit Manager, and launch your window. The Viewer should show the asterisk for each Required field, as shown in [Figure 7-7](#).

Cust Num	Country	Name	Address
1	USA	Lift Tours	276 North Driv
2	Finland	Urpon Frisbee	Rattipolku 3
3	USA	Hoops	Suite 415
4	United Kingdom	Go Fishing Ltd	Unit 2
5	USA	Match Point Tennis	66 Homer Pl
6	United Kingdom	Fanatical Athletes	20 Bicep Bridge

* Cust Num: 1 * Name: Lift Tours

Address: 276 North Driveway

* City: Burlington MA

Postal Code: 01730 * Country: USA

Balance: 903.64 Credit Limit: 66,700

Discount: 35% Sales Rep: HDM

Comments:

This customer is on credit hold.

Figure 7-7: Field Edit Manager session type

If you blank out a required field and attempt to save that change, an error message such as the one in [Figure 7-8](#) appears.

ADM2Message

Message Summary Message Detail System Information Appserver Information

The City must be specified. (AF:1)

Update cancelled.

OK

Figure 7-8: Error message

If you enter a state code in lower case, it automatically changes to upper case on Save.

7.5 Customizing an existing manager

To customize an existing manager, create a new procedure and add the existing manager as a super procedure. The new procedure will consist of overrides and RUN SUPER statements.

Understanding the Object Tables In the Progress Dynamics Repository

This chapter describes those elements of the Progress Dynamics repository that make up the definition of application Objects, which includes:

- Procedural Objects such as SDO logic procedures, custom super procedures, and business logic procedures (PLIPs)
- Object Types and the class hierarchy that defines a SmartObject
- SmartObjects, both static and dynamic (though detailed information is stored in the repository only for dynamic objects)
- Object instances as used on a particular container
- Object attributes (properties)
- SmartObject links that define the communication path between Objects
- Page Layouts and Folder Pages

Like other chapters in the *Progress Dynamics Programming Handbook*, this one observes a convention of using the word *Object* with a capital O to denote an application component that has attributes and specialized behavior associated with it. Not all of these things are in fact SmartObject. Database fields are represented in the repository as *DataField Objects*, and although these are not SmartObjects, they have attributes defined in the repository that extend their behavior beyond simple *widget objects* such as buttons and ordinary fill-in fields. Procedures registered in the repository are also Objects with extended behavior beyond ordinary operating system procedures. Their relative pathnames and other information are stored in the repository, allowing the framework to access and manage them more effectively.

While this chapter does not describe any one specific product feature, the information provided should serve as a basis for understanding all the framework features that either populate the Object tables in the repository (such as the Entity Import tool, the Object Generator, etc.) or read the information at runtime to realize the application. It can also be used as a guide to anyone developing new tools and procedures that need to read or write to the repository.

The chapter describes the actual repository tables and their fields in detail, in order to provide you with as complete an understanding of the repository database as possible. However, you should always keep in mind as you read this material that Progress Dynamics includes an extensive API to provide both design-time and runtime access to the repository, and any code you write that needs to use the repository should always use this API, which will continue to be supported and kept compatible and consistent, even as changes are made over time to the underlying specifics of the data structure.

This material is of interest to Progress Dynamics application developers, and is designed to help you to understand better how the framework operates. Remember that you can write complex and complete Progress Dynamics applications, including customizations and extensions to the framework and its Object Types, without ever writing a line of code that uses the repository API, or accesses the repository database directly. Knowledge of the repository internals is of interest to those writing new tools to manage the repository data in some way that goes beyond the support that is currently provided by the standard framework tools.

An example of this could be a custom migration tool that you build to convert your existing application, which of course has its own particular structure and architecture, into a Progress Dynamics-based application that is largely data-driven. Another example could be an application analysis tool that provides useful information to you about the structure of your application, such as a tool to analyze dependencies and provide you with information on the impact of changes, and so forth. Yet another example would be a new set of runtime driver procedures to read the data and create an interface for a new type of client platform.

Over time the Progress Dynamics development team will endeavor to provide all Progress developers with more useful tools of this type, but we will never be able to satisfy every user need, and we want you to be as independent as possible in your ability to provide yourself (and potentially others in the Progress community) with the tools you need to complete and manage your application.

The Progress Dynamics repository database contains tables that support many functions, from the definition of application components to user maintenance, security definition, session and configuration management, message maintenance, translation, deployment information, and more. This chapter focuses on those tables that support the definition of application Objects, in particular those that are represented only as data in the repository and which are then realized dynamically at runtime. The tables that are used in other parts of the framework such as session management, and the APIs that provide you with access to that information are described in other documents, including the *Progress Dynamics Managers API Reference* and the *Progress Dynamics Administration Guide*.

The discussion in this chapter is broken down into logical groups of related tables, though nearly all the tables involved are related to one another in some way. Much of the basis for this documentation is taken from the extensive internal documentation in the ERwin model for the repository database, and the reader with access to ERwin can refer to the model directly for the complete picture of the repository database and all its tables.

The following sections break down the Object repository into meaningful groups of related tables, and discuss the nature of the data, how it can be generated, and how it is used at runtime to realize a largely dynamic application:

- [Architectural principles](#)
- [Object types, SmartObjects, and instances](#)
- [Attribute tables](#)
- [SmartLink tables](#)
- [Folder page tables](#)
- [Using the Repository Manager](#)

8.1 Architectural principles

Progress Dynamics is based on the Application Development Model for Progress Version 2 (ADM2) and its SmartObjects. The definitions, and in particular some of the names, used in the repository tables discussed here reflect that. Note however that while the term SmartObject, along with other ADM terminology such as SmartLink, is prevalent throughout this part of the repository and certainly in this document, there is relatively little in the definition of the repository schema or even the data for a particular application that is specific to SmartObjects. A few characteristics, such as the definition of Supported Links and other details, are mapped to the definition of such things within the ADM2. However, there is nothing to prevent a developer from using the same schema and most of the same data to define application components that were not realized through the ADM.

In fact, as more and more Objects become logical entities that are just a collection of records in the database, realized by some driver procedure at runtime, the number of actual independent SmartObjects as such becomes ever smaller. The goal over time is that **all** application Objects will be dynamically generated, and that only application-specific business logic will be defined in 4GL source code procedures. Viewed in that way, the whole notion of a SmartObject application becomes unclear. The ADM then becomes nothing more than a convention for defining Object properties and a particular set of procedures for creating them at runtime and coordinating their behavior. Nothing would stop a developer from creating an alternative set of driver procedures to create a different interface and a different implementation of the Objects' behavior.

Indeed, a large part of why the repository is valuable is that it reduces Object definitions to an abstraction, so that an application can be realized on any platform, with any User Interface, driven by the same data. The dynamic HTML interface for Web browsers in Progress Dynamics Version 2 is a perfect example of this. The client code running in the Web browser does not consist of SmartObjects in the same sense that it does when you run the application on a Progress runtime client, but it reproduces almost all the same behavior, by reading exactly the same data from the repository. In large part that is why this document exists: to help people understand how to write such driver programs and how to use the data provided for them.

8.1.1 Object IDs in the Repository

There are numerous references to *Object IDs* in the table and field descriptions. The repository schema observes the standard Progress Dynamics convention whereby every table in every database has a unique key called an Object ID, a decimal value generated by a single trigger procedure to be an absolutely unique value throughout the database, and in principle, throughout the world. These fields always have a name consisting of the table name, minus the three-letter table prefix and underscore, plus the suffix `_obj`.

Although many of the tables in the repository schema have other unique keys, and in some cases other values that can be used to join them to other tables, relationships between tables are almost always defined in terms of Object IDs, where the Object ID for a record in one table is a Foreign Key, or part of a Foreign Key, in another table.

These Object ID values are never meant to be displayed, and would not be meaningful to any user. Because they are completely arbitrary values, they never are subject to change and therefore do not require rules about cascading of changes to other related tables. This principle is fundamental to the structure of the repository, as it to every database schema defined in observance of the Progress Dynamics design principles.

You can find other basic information about the repository and its naming conventions in the chapter on *Database Design Principles* in the [Progress Dynamics Developer's Guide](#). Throughout this chapter, we generally describe all the fields in each table except for the key fields. To aid readability, this chapter sometimes refers to repository tables (after the first definition of them) without the table prefix.

8.2 Object types, SmartObjects, and instances

Every Object in the application, whether static or dynamic, is registered in the repository, and every dynamic Object is fully defined by the data in the repository. The record that identifies the Object is stored in the `ryc_smartobject` table. The name SmartObject notwithstanding, Objects of any kind, whether they are true SmartObjects or not, are identified here.

The class hierarchy that provides every Object with its full definition is represented in the `gsc_object_type` table.

For every separate place an Object is used, in a container or in some other specific context, the repository holds an `ryc_object_instance` record to represent that particular use of the Object.

This section describes these tables and their relationships in the following sections:

- [Object diagram](#)
- [The object type table](#)
- [The SmartObject table](#)
- [The object instance table](#)
- [SmartObject table example](#)

8.2.1 Object diagram

The ERwin diagram in [Figure 8–1](#) shows the **object_type**, **smartobject**, and **object_instance** tables, their fields, and their relationships.

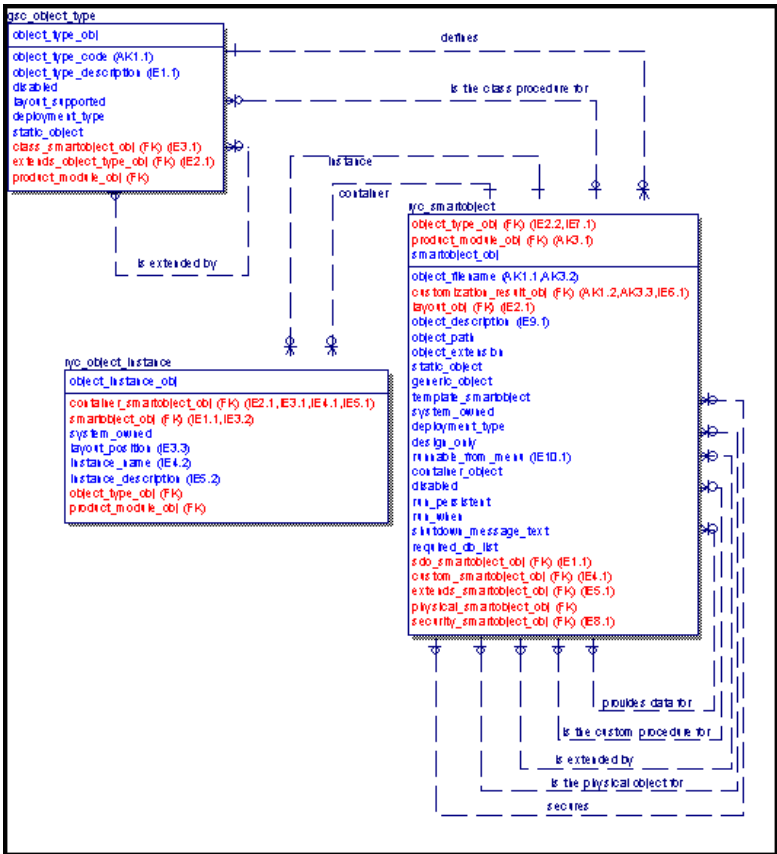


Figure 8–1: Object diagram

8.2.2 The object type table

The Object Type table **gsc_object_type** really defines an object class. Object Types are in principle hierarchical, so that an Object Type can inherit from other Object Types to define subclasses of other Objects. Although this capability is not yet fully implemented in Progress Dynamics Version 2, the data structure is there to support it, and future releases will allow both the Progress Dynamics development team and application developers to define Object Types by sub-classing other Object Types.

In the ADM2 code itself, of course, this is already done for SmartObjects, so that a SmartObject type such as a SmartDataViewer is defined in terms of a class hierarchy starting with smart.p, and continuing down through visual.p, datavis.p, and viewer.p, and optionally container.p, if the Viewer contains SmartDataFields. However, this hierarchy is not yet represented in the repository itself, and is defined by the nested include files that are part of the SmartObject's code-based definition. What will happen in the future is that the definition of this hierarchy will be fully represented in the repository itself, so that there is no need for the code that is currently compiled into the SmartObjects, including the procedures that act as drivers for Objects defined in the repository. When this happens, it will also be possible to define Objects other than SmartObjects in the same hierarchical fashion, such as a class of widget, then a fill-in that extends the class of widget with specific attributes for fill-ins, then a class of date fill-ins, etc.

Object Types are also used to define default values for many Object attributes or properties, which are inherited by every object of that type that is created.

The **gsc_object_type** table has the fields described in [Table 8-1](#).

Table 8-1: The gsc_object_type table

(1 of 2)

Field	Description
object_type_code	This CHARACTER field is a unique character string to name this type of object
object_type_description	This CHARACTER field can hold any useful description of the object type.
disabled	This LOGICAL field determines whether security checking is enabled for the Object Type. If the field is set to YES, then this Object Type is not checked by the security mechanism, and full access is granted.
layout_supported	If this LOGICAL field is set to YES, then this is a non-container SmartObject that may require a layout if built dynamically.
static_object	Every ryc_smartobject record has a static_object field that identifies whether the Object is static, that is, having its own procedure file; or dynamic, that is, generated strictly from data in the repository. This LOGICAL field defines a default value for that static_object field for all Objects of the Type

Table 8–1: The gsc_object_type table (2 of 2)

Field	Description
deployment_type	This CHARACTER field determines where Objects of this Type should be deployed. As with the static_object field, the field acts as a default for the deployment_type field stored in every ryc_smartobject record for the Type. Valid values for this field are SRV for remote server only, CLI for client only, WEB for Web browser Objects, or combinations as required, represented as a comma-delimited list, e.g. CLI , SRV. This field assists developers in deploying applications on AppServers by helping define which Objects should reside where.
class_smartobject_obj	An Object Type may be supported by a procedure that defines behavior for the Type. This is the case for SmartObject classes, and may be true for other Object Types as well. If there is a class procedure for an Object Type, it is registered in the repository in the ryc_smartobject table, like any other procedure. This Object ID field points to that class procedure if it exists.
extends_object_type_obj	This Object ID field defines the hierarchical relationship between Object Types. If it is defined, it points to the “parent” Object Type that the current Object Type extends or inherits from.
product_module_obj	Every Object Type is registered in a product module, and this Object ID points to that record.

8.2.3 The SmartObject table

The **ryc_smartobject** table has a record for every application element that can be considered an Object. This includes SmartObjects, DataFields (the repository’s record of database fields from the application), and procedure files that are in any way maintained or controlled through the framework.

NOTE: Progress Dynamics Version 1 users may be aware of a gsc_object table. The gsc_object table has been merged into the ryc_smartobject table.

In the case of dynamic SmartObjects, every piece of information about the Object definition is stored in the repository, and its ryc_smartobject record acts as a header for all those other related records. In the case of procedures, they can be registered in the repository so that they can be related to other Objects.

Procedural Objects have a SmartObject record and possibly other related records for security or other purposes, but their definition is largely in the 4GL source code in the procedure itself. Procedures that must be registered in the repository include:

- Business logic procedure (PLIPs) that are run in the application.
- Procedures that are run as the action of some menu or toolbar item.
- SmartObject class super procedures such as `smart.p`, logic procedures for SDOs.
- Custom super procedures for dynamic Objects.

Because the table holds information for many kinds of Objects, not all of its fields are meaningful or used for all Objects.

Objects must be assigned an Object Type and belong to a product module. This facilitates setting up security based on object types and modules, rather than having to secure every object individually.

The SmartObject table supports both physical (static) and logical (dynamic) Objects. If the SmartObject itself is a physical, procedural Object, then the link to the physical Object in the SmartObject record (the field `physical_smartobject_obj`) is set to 0. This is because a static SmartObject does not require another procedure to realize it at runtime. If it is a dynamic Object, then the link to the physical Object points to the procedure to use as the starting point when instantiating the dynamic Object at runtime. In other words, a logical Object such as a SmartDataBrowser is represented by a record in this table whose `physical_object_obj` value points to the procedure responsible for instantiating all dynamic SmartDataBrowsers.

If the object is flagged as a *generic_object*, i.e. a physical object that is the driver for a class of dynamic objects, then no security allocations, menus, etc. can be allocated to it as it is useless without the dynamic portions being built first against the logical objects that use it.

The name of the Object is stored in the `object_filename` field. Note that this is a bit of a misnomer, since most Objects in this table are not files on disk at all. For logical Objects, the Object name is specified without a file extension, and the path is not relevant. For procedural Objects, the name may or may not include the filename extension, depending on how the procedure is registered. Logic procedures created by the Object Generator, for example, do not have the `.p` extension as part of the Object name. Procedures that are registered through the AppBuilder, using its **File→Register In Repository** option, do have the extension as part of the `object_filename`. In either case the extension for a procedure is also stored in a separate field. The `object_path` field provides the relative pathname information to locate the object.

All Objects have an *object_type_obj*. This points to the definition of the Object Type in the *gsc_object_type* table. Security allocations can be defined against an Object Type, so that they apply to all objects of that type without each individual object needing a security allocation. Object Types are also used to define default values for many Object attributes or properties, which are inherited by every Object of that type that is created.

The **ryc_smartobject** table contains Object IDs pointing to its layout, its Object Type, its *product_module*, another Object that it may be run by, and if this object displays data for an SDO, its related *sdo_smartobject*. This table contains the following fields:

- **object_filename** — This CHARACTER field is either the physical filename (with or without extension) of a static Object, or the logical name of a dynamic Object. It is important to note that the *object_filename* by itself is not a unique key for the SmartObject table. If an Object has customizations, which basically means special attribute values for a particular user, UI Type, or other situation, then there will be multiple SmartObject records with the same *object_filename*. The primary record, which links to all of the default attributes for the Object, will have a *customization_result_obj* of 0. Other records will have the same *object_filename* along with a *customization_result_obj* pointing to a record in the *customization_result* table. Therefore you must include the qualifier **AND customization_result_obj = 0** in any query against the SmartObject table if you want to retrieve only the primary record for the Object. See the separate description of customization support for more information.
- **object_description** — This CHARACTER description is used as the default for a menu label.
- **static_object** — This LOGICAL field indicates whether the Object is static or dynamic. Static Objects are registered in the repository so that they can be used as instances within smart containers. Dynamic Objects exist in the repository so that they can be constructed at runtime.
- **system_owned** — If this LOGICAL field is set to YES, then the SmartObject record may only be modified by users with a system-owned flag. This field occurs in many tables throughout the repository to safeguard basic records without which the framework or an application can not function properly.
- **shutdown_message_text** — This message text is displayed if the user attempts to close down this Object while it contains unsaved changes.
- **template_smartobject** — If this LOGICAL field is set to YES, then this SmartObject may be used as a template for creating other similar SmartObjects. This makes it easy to define standard containers, such as a window with a standard toolbar, a standard browser, or a standard layout, and to create new Objects based on this template.

- **object_path** — This CHARACTER field is the relative file system path to the Object. This must contain forward slashes for portability. A relative path should always be used rather than an absolute path. The path must be relative to the workspace root directory. Because this path is stored in the repository, it is never necessary to identify a procedure within the repository by including its relative path; the Object name will suffice. For example, when you associate a custom super procedure with a dynamic Object, you simply provide its name, which the tools verify is the name of a registered procedure. At run time, the framework constructs the full pathname by prepending the object_path to the object_filename, in order to be able to run the procedure.
- **object_extension** — This CHARACTER field holds the filename extension for a procedure file, such as .p or .w. This is true whether the object_filename has the extension as part of the name or not. The framework uses this to construct the full filename at runtime.
- **container_object** — If this LOGICAL field is set to YES, this Object is a container window. Only container windows can appear on menus, and only container windows can have a dynamic menu structure
- **generic_object** — If this LOGICAL field is set to YES, this is a physical generic Object used as the starting point for building a dynamic Object and is not a complete Object in itself. For example, the procedure rydyncontw.w is the procedure that is used to create all dynamic windows at runtime. This is registered in the repository as a generic Object, and every other SmartObject that is itself a dynamic window points to this record as its physical_smartobject_obj.
- **required_db_list** — This CHARACTER field is a comma-delimited list of logical database names that must be connected in order to run this Object. If any databases are specified and the databases are not connected, then the program is prevented from running, and the menu option/buttons it appears on are disabled, etc.
- **runnable_from_menu** — If this LOGICAL field is set to YES, this Object can be run from a menu. Objects such as Browsers and Viewers cannot be run on their own from a menu. Object controllers and menu controllers can always be run from the menu. Smart windows can sometimes be run from a menu, depending on whether they maintain a specific record or not, that is, whether they are dependent windows that require a key to be passed to them as input. If this flag is set to NO, this Object cannot be placed on a menu or dynamic toolbar.

- **disabled** — If this LOGICAL field is set to YES, access to this Object is disabled, regardless of any other security settings.
- **run_persistent** — If this LOGICAL field is set to YES, this Object is run persistently. This is used only for static Objects.
- **run_when** — This CHARACTER option defines the following circumstances under which this program may be run:
 - ONE — Only 1 instance of this program can be run at a time.
 - NOT — This program can only be run when there is no transaction open.
 - ANY — This program can be run anytime.
 - NOR — This program can only be run when no other programs are running. While it is running, no other programs can be started.

This field is not yet actively used.

- **deployment_type** — This CHARACTER field holds the deployment type for the specific Object. Its value is normally inherited from the gsc_object_type table and the field is described there.

NOTE: A deployment type of NON is used for static objects.

- **design_only** — This LOGICAL field identifies the Object as being required only during design or development time. The flag allows the framework to identify which Objects to deploy when it is building a runtime-only deployment package that does not require any Objects that are used to build the application. The default field value is NO.
- **custom_smartobject_obj** — This DECIMAL field points to a custom super procedure used to provide behavior for a dynamic Object. The procedure itself must also be registered in the SmartObject table. This allows its Object ID to be stored in this field for each SmartObject it is associated with.
- **extends_smartobject_obj** — This DECIMAL field enables class inheritance for Objects, so that the actual Object super procedure hierarchy can be represented in the repository, and also so that non-SmartObjects such as DataFields can be defined in terms of a hierarchy of domains that define increasingly specific behavior for types of fields. Although this behavior is not yet implemented in Version 2, the repository structure is present to allow it to be done in a future release.

- **physical_smartobject_obj** — As noted in the description of the generic_object field, some SmartObject records represent the driver procedures that create dynamic Objects at runtime. If a record represents a dynamic SmartObject, this DECIMAL field holds the Object ID of the generic Object that realizes it at runtime.
- **security_smartobject_obj** — This DECIMAL field is usually the same as the smartobject_obj. It indicates that security is enabled for the Object and that this Object is used for any security checks. If this field is set to 0, security checking is disabled for this Object. A different Object may be specified as the security Object if this is required to make setting up security restrictions easier. For example, all Objects on a container can point at the container Object for their security. Possibly an entire suite of Objects can point at the same security Object, such as the menu.
- **customization_result_obj** — This DECIMAL field can contain the Object ID of a record in the customization_result table, and if it is defined, it is combined with the object_filename to identify the SmartObject record for a particular customization of a base SmartObject. This is described in more detail in a separate section on customization.

8.2.4 The object instance table

The **ryc_object_instance** table contains a record for each instance—each distinct usage or context within the application—of each SmartObject. This therefore represents a running instance of an Object on a container, and every SmartObject instance is identified and distinguished based on the container in which it is run – that is the context that makes the use of the Object distinct. The object_instance table facilitates the allocation of specific attributes, links, and page numbers for the specific instance of an Object.

The table holds Object IDs that point to its container (container_smartobject_obj), the SmartObject that it is an instance of (smartobject_obj), its Object Type (object_type_obj), and its product module (product_module_obj). It also has these other fields:

- **system_owned** — If this LOGICAL field is set to YES, this record may only be modified by users with a system-owned flag.

- **layout_position** — This CHARACTER field is a code indicating where in the layout of a container the Object belongs. The Layout Manager then uses this to automatically position Objects at runtime. The standard layout used for all container windows you build using the framework is now the “relative” layout, and for this layout type, the *layout_position* is always a three-character code. The first character is M if the Object is anywhere in the main section of its container (which effectively means anywhere except the bottom line), or B if it is on the bottom line. The second and third characters are digits representing the row and column of the object, corresponding to the grid position you assign to it in the Container Builder. The row corresponds to the row 1-9 in the Container Builder grid. The column number corresponds to the column A-J of the Container Builder grid. For older containers, which mostly means container windows that make up some of the framework tools themselves, such as the windows on the Progress Dynamics Administration menu window, other layout types are used, and in this case the *layout_position* is something different, e.g. “top”, “center”, “bottom”, etc. These other layout types are supported for backward compatibility only.
- **instance_name** — The instance name CHARACTER field identifies the Object instance uniquely within a container. It can simply be the same as the Object name, or it can be changed to distinguish between two occurrences of the same Object in the container (such as a Toolbar used in two different places in the container). In the case of SDOs as containers for DataField objects, this name field holds the actual name of the field to use in the SDO field list. It should therefore never contain a table name or other prefix. Usually the name matches the Object name of the DataField Object, but may differ, e.g. where two tables exist in a single SDO with common fieldnames. The use of the name field therefore emulates the alias functionality that exists within SDOs, where a database field can be renamed in an SDO. In the case of an Object used in a container, the field holds the name given to the instance in the Container Builder. This can be the same as the object name, or it can be different to distinguish between multiple instance of the same object in a container, or to identify its specific use within the container.
- **instance_description** — The instance description CHARACTER field defaults to be the description from the SmartObject but it can be changed to describe the use of the Object within the container, such as “top container toolbar” or “toolbar for order browser”.

8.2.5 SmartObject table example

To illustrate the relationships among these tables, we use as an example the Order Entry Maintenance Window oemaintwin. Figure 8–2 shows a rough sketch of the Object Type, SmartObject, and Object Instance records that make up Page 0 of this window.

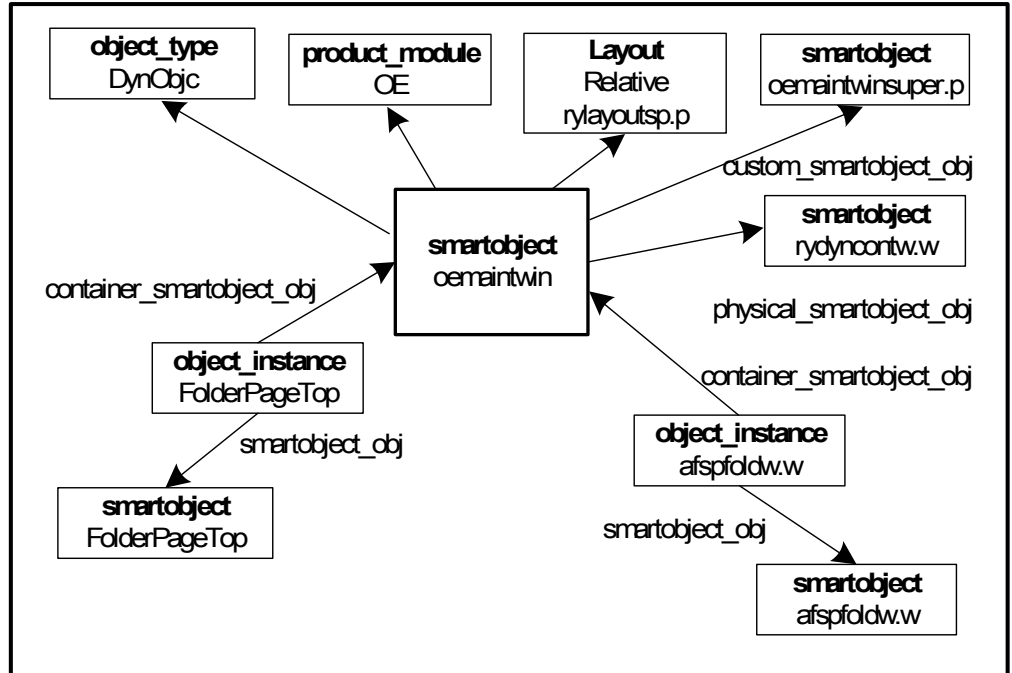


Figure 8–2: Order entry maintenance window records

It is too complicated to show all of the connections for every Object. The ones that are here show the relationships from the window itself, which you can then extend to apply to all the other Objects. To simplify the diagram a bit, we have left off the table prefixes.

To follow the diagram, start with the container window itself, the SmartObject oemaintwin. This is the master record for the window. When the framework needs to run this window, because of a request from the Dynamic Launcher, from the AppBuilder, or from an action defined for a menu or toolbar item, it locates this record to determine what to do.

There are five principal connections between the SmartObject record and other records that help to define it:

- First there is a pointer to its Object Type record, the type DynObjc, or independent window. This record in turn is connected to a number of attribute value records that define default attribute values for all windows of this type, as well as other information. For more information on attribute values and other relationships, see the [“Attribute tables”](#) section.
- Next there is the product_module record, which identifies this window as being in the OE product module.
- Next there is a pointer to the Layout record for the Relative layout, the one used for all windows built in the framework. This holds, among other information, the name of the procedure that does the layout at runtime, rylayoutsp.p.
- Finally, there are pointers to two other SmartObject records. The custom_smartobject_obj pointer connects to the SmartObject record that defines the custom super procedure for the window, the procedure oemaaintwinsuper.p. And the physical_smartobject_obj pointer connects to the SmartObject record that defines the generic object for dynamic windows, the procedure rydyncontw.w, which reads all these records and creates the window at runtime.

Those are the records from the object_type, SmartObject, and object_instance tables that together describe the window itself as an object. Just as some of its attributes come through the Object Type, the window SmartObject record also points to attribute value records for attributes defined at the Master level of the window. For more information on attributes defined at the master level, see the [“Attribute values defined at the object instance level”](#) section.

Of course the window is just a container for other objects. These are represented as object_instance records because they are instances of other SmartObjects as used in this window. In this case there are two object_instance records for the two Objects on Page 0 of the window (as well as others for the other pages that we have left off here for simplicity), the FolderPageTopToolbar instance and the folder Object instance of afsfoldw.w.

Each of these object_instances in turn points to its own SmartObject record for the Master object. Again, some of each Object’s attributes are defined in the Master, some are defined for the Instance, and some are inherited from each Object’s class (Object Type).

8.3 Attribute tables

Objects of all types can have many attributes or properties (the terms are interchangeable as used in this material) that are stored in the repository. Developers can define attributes and then associate them with one or more Object Types. For SmartObjectss, these attributes are the same as the ADM2 properties that are defined in the property include files associated with each SmartObject super procedure class, such as `smrtprop.i`, `cntnprop.i`, and so forth. In the standard Version 9 ADM code, these include files actually define the attributes for each class, by building up a dynamic temp-table with a field for each attribute. If a SmartObject such as a SmartDataBrowser inherits from numerous classes such as Smart, Visual, DataVis, and Browser, then its properties are the sum of all the properties defined for each of those classes, and its individual temp-table has a single record with a field for each of those properties.

This same mechanism is used for SmartObjects in Progress Dynamics. However, all the properties are defined in the repository, so that the property include file definitions are not needed. This provides much improved flexibility, because you can add a new attribute to an Object Type by defining it in the repository, without having to recompile every SmartObject of that type. For compatibility with ADM2 applications that do not use Progress Dynamics, the property include files are still present and are still used to compile those non-Dynamics SmartObjects. However, SmartObjects in a Progress Dynamics application do not use the contents of the include files and instead build their temp-tables up from the records in the repository.

Other kinds of Objects can have attributes as well, in exactly the same way. DataFields, for example, which are the repository's representation of application database fields, also have attributes that are built up in the same way, even though they are not SmartObjects. Procedures and other kinds of Objects recognized by the repository also have attributes.

As shipped with the product, the repository includes definitions for all the attributes used by the framework and its standard Objects. These are defined in the **ryc_attribute** table, which has a record for each distinct attribute.

The **ryc_attribute_group** table facilitates the logical grouping of attributes to simplify their use. The primary use of this table is to make the presentation of the attributes to the user more effective and usable. There is an attribute_group record for each SmartObject class that defines attributes, such as Smart, Visual, DataVis, Query, etc. There is also a large attribute_group called WidgetAttributes, with all of the field-level attributes that map to built-in 4GL widget attributes. There are other attribute groups as well. You can specify the attribute group in various framework tools to help you locate and organize attributes.

There is a record in the repository for every attribute value of every Object in the application. Some of these are defined at the Object Type level. These are default values for attributes for every Object of a given type. Each individual Object Master can also have its own attribute values, and each Instance of that Object can have its own distinct values for attributes. The **ryc_attribute_value** table holds all of those values, one value per record. Each attribute value record is linked either to the class, the master object, or the instance for which it is defined. Any attribute value that is inherited by a master from one of its classes, or by an instance of a master, is only stored at the highest level, to reduce the data in the repository and to facilitate an inheritance mechanism whereby any change to an attribute value at a higher level is automatically inherited by every Object of that type that does not specifically override the default.

Figure 8–3 is an excerpt from the repository database and shows the relationships between the attribute tables.

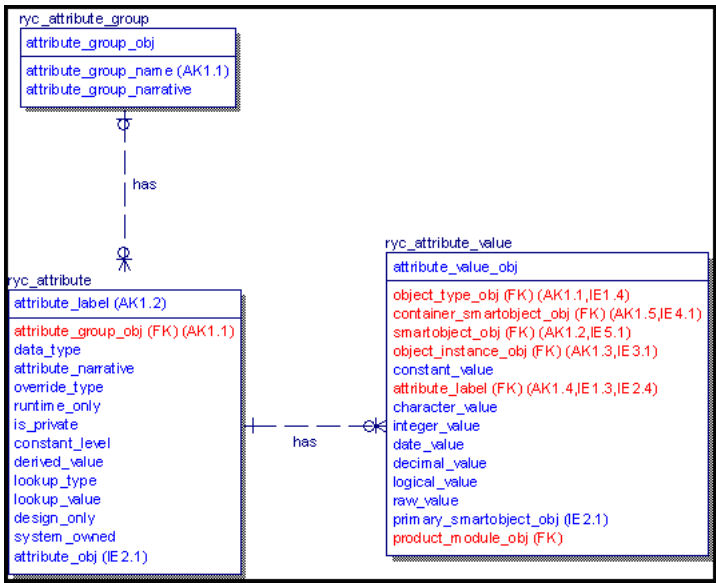


Figure 8–3: Attribute table relationships

8.3.1 The attribute group table

The **ryc_attribute_group** table has the following fields:

- **attribute_group_name** — This CHARACTER field is the name of the group, such as Smart, or WidgetAttributes.
- **attribute_group_narrative** — This CHARACTER field is a free text description of the type.

8.3.2 The attribute table

The **ryc_attribute** table has these fields:

- **attribute_label** — This CHARACTER field is the name of the attribute. For SmartObject attributes, this is the same as the name in the property include files. For widget attributes, the name is normally the same as the 4GL attribute name (including hyphens, which are otherwise normally avoided in attribute names).
- **attribute_group_obj** — This DECIMAL Object ID points to the attribute group for the attribute.
- **data_type** — This INTEGER field holds the data type of the attribute. As you will see in the description of the fields for the attribute_value table, Progress Dynamics supports a list of attribute data types corresponding to the native Progress data types, and stores the attribute value in a field of that data type. This field therefore identifies which of the attribute value fields actually holds the value for the attribute. Progress Dynamics Version 1 users may be aware that there was an ryc_attribute_type table in Progress Dynamics Version 1. This table has been removed, as the type was really nothing more than the data type, which is now stored here. This is an integer field for performance reasons. The valid values are as follows:
 - 1 — Character
 - 2 — Date
 - 3 — Logical
 - 4 — Integer
 - 5 — Decimal
 - 6 — Reserved
 - 7 — Recid

- 8 — Raw
- 9 — Rowid
- 10 — Handle
- 11 — Memptr
- 12 — reserved
- 13 — reserved
- 14 — Com-handle

These values map to the values used in the 4GL and other Progress tools.

- **attribute_narrative** — This CHARACTER field provides a full description of the purpose and use of the attribute.
- **override_type** — This CHARACTER field indicates whether the standard `get` and `set` functions must be used to set and retrieve the attribute value. Normally these functions serve only to permit other application Objects to access the properties of a SmartObject, and for access within the Object itself and its super procedures, code can directly access the temp-table that holds the Object's attribute values. In that case this field is blank. Other values this field can take are `get`, indicating the `get` function does something specific to retrieve the value and must be executed, `set`, indicating the `set` function must be executed to set the value, or `get, set` indicating both the `get` and the `set` functions must be executed. If this field has a value, then the functions to get and/or set this property need to be defined. The functions are executed instead of simply accessing the value directly in order to allow them to contain additional logic beyond simply retrieving the value from the temp-table. This is the equivalent of defining the xp preprocessors in the property include files of the ADM, but is more flexible in that it differentiates between `get` and `set`.

Note that in code that checks this value, a `CAN-DO` or `LOOKUP` should be used to ensure the order of the `get, set` pair is irrelevant.

- **runtime_only** — This is a LOGICAL field that defaults to NO. If it is set to YES, then the value of this attribute should not be stored in the repository and is only to be added to the attribute temp-table at runtime without an initial default value. An example would be an attribute that stores a handle, as the handle has no context outside of the current session, but still needs to form part of the valid attribute list. Other examples are attributes that store the current state of an Object, such as ObjectInitialized. There would be no point to assigning such attributes a default value. This field is useful in that it permits tools such as the dynamic property sheet to filter out attributes that it is not meaningful to assign a value to at design time.
- **is_private** — This is a LOGICAL field that defaults to NO. If it is set to YES, then the value of this attribute is not intended to be accessed outside of the class where it is defined. This field is also used to filter out attributes from the dynamic property sheet and other tools. Generally get and set functions are not defined for private attributes.
- **constant_level** — This is a CHARACTER field and identifies the level at which a property can be modified. The valid values for this are **class**, indicating it may only be specified at the class level, **master**, indicating that the value may be modified at the master level, and **blank**, indicating that there are no restrictions on where the attribute can be assigned a value. This field also assists tools such as the dynamic property sheet in filtering out attributes from its display. An attribute such as ObjectType is defined once for the class and should not be changed in a master or instance. A Master attribute such as the ObjectName should not be changed in an instance. Thus the dynamic property sheet and other tools disable updates to attributes with a constant_level of class when you are defining a master or instance, and disable updates to attributes with a constant_level of master when you are defining instance attributes for an Object in a container. There is also a constant_value field in the attribute_value table that is a LOGICAL indicating whether further modifications are allowed to the property. The attribute table's constant_level field therefore affects how the constant_value flag is set in the attribute_value table.
- **derived_value** — This is a LOGICAL field that defaults to NO. It can be set to YES to indicate that the property value is derived from other properties or other runtime information and doesn't need to be defined in the attribute temp-table or stored in the repository. Its value is therefore **always** set and retrieved using its get and set functions, which must contain the code needed to determine the value. As an example, the Visual class has three attributes that provide information derived from the getSysColor built-in function and the COLOR-TABLE 4GL object. These are called **color3DFace**, **color3DHighlight**, and **color3DShadow**. Because the attribute values are always derived from this system function, there is no need to store the value in the attribute temp-table. Code must always use the get functions to retrieve the values.

- **lookup_type** — This CHARACTER field defines the supported means of validating the attribute value for a specific object type. It is used in the dynamic property sheet to determine whether to overlay a combo-box, a lookup button or nothing at all on the attribute value field. Possible values are:
 - **LIST** — If the lookup_type is LIST, then the lookup_value field described next contains a string of possible attribute values in list-item-pairs format. In the property sheet, a combo-box containing these values will overlay the attribute value field, so that the user must select one of the defined values for the attribute.
 - **DIALOG** — In this case the lookup_value field contains the relative path and filename of a dialog container where the attribute value can be selected. In the property sheet, a lookup button will overlay the attribute value field. When the user presses the button, the specified dialog is launched. The user can also enter a value directly without using the dialog.
 - **DIALOG-R** — This is like the dialog option, but makes the attribute value field read-only, forcing the use of the dialog. This is necessary for dialogs that would return delimited lists that you would not want users to enter manually, or simply to control the value entered.
 - **PROC** — Here, the lookup_value field contains the relative path and filename of a procedure to execute to determine the list of valid attribute values in some special way. The Progress RETURN-VALUE of the procedure must contain a string of the list-item-pairs used to populate the combo-box that overlays the attribute value field.
 - **“”** — Blank indicates a free text entry. The dynamic property sheet simply enables the browse cell in this case so that the user can enter a value.
- **lookup_value** — This CHARACTER field can be used to specify a list of distinct values allowed for this attribute, or it can specify a procedure call that returns a list of values, or it can specify the name of a dialog, as discussed in the description of lookup_type. When it contains a list, the delimiter must be CHR(3) rather than a comma, to allow for the possibility that a value may itself contain a comma. The value and format of the lookup_value field are dependent on the value of the lookup_type field, as follows:
 - If the lookup_type is blank, then the lookup_value field is blank. Here, there is no restriction on the value the user can enter, and also no assistance in choosing a value.
 - If the lookup_type = LIST, then the lookup_value specifies the LIST-ITEM-PAIRS values. This is a delimited list of labels and values in the form:
label1,value1,label2,value2,...

- If the `lookup_type` = `DIALOG`, then the `lookup_value` specifies the relative path and file name of the dialog to be run from the property sheet. The dialog procedure must return two `OUTPUT` parameters, the first a `LOGICAL` indicating whether the property was changed, and the second a `CHARACTER` output parameter containing the value, e.g. `RUN colorChooser.w (OUTPUT 10K, OUTPUT cValue)`
- If the `lookup_type` = `PROC`, then the `lookup_value` field specifies the relative path and file name of an external procedure to be run from the property sheet at initialization, which returns a delimited list of list-item-pairs.
- **design_only** — If this `LOGICAL` field is set to `YES`, then this attribute is only modifiable at design time, not at run time. The default is `NO`.
- **system_owned** — If this `LOGICAL` field is set to `YES`, this attribute may only be modified by users with a system-owned flag in their privilege definition. Certain attributes are required for the application to function correctly and these are set to `system_owned` to prevent accidental deletion. Only users classified as able to maintain `system_owned` information may manipulate this data. In many cases, the actual attribute label needs to match to a valid Progress supported SmartObject property.

8.3.3 The attribute value table

The `ryc_attribute_value` table contains these fields:

- **constant_value** — This `LOGICAL` field is set to `YES` if the value cannot be modified at any lower level of object definition. Its value is therefore `YES` if the attribute value is for an Object Type and the `constant_level` field of the corresponding attribute is **class**. Its value is also `YES` if the attribute value is for a Master object and the `constant_level` field of the corresponding attribute is **master**. This field simply allows the `constant_level` to be verified in the `attribute_value` without having to refer back to the attribute record.
- **attribute_label** — This `CHARACTER` field is the name of the attribute as defined in the `ryc_attribute` record. Note that this is the join field used to relate this table to the `ryc_attribute` table when necessary.

The `data_type` field in the `ryc_attribute` table identifies the native data type of the attribute. Depending on that value, the actual attribute value is stored in one and only one of the following six fields; the others are unused. This allows the framework to avoid very frequent conversions back and forth to and from `CHARACTER` types, and also avoids globalization issues that can be caused when locale-sensitive values such as decimals (containing either a comma or decimal point) or dates (with different ways to order the month and day) are converted to `CHARACTER` strings.

- **character_value** — Attribute values of type CHARACTER are stored in this field.
- **integer_value** — Attribute values of type INTEGER are stored in this field.
- **date_value** — Attribute values of type DATE are stored in this field.
- **decimal_value** — Attribute values of type DECIMAL are stored in this field.
- **logical_value** — Attribute values of type LOGICAL are stored in this field.
- **raw_value** — Attribute values of type RAW are stored in this field.

The ryc_attribute_value table also contains Object IDs that point to the Object Type of the Object with this attribute value, as well as the Product Module of the Object. There are also pointers that identify whether this is a class, master, or instance attribute value. These are discussed next.

8.3.4 Identifying the level of an attribute value

Because the attribute value can be associated with any of three other tables (gsc_object_type, ryc_smartobject, or ryc_object_instance), it is necessary to identify which of the tables the value relates to. This tells us whether this is:

- A default value for an Object Type.
- A default value for a Object master.
- A value for an individual Object instance.

The `ryc_attribute_value` table therefore contains an **object_type_obj** Object ID field, a **smartobject_obj** Object ID, and an **object_instance_obj** Object ID. There is also an Object ID field for the **container_smartobject_obj** if this value is for an object_instance. Figure 8–4 is a diagram from the database model and illustrates how these fields relate the attribute_value to the object_type, smartobject, and object_instance tables.

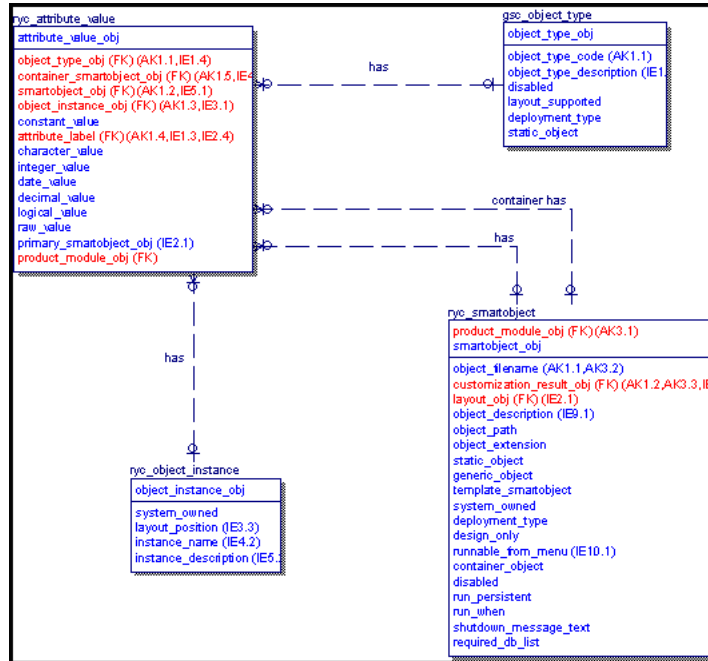


Figure 8–4: Attribute value and table relationships

The following sections describe the different ways these fields can be used:

- Defining attributes at the object type level
- Defining attribute values at the object master level
- Attribute values defined at the object instance level
- Code examples for an attribute value at the class level
- Code examples for an attribute value at the master level
- Code example for an attribute value at the instance level

Defining attributes at the object type level

When the framework creates entries for attributes of an Object Type, the `object_type_obj` points to the Object Type class, and the SmartObject and instance Object IDs are set to 0.

To illustrate, [Figure 8–5](#) shows an example diagram for the `NavigationSourceEvents` attribute, defined for the `Query` class. A `Query` object such as an SDO subscribes to various events in its Navigation-Source, and this attribute lists those events. These are defined at the class level, and normally not changed by individual objects built using the class. So we can expect that there will be a single `ryc_attribute_value` record in the repository database for this attribute for the `Query` class, but no other records for specific SDOs in the application. [Figure 8–5](#) shows the relationships for the `attribute_value` record in this case.

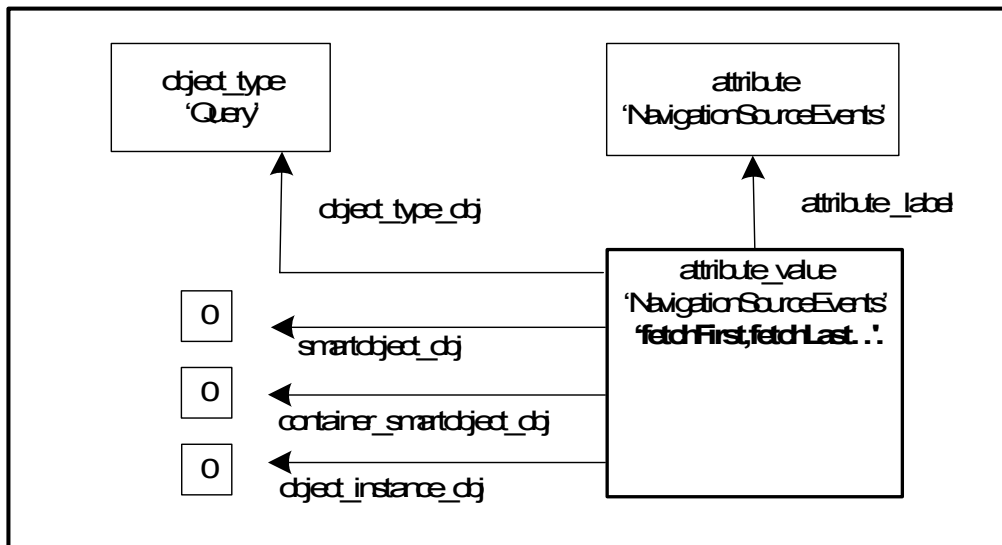


Figure 8–5: NavigationSourceEvents attribute

Here you see that only the Object Type relationship is defined for attribute values that are default values for a class.

Defining attribute values at the object master level

When the framework creates entries in the table for an *Object master*, it populates the `object_type_obj` field to avoid having 0 in the key. It also sets the `smartobject_obj` to point to the `ryc_smartobject` record that defines the Object.

To illustrate this case, [Figure 8–6](#) shows the relationships for the MinHeight attribute of the dynamic Viewer customerviewv. This attribute sets an Object’s initial height and is defined when the Object is created. Because Viewers come in all different sizes, this attribute is defined at the master level, along with the MinWidth attribute, to define the size of the Viewer. You can see this attribute in the dynamic property sheet for the Viewer when you edit it in the AppBuilder.

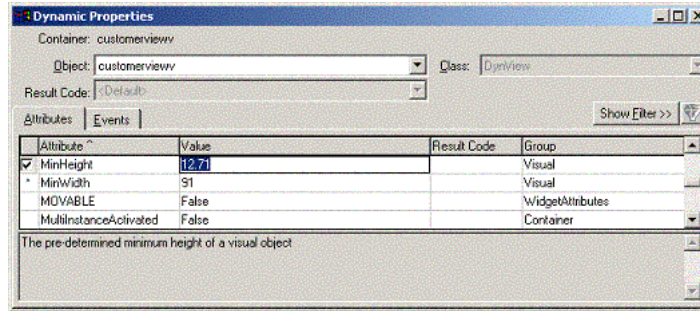


Figure 8–6: Attributes defined at the object master level

[Figure 8–7](#) shows how the attribute_value record related to other records.

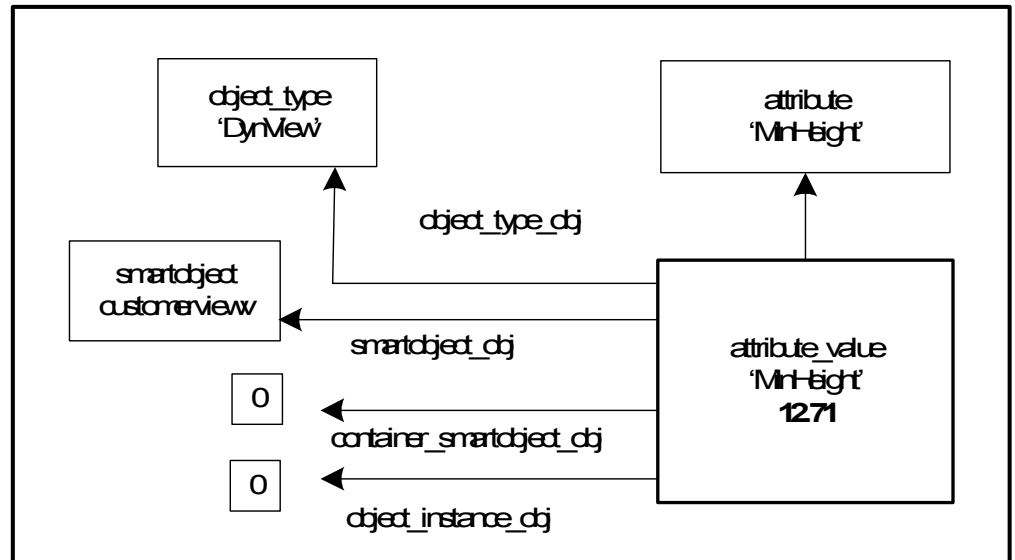


Figure 8–7: Relationships of attribute value records to other records

In this illustration you can see that the SmartObject relationship is defined for the attribute_value, because it is a value for a particular SmartObject. Because this is the master for the Object, there is no instance or container relationship.

Attribute values defined at the object instance level

When the framework creates attributes for an **Object instance**, it populates the `object_type_obj` and the `smartobject_obj` fields, and also sets the `object_instance_obj` field to point to the `ryc_object_instance` record that defines the instance, as well as the `container_smartobject_obj` field, which points to the container's `ryc_smartobject` record.

To illustrate this case, we can extend the second example to show another attribute of the same dynamic Viewer. In this case we pick an attribute that can be changed at the instance level, such as `DisableOnInit`. You can change this from its default value of `No` to `Yes` in the dynamic property sheet, as shown in [Figure 8–8](#).

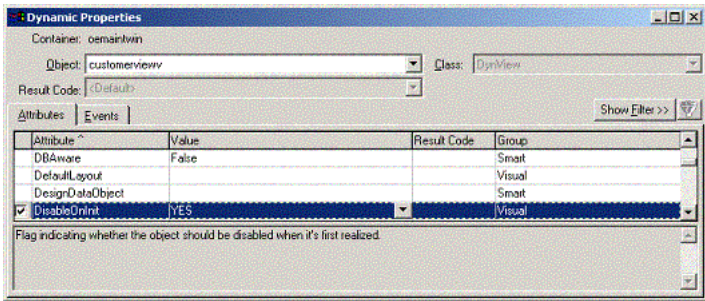


Figure 8–8: Attributes defined at the object instance level

If you do this in the Container Builder, when you are building a window such as the *oemaintwin* example, you are changing the attribute value just for that single instance of the Viewer, as used in that window. The framework creates an `attribute_value` record to represent this overridden value, and connects it to the Object Type, the Viewer SmartObject, the Object Instance record for the Viewer instance, and the SmartObject record for the container window.

Figure 8–9 shows the relationships for the instance attribute_value.

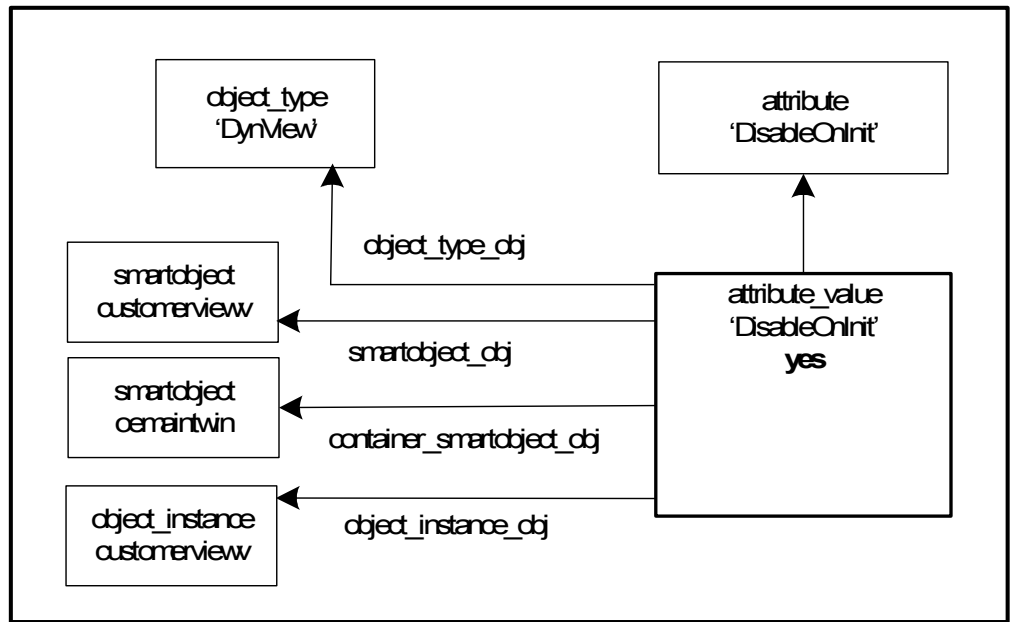


Figure 8–9: Instance attribute value relationships

The container_smartobject_obj points to the SmartObject for the window, and the object_instance_obj points to the instance of the viewer created for this window. When you place the Viewer into the window in the Container Builder, you are creating an object_instance for it, which can have its own instance name and its own attribute values, as shown in Figure 8–10.

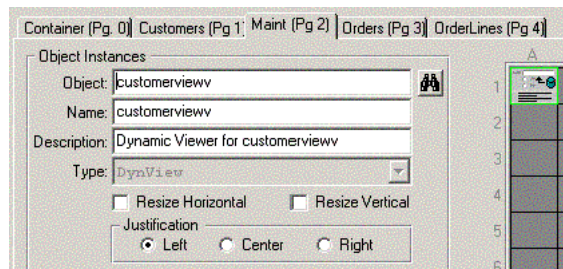


Figure 8–10: Object instances in the container builder window

All of this means that there are several different combinations of Object ID pointers that framework code needs to check for to determine the level of an attribute value and what Object it is defined for. To assist in this, another Object ID field holds the meaningful identifier for the value, so that code can look at a single field to determine what the significant key is. This field is called the **primary_smartobject_obj**. This field contains the value of the container_smartobject_obj if that is not 0; otherwise the smartobject_obj if that is not 0; otherwise 0. It is used as the replication key field when writing replication triggers to cascade changes to the version database, as the replication triggers could not handle the use of alternative fields (e.g. container_smartobject_obj or smartobject_obj). The update of this field is done in the write trigger for the table.

NOTE: Be careful when looking for attributes associated with an object_type. Make sure that you look for the specific object_type and 0 values for the SmartObject and object_instance Object ID fields.

This may all sound complicated, but the rules that determine which fields are used are fairly straightforward. We can summarize all this with some code samples in the following sections:

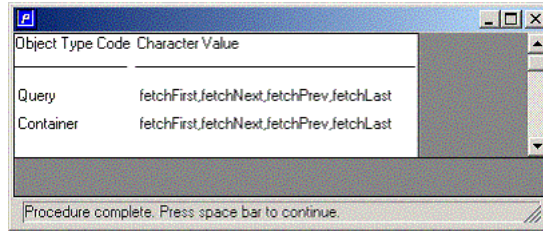
Code examples for an attribute value at the class level

If an attribute value is defined for a Object Type, or class, then the object_type_obj is defined and points to the object_type record. The other Object IDs, for SmartObject, instance, and container, are not defined (therefore 0), because the value is not associated with a specific Object. You can determine all the default attribute values for a class by retrieving the attribute_value records for which smartobject_obj, object_instance_obj, and container_smartobject_obj are all 0.

The next example shows a simple code block that locates all the Object Type records for the NavigationSourceEvents attribute. The code retrieves only records where the smartobject_obj is 0. (If this is the case, then object_instance_obj and container_smartobject_obj are also 0.) This locates the attribute values defined at the class level:

```
FOR EACH ryc_attribute_value WHERE
    attribute_label = 'NavigationSourceEvents' AND smartobject_obj = 0:
    FIND gsc_object_type OF ryc_attribute_value.
    DISPLAY object_type_code character_value FORMAT "x(40)".
END.
```

This result in [Figure 8–11](#) shows that there are two classes that define this attribute.



Object Type Code	Character Value
Query	fetchFirst,fetchNext,fetchPrev,fetchLast
Container	fetchFirst,fetchNext,fetchPrev,fetchLast

Procedure complete. Press space bar to continue.

Figure 8–11: Class level attribute value

We have already discussed the Query class. The Container class uses this attribute because a container can be a pass-through object for a Navigation link coming in from outside the container. We can see that both classes define the same initial value for the attribute. Because it's a CHARACTER attribute, the value is stored in the character_value field.

To confirm that this attribute does not change below the class level, you can leave out the WHERE clause qualifier **AND smartobject_obj = 0**:

```
FOR EACH ryc_attribute_value WHERE
    attribute_label = 'NavigationSourceEvents' /* AND smartobject_obj = 0 */ :
    FIND gsc_object_type OF ryc_attribute_value.
    DISPLAY object_type_code character_value FORMAT "x(40)".
END.
```

When you run this, the result is the same: only the two records show up. So no master SmartObjects override this default value.

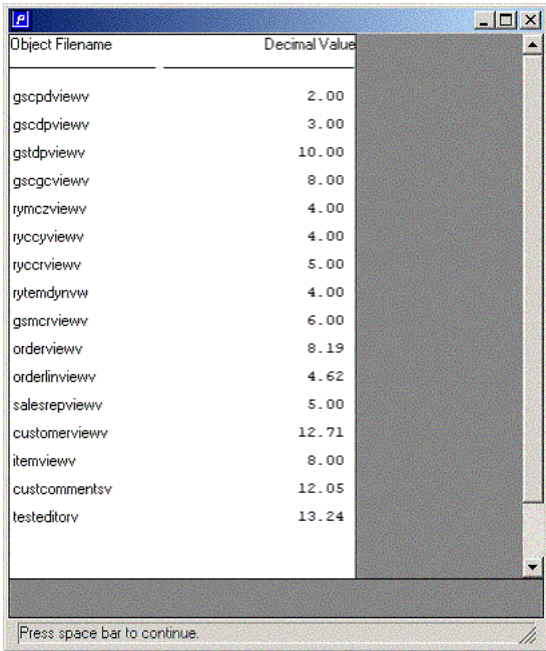
Code examples for an attribute value at the master level

If an attribute value is defined for an **Object master**, then the significant pointer is the smartobject_obj, which joins the record to the SmartObject record for the master Object. The object_type is also filled in the previous example.

Another simple code block illustrates this relationship. The code looks for any attribute_value for the MinHeight attribute that has a SmartObject Object ID. These all join to a SmartObject master. We only want to see those values that are defined for the master and not for one of its instances, so we also include the qualifier **object_instance_obj = 0**:

```
FOR EACH ryc_attribute_value WHERE attribute_label = 'MinHeight'
    AND smartobject_obj NE 0 AND object_instance_obj = 0:
    FIND ryc_smartobject OF ryc_attribute_value.
    DISPLAY object_filename FORMAT "x(20)" decimal_value .
END.
```

This is the result of the request. [Figure 8–12](#) shows the customerviewv Viewer among the Objects with an assigned MinHeight value.



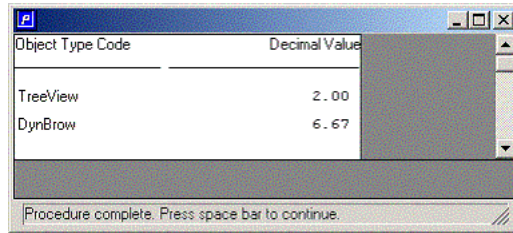
Object Filename	Decimal Value
gscpdviewv	2.00
gscdpviewv	3.00
gstdpviewv	10.00
gscgcviewv	8.00
rymczviewv	4.00
ryccyviewv	4.00
ryccrviewv	5.00
ryltemdynvw	4.00
gsmrcviewv	6.00
orderviewv	8.19
orderlinviewv	4.62
salesrepviewv	5.00
customerviewv	12.71
itemviewv	8.00
custcommentsv	12.05
testeditorv	13.24

Figure 8–12: Objects with assigned minimum height values

To find out which Object Types have a MinHeight defined for the class itself, look for Object Types where the smartobject_obj is equal to 0:

```
FOR EACH ryc_attribute_value WHERE attribute_label = 'MinHeight'
  AND smartobject_obj = 0:
  FIND gsc_object_type OF ryc_attribute_value.
  DISPLAY object_type_code FORMAT "x(20)" decimal_value.
END.
```


There are in fact a few classes that define an initial value for the MinHeight, as shown in [Figure 8-13](#).



Object Type Code	Decimal Value
TreeView	2.00
DynBrow	6.67

Figure 8-13: Classes with defined initial values for MinHeight

In this example, the default height for a dynamic Browser is 6.67 rows. Because the Browser is a resizable object, this value applies initially to **all** dynamic Browsers, unless you set it otherwise for the master or for an instance of the master in a window. By contrast, the dynamic Viewer class does not have a default for MinHeight, because the height of the Viewer is always determined by the layout of the fields it contains.

Code example for an attribute value at the instance level

If an attribute value is defined for an **Object instance**, then there are really three significant pointers. As for the master attribute, the `smartobject_obj` joins the `attribute_value` record to the master SmartObject. In addition, the `object_instance_obj` joins the record to the `object_instance` of the SmartObject it's defined for. Because the instance is always defined in the context of a particular container, the `container_smartobject_obj` is also defined, and points to the SmartObject record of the container window. Again, for completeness, the `object_type_obj` is also filled in.

Here's an example of this relationship using the `DisableOnInit` attribute. This block of code locates all `attribute_value` records where the `DisableOnInit` attribute has been set to YES at the `object_instance` level.

The code follows the join to locate the master SmartObject record for the value.

It also follows the join to the `object_instance` for the value, from which it displays the layout position.

Then it follows the join to the SmartObject record for the container. Because there are two different SmartObjectss the attribute value joins to, the master and the container, the code needs a second buffer for the container:

```
DEFINE BUFFER container_smo FOR ryc_smartobject.

FOR EACH ryc_attribute_value WHERE object_instance_obj NE 0 AND
  attribute_label = 'DisableOnInit' AND logical_value = YES:
  FIND ryc_smartobject OF ryc_attribute_value.
  FIND ryc_object_instance OF ryc_attribute_value.
  FIND container_smo WHERE container_smo.smartobject_obj =
    ryc_attribute_value.container_smartobject_obj.
  DISPLAY ryc_attribute_value.logical_value VIEW-AS FILL-IN
    ryc_smartobject.object_filename FORMAT "x(20)"
    container_smo.object_filename    FORMAT "x(20)"
    ryc_object_instance.layout_position.
END.
```

When you run this block of code, you see the three Viewers in the oemaintwin test window shown in [Figure 8–14](#). These are all disabled on initialization. (Note that the OrderLine Viewer and the Order Viewer have the same layout position because they are on different pages.)

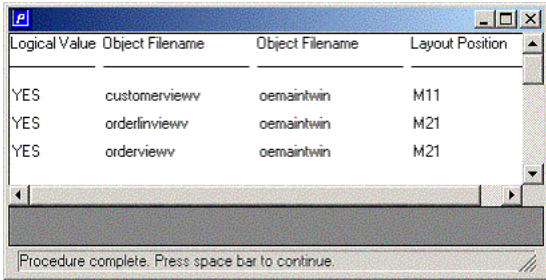


Figure 8–14: DisableOnInit properties for oemaintwin

8.4 SmartLink tables

The object repository also stores the representation of links between objects, which are paths of communication for events. As with the general notion of SmartObjects, this discussion of SmartLinks does not need to be taken as being entirely specific to support for SmartObjects. The notion of a path between two object procedure handles, which is used to identify where named events are published and subscribed to, could be applied to almost any implementation of an architecture in which Progress 4GL persistent procedures are used as application components.

Two general support tables identify what the possible link names and uses are.

8.4.1 The SmartLink type table

The **ryc_smartlink_type** table defines the SmartLinks available for linking objects. Examples of standard links include Page, Container, Update, Commit, TableIO, etc. The main purpose of this table is to provide a valid list of Links to choose from when building generic containers. Developers can define additional links using the `user_defined_link` field. The actual link name is cascaded down onto the `ryc_smartlink` table when this is not a user-defined link. The `ryc_smartlink_type` table contains these fields:

- **link_name** — This CHARACTER field is the actual link name (excluding the source or target suffix). This name will always be cascaded down to the `ryc_smartlink` table when the user-defined link flag is set to NO, therefore avoiding having to always read this table to get the actual link name.
- **user_defined_link** — If this LOGICAL field is set to YES, this is a user-defined link and the link name specified here is for information purposes only, i.e. this is a custom link. If this is set to NO, then this is a system link and the link name specified is cascaded down onto the `ryc_smartlink` table for performance reasons.
- **system_owned** — If this LOGICAL field is set to YES, this record may only be modified by users with a system owned flag in their user profile definition.

8.4.2 The supported link table

The **ryc_supported_link** table defines the supported SmartLinks for the various types of SmartObjects, and identifies whether the link can be a Source, Target, or both. User-defined links should not be set up in this table. This table is purely to ensure that when developers link objects on containers, only valid system links are used, plus user-defined links. It is merely a developer aid. Not all types of SmartObjects support links, in which case there will be no entries in this table for them. Because this table acts as a kind of cross-reference table between `ryc_smartlink_type` and `ryc_smartlink`, it holds, in addition to its own Object ID field, the Object IDs of the object type for which a link is supported, and the Object ID of the `smartlink_type` record where the link is defined. The table has these other fields:

- **link_source** — If this LOGICAL field is set to YES, Objects of this SmartObject type are capable of acting as the Source for the specified link.
- **link_target** — If this LOGICAL field is set to YES, Objects of this SmartObject type are capable of acting as the Target for the specified link.
- **deactivated_link_on_hide** — If this LOGICAL field is set to YES, this link for this type of SmartObject is automatically deactivated when the object is hidden, and activated again when the object is viewed.

8.4.3 The SmartLink table

The **ryc_smartlink** table defines the actual SmartLinks between objects on a container and enables object communication. The link name is either user-defined, or automatically copied from the `ryc_smartlink_type` for system-supported links. If the Source Object instance is not specified (i.e., that Object ID field equals zero), then the Source is assumed to be the container. Likewise if the Target Object instance is not specified, then the Target is assumed to be the container. Example links are a TableIO link between a SmartDataBrowser and a SmartToolbar, a Record link between a SmartDataBrowser and a SmartDataViewer, etc. The table contains fields for the Object IDs of the `ryc_smartlink_type` record for which this is an instance, and the container SmartObject in which this link instance occurs. The table has these other fields:

- **link_name** — This is the actual link name. The link name may be user-defined, or automatically copied from the `ryc_smartlink_type` for system-supported links.
- **source_object_instance_obj** — This is the Object ID of the Object instance which is the Source end of the link.
- **target_object_instance_obj** — This is the Object ID of the Object instance which is the Target end of the link.

Figure 8–15 illustrates the relationships between the Link tables and the SmartObject and object_instance tables.

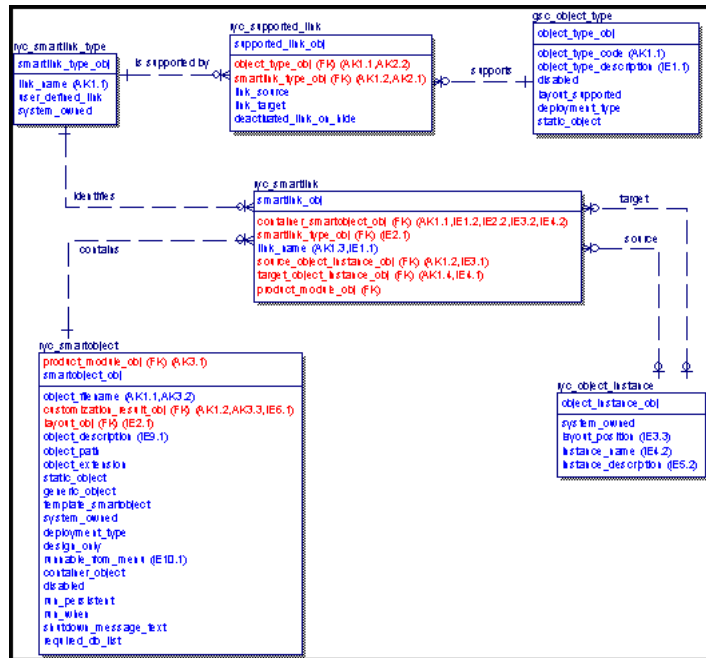


Figure 8–15: Link, SmartObject, and object instance tables

8.5 Folder page tables

To support the definition and use of different layout combinations, there is a layout table to define the basic characteristics of each distinct type of page layout. Other tables define which objects in a container are on which pages. This section contains information about the following tables:

- [The layout table](#)
- [The page table](#)
- [The page object table](#)

8.5.1 The layout table

The `ryc_layout` table defines the available page layouts for pages on SmartFolder™ windows, e.g. 1 browser with 1 toolbar underneath, *n* viewers above each other, 2 side by side viewers, 2 side by side browsers, etc. It also defines the available frame layouts for objects on a frame, e.g. 1 column, 2 columns, etc. The purpose of this table is to specify the program that is responsible for the layout when the window / frame is constructed or resized.

Most of the layout types defined in the layout table are in fact no longer actively used in building Progress Dynamics container windows. They are still used for windows in the framework tools themselves that have been built in earlier versions of the framework. The primary layout type now used is the Relative layout. However, the structure of the table and its fields remains the same. It has these fields:

- **layout_name** — This CHARACTER field is a unique name to identify the layout, for example, Relative.
- **layout_type** — This field is a three-character code to identify the type of layout, specifically the kinds of context in which it can be used. Currently defined values for the layout type are PAG = Page Layout, FRA = Frame Object Layout, BTH = Both.
- **layout_narrative** — This CHARACTER field is a free-form description for the layout.
- **layout_filename** — This CHARACTER field value is the filename to run, including its relative path, in order to actually perform the layout of the Objects on the page during initial construction and any later resize event. The Layout Manager procedure that uses this data is `ry/prc/rylayoutsp.p`, which is accessed from the dynamic SmartWindow procedure `ry/uib/rydyncontw.w` to do dynamic layout of container windows.
- **sample_image_filename** — This CHARACTER field contains the name and relative path of an image file that can be displayed illustrating the page layout, if one is available.
- **system_owned** — If this LOGICAL field is set to YES, this record may only be modified by users with a system-owned flag in their user profile.
- **layout_code** — This CHARACTER field is a suffix to add to internal procedures in the layout manager PLIP to identify the specific layout manager, e.g. ‘01’ would run the procedure version `resize01` for this layout. The Relative layout uses layout code ‘06’.

When containers of any kind are created to hold Object instances, page and page_object records are created to define the contents of each page of a container.

8.5.2 The page table

The **ryc_page** table holds a record for every page of every dynamic container in the application. This table defines the actual pages in the container. All containers must have at least one page, which is Page 0 and is always displayed. All objects on Page 0 are always displayed. If there are no other pages, then no tab folder is visualized. Example pages (identified by their **page_label**) could be Page 1, Page 2, Customer Details, etc. The table holds the Object ID of the associated **ryc_layout** record, which in turn identifies the procedure to be used to realize the page at runtime. The primary key of the **ryc_page** table also includes the container SmartObject Object ID in addition to its own page Object ID, even though the page Object ID is unique in its own right, so that pages can be specifically associated with the container on which they appear. The **ryc_page** table also has these other fields:

- **page_sequence** — This INTEGER field is the actual page number and determines the sequence in which the pages are displayed. There must always be a Page 0 and any objects on Page 0 are always displayed. Page 0 does not appear on a tab folder. If there are no other pages, then no tab folder is visualized.
- **page_label** — This CHARACTER field is the actual label to display on the tab folder describing the contents of the page. The label can be entered with an ampersand (&) denoting the shortcut key to select the page, e.g. &Details would enable the user to press ALT-D to select the page.
- **security_token** — This CHARACTER value defaults to the page label, without the ampersand, but may be different if required. The security token is used to automatically enable and disable folder pages according to user security permissions (via security allocation tokens).
- **enable_on_create** — If this LOGICAL field is set to YES, this folder page is enabled during an Add operation. If set to NO, then this folder page is disabled during an Add operation. This only affects the sensitivity of the folder tab for the page, not the Objects contained on the page.
- **enable_on_modify** — If this LOGICAL field is set to YES, this folder page is enabled during a modify operation. If set to NO, then this folder page is disabled during a modify operation. This will only affect the sensitivity of the folder tab for the page, not the Objects contained on the page.
- **enable_on_view** — If this LOGICAL field is set to YES, this folder page is enabled during a view operation. If it is set to NO, this folder page is disabled during a view operation. This only affects the sensitivity of the folder tab for the page, not the Objects contained on the page.

8.5.3 The page object table

The **ryc_page_object** table acts as a cross-reference table between pages and the objects on them. Remember that each object_instance defines a particular use of a SmartObject, so there is a one to one relationship between ryc_object_instance records and the ryc_page_object records that associate them with the pages of the container. Object ID fields relate this table to the Page table and the object_instance table; the container Object ID is also stored in this table. The only other field in the table is **page_object_sequence**, which is an INTEGER field that identifies the sequence of objects on the page.

Figure 8–16 is an excerpt from the repository database model and shows the relationships among the tables that support layouts and paging.

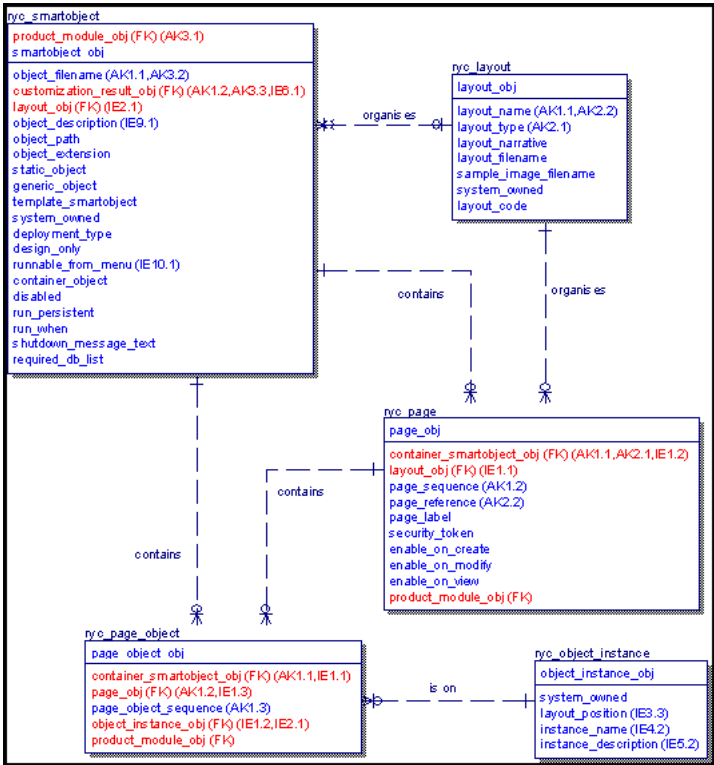


Figure 8–16: Folder page table diagram

8.6 Customization tables

A full discussion of the Progress Dynamics support for customization is beyond the scope of this document, but because the customization tables affect the use of the other Object tables, it is important to understand what the customization tables are and how they are used to modify Objects at runtime.

Customization support allows the developer to define types of end-user specifics that should affect the appearance or behavior of the application at runtime. This might include changes based on the user ID, the client display platform, the task the user is doing, or anything else that can be categorized. Customization basically means that certain attribute values can be changed for an Object depending on how it is used and who uses it.

There are three tables that provide support for customization:

- The **ryc_customization_type** table defines different categories of customization that are supported.
- The **ryc_customization_result** table stores specific result code values for a given customization type.
- The **rym_customization** table acts as a kind of cross-reference table that can be used to provide a more complex mapping between types and codes when no direct mapping is possible or practical.

Figure 8–17 is an excerpt from the database model and shows the three customization tables and how they relate to the SmartObject table.

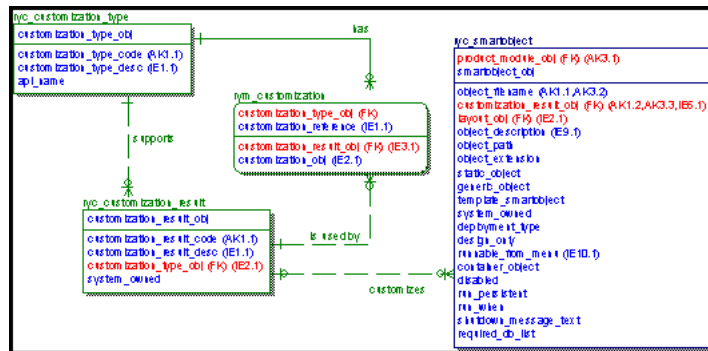


Figure 8–17: Customization tables

8.7 The customization type table

The **ryc_customization_type** table defines different categories of customization that are supported. The repository database comes with some types built in: UIType, Language, LoginCompany, System, User, and UserCategory. The type is used as a grouping mechanism for specific values or result codes that identify the end user's environment. You can populate these with values meaningful to your application environment, and you can also define any additional customization types your application may need. The table includes a field where you can store the name of a procedure to call to determine the value for the current user for a type. This is effectively the result code that allows the framework to identify the right customizations to apply.

The table contains these fields:

- **customization_type_code** — This CHARACTER field is a unique code to identify the type, such as UIType.
- **customization_type_desc** — This CHARACTER field is a description of the code, and must also be unique within the table.
- **api_name** — This CHARACTER field holds the name of a function to invoke that will return the value for the type for the current session. This might be a UI type, a user ID, a Language name, or whatever is appropriate to the type. The function can of course do whatever work it needs to in order to identify the correct value to return based on the type. This return value must map either directly to one of the result codes in the customization_result table, or to a value in the customization table that in turn can be mapped to a result code.

8.7.1 The customization result table

The **ryc_customization_result** table stores the specific result code values that are returned by the API function call for a given customization type. For example, the UIType customization type has several built in result codes defined in the repository:

- APP — AppServer
- BTC — Batch
- CUI — Character User Interface
- GUI — Graphical User Interface
- WBC — WebClient
- WBS — WebSpeed

You can define whatever result codes you need for different customization types required by your application. For example, the Language type could join to language result codes such as English and French. The User type could join to individual user IDs such as Anthony and Bruce. The UserCategory type could join to job types such as Manager and Engineer.

The customization_result table can therefore be viewed as a list of valid values for a customization type. Result codes need to be as distinctive as possible, because they must be unique within the customization_result table.

In addition to its own Object ID field, the customization_result table holds the Object ID of the associated customization_type. The customization_result table contains these other fields:

- **customization_result_code** — This CHARACTER field is the result code to match against what is returned by the API call. The result code must be unique across all customization types.
- **customization_result_desc** — This CHARACTER field is a description of the result code. It must also be unique within this table.

8.7.2 The customization table

In many cases the customization_result table is sufficient to define all the information the application needs to identify which customizations should apply for a given session. In some cases, however, it is not possible to provide a simple, direct mapping between the value returned by the API call for a given customization type and the result codes stored in the customization_result table. For example, the customization type may be one that defines general categories or groups, such as UserCategory. It may be useful to define result codes that are categories such as Manager or Engineer, but these may not be directly represented anywhere in the application for the customization type API to return. And it is not practical to have to define result codes for every single user when in fact the customizations are defined not at the User level, but at the UserCategory level.

The **rym_customization** table allows you to define a mapping to solve this problem when such a mapping is needed. It is in effect a cross-reference table between customization_type and customization_result, to define a one-to-many relationship between the two. In addition to its own Object ID, the table has Object IDs to join it to both the customization_type table and the customization_result table.

It has just one other field called **customization_reference**. This CHARACTER field holds the cross-reference value that supports the mapping between type and result. For the UserCategory example, this field could contain individual user IDs as returned by the API function for the type. Each of these IDs can in turn be mapped to a result code record for the category. This would allow you to have a single result code called Manager, and then have many customization_reference values that would map to that single UserCategory. When you define customizations, you define them against the Manager result code. When you run the API function for the type, it returns a value that you look up in the customization table as a reference value. You can then link from there to the result code to identify the general result code for this category of user.

8.7.3 Joining customizations to the SmartObject table

The customizations by themselves do not affect the application. The key data relationship here is between the customization_result table and the SmartObject table. This is where the run time effects of the customizations are defined.

As we noted in the discussion on the SmartObject table, the unique key for that table (beyond its Object ID) is not just the object_filename, but also the customization_result_obj field. For the standard Object behavior, the customization_result_obj is 0. For customized behavior, it is joined to the customization_result table using this field. This means that there must be a distinct SmartObject record for each customization result code that applies to the Object. If a Viewer, for example, has been customized to have different behavior (such as different field positions or different fields enabled or displayed) for each of six different user categories and three different user interface types, such as GUI, CUI, and WBS, then there will be a total of ten ryc_smartobject records representing that one Object: one for the default definition, six for the user categories, and three for the UI types.

Each of these ryc_smartobject records will have its own set of attribute_value records that define it. The SmartObject record with no customization_result_obj will define all the default attribute values. Each of the other SmartObject records will define only whatever specific attribute values have been modified for that customization. Thus the SmartObject records with a customization_result_obj are incomplete, and cannot be used by themselves to build the Object at runtime. Their values can only be applied to modify the base attributes defined by the default SmartObject record.

At runtime, the Customization Manager determines the result codes that apply to the session for each customization type by running the functions defined in the customization_type API_reference_name field. This yields a delimited list that remains available for the remainder of the session. The Repository Manager then combines the various attributes joined to all of the applicable SmartObject records. It does this by using the result code list to identify which customization_result records to join to the SmartObject table to retrieve all the applicable attribute_values. If multiple customizations apply, a function called getSessionResultCodes in the Customization manager determines the priority of them. Those with the highest priority are applied **last**, so that they take precedence if there are conflicting values for other customization types.

For example, suppose that user George logs in to a Progress Dynamics session running on WebSpeed using the new browser-based Progress Dynamics UI. George is a data entry clerk. Three result codes apply to this session:

- User: George
- UserCategory: Clerk
- UIType: WBS

Suppose that this list represents the priority order of the customization types. (Customization type priority can in fact be defined and modified by session type.) When George runs a dynamic Customer Maintenance window as part of the application, a Viewer inside the window has been customized in several different ways:

- George requested that the tab order of the fields be modified so that the data entry process for the Customer Comments field and the Contact field are more convenient for his data entry routine. He also requested that the Comments editor be made larger.
- George's group manager defined a customization to the same Viewer to disable the Contact field for clerks.
- The web application customization chief determined that the Comments editor for the dynamic HTML application interface should be made smaller to look better in that environment.

The result of all of this is that:

- The Comments field tab order is set as George requested it.
- The Contact field tab order is irrelevant because the field is disabled for George as a clerk.

- The size of the Comments editor is determined by the WBS definition because that takes precedence over George's personal request that is defined at the User level.

Customizations can be defined only for attributes of SmartObject records, not directly for individual instances. However, you **can** define customizations at the level of a container window, which can be a way of accomplishing the same thing. In other words, the customizations are always done starting at the level of a SmartObject record. If that SmartObject record represents a container window, then it points to not only its `attribute_value` records, but also to `object_instance` records, and their links and page records. Customizations can be done to the container by adding Objects to the container, or by modifying the attributes of objects inside the window for that window customization.

Figure 8–18 shows some of the records involved in defining these relationships.

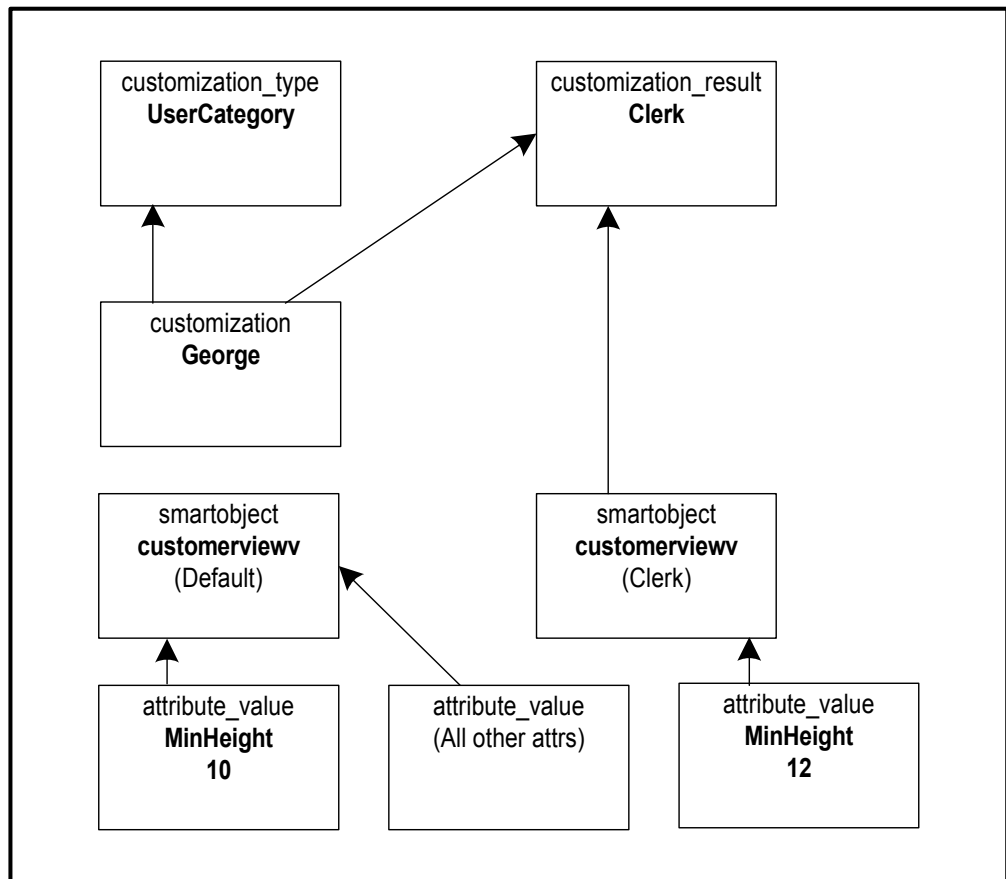


Figure 8–18: Customization example

This figure illustrates the following customization:

- George is a clerk, and customizations have been defined for the UserCategory Clerk, one of which is that the Customer Viewer `customerviewv` has a height of 12 rather than its default height of 10.
- When George logs in, the Customization Manager assembles a list of all the appropriate result codes for his session. One of those is for UserCategory. The customization table defines a `customization_reference` for George for the UserCategory `customization_type` that maps to the result code Clerk in the `customization_result` table. As a result, the Customization Manager stored **Clerk** as the session result code for the UserCategory type.
- When the system goes to run a window using the `customerviewv` Viewer, the Repository Manager must build a list of all the Viewer's attributes. To do this, it locates **all** the SmartObject records that apply for this session for the Viewer. In this case, that means both the default SmartObject, with no `customization_result` code, and also the SmartObject with the `customization_result` code for Clerk. It applies all the standard attribute values for the default version of the SmartObject (including the `MinHeight` value of 10), and then applies the attribute values for the customization, one of which overrides the `MinHeight` and sets it to 12. These values are then sent to the Layout Manager to use to build the window.

8.7.4 Tools to support customization maintenance

A full description of the customization process is beyond the scope of this chapter, but there are some tools where you can define the customizations.

In the Progress Dynamics Development menu, there is a Customization Maintenance utility in the Objects menu shown in [Figure 8–19](#).

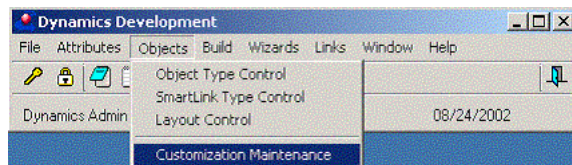


Figure 8–19: Customization maintenance utility

Figure 8–20 provides a tree view showing the various levels of customization.

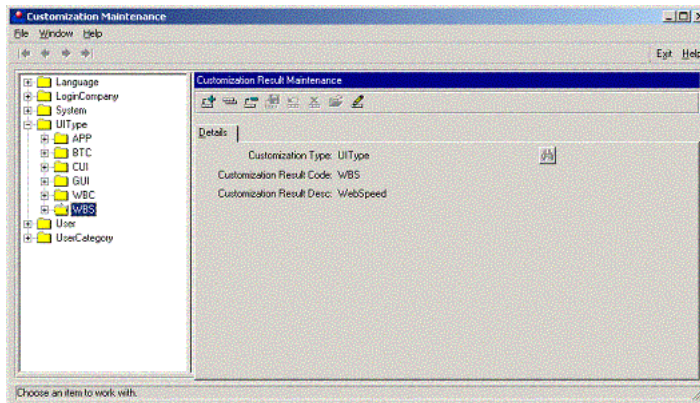


Figure 8–20: Levels of customization

In addition, the Container Builder allows you to specify a result code for an Object or a window, so that you can define attribute values specific to that Object.

8.8 Using the Repository Manager

This section describes the following Repository Manager topics:

- [The client cache](#)
- [Repository Manager API](#)
- [Retrieving related information for an object](#)
- [How objects are instantiated using prepareInstance](#)

8.8.1 The client cache

As a Progress Dynamics runtime client or WebClient session runs, Object descriptions are cached on the client in temp-tables. The data in these tables is then accessed by the drivers such as the dynamic window procedure rydyncontw.w that realize the client portion of the application at runtime.

This section summarizes the temp-tables that make up the client cache.

The cache_Object table

This temp-table contains information about Master Object records as well as contained instances. This table contains information about the static (physical) object required to instantiate a dynamic Object, the logical object name and other data, such as the result code, run attribute, etc. There will only ever be one record in the cache for any given Master Object (and thus for its contained instances) at any given time. This table is (roughly) a combination of the information from the ryc_smartobject and ryc_object_instance repository tables.

The cache_Object table is joined to records in the next four tables, cache_ObjectPage, cache_ObjectLink, cache_ObjectPageInstance, and cache_ObjectUIEvent, using the key field tRecordIdentifier.

The following code sample shows a slightly simplified definition of the cache_Object temp-table (without formats and indexes). All the cache temp-tables are defined in the include file ry/app/ryobjretri.i:

```

DEFINE TEMP-TABLE cache_Object
  FIELD tLogicalObjectName          AS CHARACTER /* Logical Object Name */
  FIELD tUserObj                   AS DECIMAL   /* User Object ID */
  FIELD tResultCode                AS CHARACTER /* set of result codes */
  FIELD tRunAttribute              AS CHARACTER /* Run Attribute */
  FIELD tLanguageObj              AS DECIMAL   /* Language Object ID */
  FIELD tRecordIdentifier          AS DECIMAL   /* Instance Id key field */
  FIELD tContainerRecordIdentifier AS DECIMAL   /* Container Record Id */
  FIELD tMasterRecordIdentifier    AS DECIMAL   /* Master Record Id */
  FIELD tClassName                 AS CHARACTER /* Class Name */
  FIELD tClassTableName            AS CHARACTER /* Class Table Name */
  FIELD tClassBufferHandle         AS HANDLE   /* Class Attr Buffer Handle */
  FIELD tContainerObjectName       AS CHARACTER /* Container Name */
  FIELD tInstanceIsAContainer      AS LOGICAL   /* Is this a Container? */
  FIELD tObjectPathedFilename     AS CHARACTER /* Pathed Filename */
  FIELD tObjectInstanceHandle     AS HANDLE   /* Object Instance handle */
  FIELD tObjectInstanceObj        AS DECIMAL   /* Object Instance
ObjectID*/
  FIELD tObjectInstanceName       AS CHARACTER /* Object Instance Name */
  FIELD tObjectInstanceDescription AS CHARACTER /* Instance Description */
  FIELD tDbAware                  AS LOGICAL   /* Is Object DB Aware? */
  FIELD tLayoutPosition           AS CHARACTER /* Layout Position */
  FIELD tCustomSuperProcedure     AS CHARACTER /* Super Procedure */
  FIELD tCustomSuperHandle        AS HANDLE   /* Super Proc. Handle */
  FIELD tDestroyCustomSuper       AS LOGICAL   /* Destroy Super on exit? */
  FIELD tInstanceOrder            AS INTEGER   /* Instance Order */
  FIELD tPageNumber               AS INTEGER   /* Page Number */
  FIELD tSmartObjectObj           AS DECIMAL   /* Smartobject Object ID */
  FIELD tSdoSmartObjectObj        AS DECIMAL   /* SDO Object ID if any */
  FIELD tSdoPathedFilename        AS CHARACTER /* SDO Pathed Filename */
  FIELD tInheritsFromClasses      AS CHARACTER /* Class list */

```

A few fields merit additional explanation:

- **tRecordIdentifier** — This field is the unique DECIMAL key for each Object record in the temp-table.
- **tContainerRecordIdentifier** — This field holds the Record Identifier of the cache_Object record that acts as a container to the current Object record.
- **tMasterRecordIdentifier** — This is the Record Identifier of the cache_Object record that is the Master of the current instance record.
- **tClassBufferHandle** — This is the buffer handle that points to the class temp-table containing the attributes for this Object.
- **tSdoSmartObjectObj** — This is the design-time SDO associated with the Object, if any.
- **tInheritsFromClasses** — This field stores a list of the class names of all classes that this class inherits from.

The cache_ObjectPage table

This temp-table contains information about Pages within a container. It uses the information in the repository database table ryc_page. This is the temp-table definition for this table:

```
DEFINE TEMP-TABLE cache_ObjectPage
  FIELD tRecordIdentifier      AS DECIMAL    /* Record Identifier key
field */
  FIELD tPageNumber           AS INTEGER     /* Page Number */
  FIELD tPageLabel            AS CHARACTER   /* Page Label */
  FIELD tLayoutCode           AS CHARACTER   /* Layout Code */
  FIELD tPageInitialized      AS LOGICAL     /* Is the Page Initialized? */
  FIELD tPageObj              AS DECIMAL     /* Page Object ID */
```

The **tPageObj** Object ID is used for finding this record from the page object instance.

The cache_ObjectLink table

This temp-table contains link information, derived from the repository table ryc_smartlink. This is the definition of the table:

```

DEFINE TEMP-TABLE cache_ObjectLink
  FIELD tRecordIdentifier      AS DECIMAL    /* record Identifier key
field */
  FIELD tSourceObjectInstanceObj AS DECIMAL  /* Link Source Object ID */
  FIELD tTargetObjectInstanceObj AS DECIMAL  /* Link Target Object ID */
  FIELD tLinkName              AS CHARACTER /* Link Name */
  FIELD tLinkCreated           AS LOGICAL   /* Has the link been created? */

```

The cache_ObjectPageInstance table

This temp-table contains information about Object instances on a given page, derived from the repository table ryc_page_object. This is the definition of the table:

```

DEFINE TEMP-TABLE cache_ObjectPageInstance
  FIELD tRecordIdentifier      AS DECIMAL    /* Record Identifier key */
  FIELD tPageNumber           AS INTEGER    /* Page Number */
  FIELD tObjectInstanceObj     AS DECIMAL    /* Object Instance Object ID */
  FIELD tObjectInstanceHandle  AS HANDLE    /* Object Instance Handle */
  FIELD tObjectInstanceName    AS CHARACTER /* Object Instance Name */
  FIELD tObjectTypeCode        AS CHARACTER /* Class Code */
  FIELD tLayoutPosition        AS CHARACTER /* Layout Position */

```

The cache_ObjectUIEvent table

This temp-table contains information about UI events, derived from ryc_ui_event. This is the definition of the table:

```

DEFINE TEMP-TABLE cache_ObjectUiEvent
  FIELD tClassName            AS CHARACTER /* Class Name */
  FIELD tRecordIdentifier     AS DECIMAL
  FIELD tEventName           LIKE ryc_ui_event.event_name
  FIELD tActionType          LIKE ryc_ui_event.action_type
  FIELD tActionTarget        LIKE ryc_ui_event.action_target
  FIELD tEventAction         LIKE ryc_ui_event.event_action
  FIELD tEventParameter      LIKE ryc_ui_event.event_parameter
  FIELD tEventDisabled       LIKE ryc_ui_event.event_disabled

```

NOTES:

- **tClassName** — For the master class, this field is set equal to `buildDenormalisedAttributes` temporarily, and then the actual class name.
- **tRecordIdentifier** — This is the Record Identifier for the table within the class name; the value 0 is used for the master.

The cache_<classname> tables

There is a dynamic temp-table containing attribute information for each defined class or Object Type in the repository, with the name **class_<classname>**. Each of these temp-tables has a field for each attribute defined for the class, whose initial value is the default attribute value defined for the attribute for that class, using the Attribute Value Maintenance and Object Type Control utilities. These temp-tables are dynamic because they are defined at runtime based on the set of attributes associated with each class. When you use the framework utilities to add attributes to a class and give them an initial value for the class, it is this information that the framework uses to create the temp-table for each class at runtime. In this way attributes can be

added to the classes in the framework without the need to define them anywhere in source code or to recompile any procedures in order for them to become a part of all the dynamic Objects in that class.

The example in the section on Using the Profile Manager in [Chapter 6, “Using the Progress Dynamics Managers,”](#) illustrates how you can define new attributes and associate them with Object Types.

The ttClass table

This temp-table stores the names of all the classes (Object Types) and their associated temp-table names, along with the handles to the default buffers of the temp-tables that have been created to store each class's default attributes. As described above, a separate dynamic temp-table is defined for each class, containing a separate field for every attribute in the class, plus some extra control fields. The table for each class is called `cache_ + the class name`, e.g. `cache_dynview`.

The ttClass table is updated in the function `createClassCacheRecord` in the Repository Manager, which is called from the `createClassCache` procedure, which caches the attribute values for an Object Type.

Record identifiers

Every `cache_Object` record has a unique key, called **tRecordIdentifier**. This DECIMAL value is used as a foreign key to join the Object record to other tables. This value is also stored at runtime in the Object property called **InstanceID**.

The **tContainerRecordIdentifier** field in the `cache_Object` table contains the Record Identifier of the `cache_Object` record that represents the container Object on which the current `cache_Object` is an instance. This is zero when the `cache_Object` represents a Master object.

The **tMasterRecordIdentifier** field in the `cache_Object` table is the Record Identifier of the `cache_Object` representing the Master object of this `cache_Object` record. This value is the same as the Record Identifier if this cache record represents a Master object.

8.8.2 Repository Manager API

These are a few of the most useful methods in the Repository Manager API. The principal body of code for the Repository Manager is in the include file `ry/app/ryrepmgrp.i`.

cacheObjectOnClient

The function **cacheObjectOnClient** checks if the requested object exists in the client-side cache. If the requested object is not in the cache, it is retrieved from the repository and placed in the cache. This function returns a logical value signifying the success of the operation.

The `cacheObjectOnClient()` function ensures that all the requested Objects, their contained instances, their associated Master Objects, and all related information is placed in the cache.

When `cacheObjectOnClient` is called, it first checks to see if the requested Object is cached. If it is, then the relevant record is positioned to, and nothing further happens. If the record does not exist in the cache, then the function needs to retrieve the data from the repository and place it into the cache. If the repository database is connected, then `cacheObjectOnClient` runs **bufferFetchObject**. This performs the actual retrieval, and puts the objects into a cache using **putObjectInCache**.

If the repository database is not connected locally, then the function runs **doServerRetrieval**. This procedure runs **serverFetchObject** on the `AppServer`. `serverFetchObject` runs **bufferFetchObject** and puts the information into temp-tables, which are then returned to the caller. `doServerRetrieval` then calls `putObjectInCache`, which ensures that the Objects are in the cache. The last thing that `cacheObjectOnClient` does is to do a FIND on the requested record in the cache so that it is AVAILABLE to the caller.

The `cacheObjectOnClient` function takes the following INPUT parameters:

- **pcLogicalObjectName** — This CHARACTER parameter is the name of the Object to cache. If this is a container, then all of its contained Objects are cached as well so that the entire container is ready to be created on the client.
- **pcResultCode** — If the client is using a customization, then this CHARACTER parameter is the Result Code for that customization. If this parameter is unknown or blank, then the function retrieves the `SessionResultCodes` session property to establish any default result code(s).
- **pcRunAttribute** — If the Object requires a Progress Dynamics Run Attribute as an input value, this CHARACTER parameter holds that value. If this argument is the unknown value, the function retrieves the `RunAttribute` session property to see if that defines a default run attribute.
- **plDesignMode** — This LOGICAL parameter is True if the caller is running in design mode, rather than runtime. In design mode, the client cache is always cleared on each call so that the developer is assured of having the latest changes made to the repository data by the development tools.

The methods `bufferFetchObject`, `serverFetchObject`, `doServerRetrieval`, and `putObjectInCache` should be considered private to the Repository Manager. You should not normally run these directly.

Functions used to retrieve cache table buffer handles

All operations that need to use the cache information use the one set of temp-tables where all cached data resides. There is a set of functions to return the buffer handles of the tables in the cache. These are:

- **getCacheObjectBuffer**
- **getCachePageBuffer**
- **getCachePageInstanceBuffer**
- **getCacheLinkBuffer**
- **getCacheUIEventBuffer**
- **getCacheClassBuffer**

With the except of `getCacheObjectBuffer` and `getCacheClassBuffer`, these functions take no INPUT parameters. Each returns the HANDLE of the requested buffer.

getCacheObjectBuffer() takes as an input parameter an optional Object Instance ID and returns the buffer handle of the cache_Object table. If the input value is a non-null Instance ID, the function finds the relevant record in the cache_Object table before returning the buffer handle. This is important because it means that, with an Instance ID, you can get back an AVAILABLE buffer handle to the record in the cache. This saves having to do FIND-FIRST() to position to the desired record yourself.

getCacheClassBuffer() takes as an input parameter the name of a class (the Object Type code) and returns the buffer handle of the ttClass temp-table. With this you can read class attributes without needing to start from a particular Object. If you pass in a single, valid class name, the buffer that it returns points to the ttClass record that corresponds to that class. If the class name passed in to the function does not yet exist in the ttClass table portion of the client cache, the function retrieves it from the repository. You can also pass in an asterisk (*) to represent all classes, or a comma-separated list of class names, and all the valid classes in the list are retrieved. However, in this case no ttClass record is positioned to at the end of the function, since there is no single right record to find.

Using the API to retrieve class attributes and values

To be able to access the attributes for a class, you need to first get the handle of the attribute buffer. The ttClass buffer handle that getCacheClassBuffer returns is the handle of the record for the class in the ttClass table, not the handle of the attribute table for the class. The buffer handle of the attribute temp-table is stored in the classBufferHandle field of the ttClass record.

This code sample shows how you can retrieve all of the attributes and their default values for a given class, for example the dynamic SDO or **DynSDO**. The first call is to the function getCacheClassBuffer in the global handle of the Repository Manager. This ensures that the DynSDO class is in the client cache, and returns the handle of the ttClass buffer. If the data is not already there, it is retrieved from the server. The function does a FIND of the correct record in the cache when it is done:

```
DEFINE VARIABLE httClassBuffer          AS HANDLE    NO-UNDO.
DEFINE VARIABLE hClassAttributeBuffer AS HANDLE    NO-UNDO.
DEFINE VARIABLE iFieldCount             AS INTEGER   NO-UNDO.
DEFINE VARIABLE cAttributeName          AS CHARACTER NO-UNDO.
DEFINE VARIABLE cClassAttributeValue   AS CHARACTER NO-UNDO.

ASSIGN httClassBuffer = DYNAMIC-FUNCTION("getCacheClassBuffer":U IN
                                         gshRepositoryManger, INPUT "DynSDO").
```

The classBufferHandle field holds the handle of the buffer handle for the specific dynamic class table for the class's attributes, in this case cache_DynSDO:

```
ASSIGN hClassAttributeBuffer =  
      httClassBuffer:BUFFER-FIELD("classBufferHandle"):BUFFER-VALUE.
```

The variable hClassAttributeBuffer now points to the **cache_DynSDO** buffer. Now you can get all the attribute values for the class. The INITIAL value of the field represents the default value of each attribute for this class:

```
DO iFieldCount = 1 TO hClassAttributeBuffer:NUM-FIELDS:  
  ASSIGN cAttributeName =  
        hClassAttributeBuffer:BUFFER-FIELD(iFieldCount):NAME  
        cClassAttributeValue =  
        hClassAttributeBuffer:BUFFER-FIELD(iFieldCount):INITIAL  
        .  
  /* Process the attribute values */  
END.
```

One important thing to keep in mind is that with only one set of buffers to access everything in the cache, you need to be aware of the fact that records can go out of scope. So if you find a cache_Object record, and then run an Object based on it, chances are that by the time the run statement returns to your procedure, the cache_Object record you were positioned to is no longer current. This is mainly because the **prepareInstance** function (described later) is run for every SmartObject, and it interacts with the cache_Object records. It is very easy to reposition the cache_Object record back to the desired record. If you pass getCacheObjectBuffer() a valid

Record Identifier (Instance ID), it repositions the cache_Object buffer to the correct record. Alternatively, you can use your own set of defined buffers to give you additional buffers whose records won't be changed out from under you.

Accessing repository data without the cache

If you would rather not use the cache, as would be the case in application design tools, then when making calls from a development tool, you can call **serverFetchObject** directly. This procedure returns the requested information in table form. This means that you are guaranteed to get the data as it is in the repository at any given moment. However, what you lose is the caching. A full description of serverFetchObject and other calls that access the data more directly is beyond the scope of this chapter. You should study the code in ry/app/ryrepmgrp.i if you are building a development tool that you feel needs to use these internal calls. Be aware that serverFetchObject, for example, has a complex calling sequence that includes a large number of TABLE-HANDLES, each one for a separate temp-table for attribute data related to a particular class of Object.

Reasons not to use the cache would be if you want to use the information for design purposes in a tool. Running `cacheObjectOnClient()` returns the object in unaggregated form: each result code is separated. If you want to design with aggregated data, then you would need to run this procedure, or clear the cache with every call.

You can also use the `serverFetchObject` procedure when you want to retrieve repository data for external use, for example in a tool that does some form of analysis or reporting of the repository data.

8.8.3 Retrieving related information for an object

This section outlines how you retrieve various kinds of information in the cache that is related to an Object, and which completes the definition of the Object.

Retrieving the instance ID for an object

Before finding any related information, you need to have the Instance Id (the **tRecordIdentifier** field value) of the record whose associated information you require. There are several ways of doing this.

If the object is running, you can retrieve the Instance ID as an Object property using the ADM property include file syntax. In this example, the Object handle is in the variable `hObject` and the Record Identifier is retrieved into the variable `dInstanceId`:

```
{get InstanceId dInstanceId hObject}
```

Or you can use the equivalent function call:

```
dInstanceId = DYNAMIC-FUNCTION('getInstanceId' IN hObject).
```

Alternatively, the Object that you request using `cacheObjectOnClient()` is available immediately after the call to that function. You can then use a statement such as the following to return the buffer handle to the `cache_Object` temp-table, which will also be positioned to the requested record:

```
ASSIGN hObjectBuffer =  
    DYNAMIC-FUNCTION("getCacheObjectBuffer":U IN gshRepositoryManager, INPUT  
    ?).
```

After finding the cache_Object buffer, do a FIND based on your own criteria to position to the relevant record. Once you have found an available hObjectBuffer record, you can populate the dInstanceId variable using a statement such as this:

```
ASSIGN dInstanceId =  
      hObjectBuffer:BUFFER-FIELD("tRecordIdentifier":U):BUFFER-VALUE.
```

Getting the Instance Id is the first step to getting all the other related information out of the cache.

Retrieving other object information

To get Page, Page instance, Link and UI event information for a particular Object, first retrieve the buffer handle for the relevant cache buffer, using one of the functions getCachePageBuffer, getCachePageInstanceBuffer, getCacheLinkBuffer, or getCacheUIEventBuffer. In this code example, hBuffer is the buffer handle retrieved using one of those function calls. Create a dynamic query on that table, with a WHERE clause such as this:

```
hQuery:QUERY-PREPARE(" FOR EACH ":U + hBuffer " WHERE ":U  
+ hObjectBuffer:NAME + ".tRecordIdentifier = " + QUOTER(dRecordIdentifier) ).
```

Retrieving attribute values for an object

To retrieve attribute values you first need to obtain the Instance ID (the tRecordIdentifier field value) from the cache_Object record. Remember that the correct cache_Object record is AVAILABLE after a call to cacheObjectOnClient(). Otherwise, if you have the Object's Instance ID, you can make a call to getCacheObjectBuffer(), passing in the Instance ID. This ensures that the correct record is available to you. The tClassBufferHandle field contains the buffer handle of the record that contains the attributes for the object.

Using a statement such as the following, you can retrieve the attributes for the specific object. Note the use of the recently introduced dynamic FIND-FIRST method on a buffer handle, which allows you to position the buffer to a record using a dynamic WHERE clause:

```
hAttributeBuffer:FIND-FIRST(" WHERE ":U  
+ hAttributeBuffer:NAME + ".tRecordIdentifier = " + QUOTER(dRecordIdentifier)  
).
```

You can now retrieve the values for the relevant field, as shown in the code sample earlier in this section.

Retrieving an object's contained instances

There are two primary ways of getting all the contained instances for a container Object. You can query all `cache_Object` records where either:

- The **tContainerRecordIdentifier** equals the Instance ID of the container Master object.
- The **tContainerObjectName** equals the logical object name of the container object. This second method requires you to also include in your query the user, language, result code and run attribute, as well as checking that the object instance Object ID is zero.

The first alternative is simpler, and is also less programming work since all that is needed is the Instance ID, which is readily available. The other information required to use the second alternative requires various calls and possibly further calculation.

How attributes are stored

Each class has its own attribute temp-table, with a field for each attribute in the class. This includes all attributes inherited by the class from its parent class(es). The initial value of each attribute field is the initial or default value of the attribute for that class. The temp-table is created dynamically by **buildClassCache()** and is called **cache_<ClassName>**. These tables and their buffer handles are then cached in the **ttClass** temp-table.

There are also a couple of fields which the `cache_<ClassName>` table uses internally:

- **tRecordIdentifier** — this field contains the Record or Instance ID of the `cache_Object` record to which the attributes belong.
- **tWhereStored** — this field contains an integer indicating at what level the attribute is stored (Class, Master or Instance) and whether customization has been applied to the attribute. The **getWhereStoredLevel()** function returns a character description of this value, for a given field within an attribute buffer.

8.8.4 How objects are instantiated using `prepareInstance`

This section outlines the steps that take place to bring a container window or other container Object into being at runtime, using the cached repository information. If you want to understand more about exactly how containers are created, you can study the relevant code in the ADM2 support procedure `src/adm2/container.p`, as well as the dynamic window procedure `ry/uib/rydyncontw.w`.

When an object is launched, whether it is by `launchContainer`, `constructObject` or just from the editor, a Repository Manager function called `prepareInstance` is called from the main block of the base ADM property include file `smrtprop.i`. This is just about the earliest that anything can run in the ADM world, and the call needs to be as early as possible, so that the ADMProps temp-table where all of each Object's attributes are held is built as soon as possible.

The `prepareInstance` function attempts to determine the logical name of the object being launched, since this is what is used to retrieve the objects from the Repository. It does this by calling a function called `getCurrentLogicalName` in the procedure that launched the object being run. This function is defined in the Session Manager (which contains `launchContainer`, the primary starting point for objects run from the menu), the dynamic container procedure `rydyncontw.w`, the dynamic viewer `rydynvieww.w`, and the dynamic treeview `rydyntreew.w`.

The value returned by `getCurrentLogicalName` can be in one of two forms (errors and blanks excepted):

- The actual value of the Object's logical name, equivalent to the repository field `ryc_smartobject.object_filename`.
- A value in the form of `InstanceId=<value>`.

If there is a single valid Object name, rather than the form `InstanceId=*`, the relevant data is retrieved from the cache, or the repository, using `cacheObjectOnClient()`. Once this data is returned, the ADMProps temp-table is constructed based on the Object's attribute buffer.

The InstanceID ADM attribute is set at this point. Each Object that is run will have a unique InstanceID.

The reason for the two different forms of the return value of `getCurrentLogicalName` is this:

When you are instantiating a Master Object (which would be the container of other Objects) the function returns the actual logical Object name that you want to retrieve. When you want to retrieve Object instances **within** a container, the function returns the `InstanceId=` form. This is because there may be many records with the contained instance's logical Object name. If a toolbar is placed on many windows, for example, there will be one record for the Master, and one for each Object instance. By contrast, there will only ever be one record with a particular Instance Id (`tRecordIdentifier`).

Here is a rough example of how a window is instantiated:

Consider a window called **window1**. It contains an SDO called **fullo1** and a browser called **browser1**. If you launch window1 from a menu, launchContainer sets the CurrentLogicalName value to window1 before running the dynamic container object (the procedure rydyncontw.w). When rydyncontw.w runs, it picks up the logical Object name and passes that into the function cacheObjectOnClient(), which then retrieves into the cache the five sets of Object records that make up the window:

- window1 itself (the Master Object)
- fullo1 (the Master and Instance of fullo1 on window1)
- browser1 (the Master and Instance of fullo1 on window1)

Two standard ADM2 support procedures in the container class have been extended for Progress Dynamics to create dynamic containers from cached repository data as opposed to simply running static procedure Objects, as would be the case with a standard Version 9 ADM2 application. These are **createObjects** and **constructObject**.

As window1 is created, the ADM container startup code runs **createObjects**. This procedure has the responsibility of creating the objects that are to appear on window1. createObjects retrieves the InstanceID property of window1 and then makes sure that the correct cache_Object record is available, by calling getCacheObjectBuffer() and passing in the InstanceID. This InstanceID is used to build a query to loop through all of the contained instances (... WHERE tContainerRecordIdentifier = dInstanceID).

The framework then runs **constructObject** to construct browser1 and fullo1. The code needs to make sure that the correct set of attributes is used for the instance of each Object on window1, so it sets the CurrentLogicalName to **Instanceid=** since this is already available.

Index

A

Action rules

- alternate image 3–42
- defined for menu/toolbar 3–42
- enable 3–42
- functions 3–50
- hide 3–42
- properties 3–48
- syntax 3–47

Action target 1–18

Action type 1–17

Actions, enabled/disabled 3–42

Add query where function 5–35

ADM2 8–4

- code 8–7
- support procedures 8–61

ADM2 customization 2–10

ADM2 properties 8–17

Alternate image rule 3–42, 3–45

API

- manager 6–19
- multiTranslation call 6–76

- repository manager 8–53
- universal 6–44

AppBuilder

- dynamic launcher 6–5
- property sheet 1–8

AppBuilder, adding new class to 2–21

Applying security restrictions 6–80

AppServer, managers stateless connection 6–3

Architecture

- Progress Dynamics 8–4
- Progress Dynamics managers 6–14

askQuestion procedure 6–65

assignPage property 5–51

Attribute definitions 8–16

Attribute level 8–59

Attribute maintenance window 3–24

Attribute tables 8–17, 8–19

- relationships 8–18

Attribute value level 3–22

Attribute values 8–22

class level 8–30, 8–31

instance level 8–33

levels 8–24

object 8–58

object instance level 8–28

object master level 8–26

table 8–23

attribute_group_name field 8–19

attribute_group_narrative field 8–19

attribute_group_obj field 8–19

attribute_label field 8–19

Attributes

adding 6–52

COLUMN-MOVABLE 6–71

control 3–23

defining 6–50

hierarchy 3–24

minHeight 3–10, 3–11

object type level 8–26

resize vertical 3–11

value

changing 3–25

Attributes, adding to the Repository 2–15

Attributes, creating 2–15

Attributes, naming 2–18

Attributes, removing invalid 2–26

Attributes, removing redundant 2–26

Attributes, support functions 2–21

Audience xv

Auto commit property 5–51

B

Base procedure 1–6

BrowseColumnsMovable property 6–49

Browser

attributes 1–49, 6–71

custom super procedure 1–45, 6–54

new properties 6–49

profile data 6–72

properties 1–46

Browser handle 5–7

Buffer handle 5–7, 8–50

Business logic 1–4, 1–35

procedure 1–43

running 1–41

Business logic procedure 8–9

Buttons, disabling and hiding 3–28

C

Cache

clearing 7–28

procedure server-side 7–25

Cache APIs 4–14

Cache key 4–15

Cache types 4–2

Cache, shared 4–5

Cache_object table 8–49

CacheDuration property 4–6

cacheObjectOnClient function 8–53

Caching, application data 4–1

Caching, combos and 4–9

Caching, field-based 4–7

Caching, lookups and 4–8

Caching, SDO-based 4–6

canNavigate function 3–44, 3–50

- Changing
 - attribute value 3–25
 - profile/preferences 6–47
- Character properties 5–3
- Check value 3–38
- checkProfileDataExists 6–57
- Class
 - attribute value 3–24
 - behavior 1–7
 - inheritance 8–50
 - sub-class 2–12
- Class hierarchy 2–4
- Class level 8–31
- class_smartobject_obj field 8–8
- Clear client cache procedure 7–28
- clearCache function 4–17
- Client
 - cache 8–48
 - calls, minimizing 1–3
 - code, super procedures 1–3, 1–5, 7–5
 - logic 1–35
 - logic API 1–50
 - logic support 1–22
 - proxy 7–4
- Client logic
 - functions 1–50
 - object qualification 1–50
- Client logic API 1–50
- Close query function 5–22
- Code
 - client 1–3
 - client/server 7–5
 - common for client/server 6–15
 - custom 5–17
 - custom super procedure 1–10
 - server 6–21
- Collect changes event 5–50
- Column attribute functions 5–9
- COLUMN-MOVABLE attribute 6–71
- Configuration file manager 6–6, 6–39, 6–44
- Connection manager 6–9, 6–41, 6–44
- Connections between records 8–16
- Container
 - handle property 5–7
 - type 5–4
- container_object field 8–11
- Containers, pages defined 8–39
- Context
 - database table 6–23
 - management 6–23
 - table 6–25
- Control
 - attribute 3–23
 - object type 3–23
- Control window
 - manager type 6–6
 - service type 6–7
 - session type 6–7
- createBrowsePopupMenu 6–60
- createSharedBuffer function 4–16
- Creating
 - dynamic objects 5–13
 - manager type 7–8
- Cross-reference table 8–40
- Cross-reference value 8–44
- Custom classes 2–1
- Custom classes, migrating 2–60
- Custom SmartLinks 5–30

- Custom SmartObject object field 8–12
- Custom super procedures 1–6, 3–30
 - browser 1–45, 6–54
 - code 1–10
 - defining logic 1–11
 - registering 3–30
 - viewer 3–31
- Customer Maintenance viewer 3–7
- Customization
 - categories 8–42
 - maintenance tools 8–47
 - result table 8–42
 - table 8–43
 - tables 8–41
- Customization manager 6–14, 8–45
- Customization result object 8–13
- Customization type table 8–42
- Customizing function and property rules 3–51

D

- Data
 - browser profile 6–72
 - caching 7–21
- Data available procedure 3–34, 5–21
- Data handle property 5–51
- Data links, disabling 3–14
- Data object functions 5–11
- Data visualization property 3–50
- data_type field 8–19
- DataField objects 8–2
- DataModified property 3–49
- DB-AWARE setting 7–3
- DB-REQUIRED setting 7–4
- Default session type 7–11
- Defining attributes 6–50
- Defining properties 6–26
- Defining super procedure 1–8
- deletePage procedure 5–52
- Deployment type 8–7, 8–12
- Derived value 8–21
- destroyObject procedure 5–24
- destroySharedBuffer function 4–17
- Disable
 - actions 3–28
 - buttons and menu items 3–28
 - fields 5–21
 - object 5–19
 - security 8–7
- Disable access 8–12
- DisableActions function, when not to use 3–42
- Disabled actions 3–37
- disableFields procedure 3–35
- DisplayedFields property 1–46, 5–5
- DisplayObjects procedure 5–20
- Distributed applications 1–3
- Distributed environment, performance issues 1–4
- Dynamic container, starting 6–28
- Dynamic launcher 6–5
- Dynamic object 5–13, 8–7
 - generator 1–14
- Dynamic property sheet 3–19

Dynamic viewer, testing 1–22

Dynamics

Manager 7–1
startup 7–11

E

Editable property 3–49

Enable rule 3–42, 3–44

EnableActions function, when not to use
3–42

EnableFields procedure 3–35, 5–21, 7–38

EnableObject procedure 5–19

ERwin diagram 8–3, 8–6

Event

action 1–19
disabled 1–19
name 1–17
parameter 1–19
procedure 5–1
procedures, defining for popup menu
6–62

Event information 8–58

eventMenuItemChoose procedure 6–63

ExitObject procedure 5–24

Extending

object type 8–8
SmarObject 8–12

F

Fetch procedures 5–23

fetchDataFromCache function 4–16

Field attributes, functions 5–8

Field edit

data 7–29
information 7–18
manager 7–17

Field values, functions 5–8

FieldHandles property 1–46

Fields

enabled 5–5
enabling/disabling 3–35
handles 5–5

findCacheItem function 4–16

Folder, adding to test window 3–4

Folder page tables 8–37

Foreign fields 5–33

property 3–8

Function reference 1–32, 1–33

Functions

action rule 3–47
assignQuerySelection/removeQuerySele
ction 5–36
cache table buffer handles 8–54
cacheObjectOnClient 8–53
canNavigate 3–44, 3–50
closeQuery 5–22
column attribute 5–8
data object 5–11
getManagerHandle 6–40
getPhysicalSessionType 6–43
getPropertyList 6–26
getUserProperty 5–55
hasActiveAudit 3–50
hasActiveComment 3–50
instancePropertyList 5–54
isICFRunning 6–39
linkHandles 5–31
linkProperty 5–31
openQuery 5–22
prepareInstance 8–59
prepareQuery 5–23
propertyType 5–55
resortQuery 5–34

- setOpenQuery 5–35
- setPropertyList 6–26
- setQuerySort 5–34
- setQueryWhere 5–34
- setUserProperty 5–55
- targetPage 5–53
- translatePhrase 6–74

G

- General manager 6–12, 6–84
- get function 8–20
- getEntityDescription procedure 6–21
- getManagerHandle function 6–40
- getPhysicalSessionType function 6–43
- getProfileData 6–58
- getPropertyList function 6–26
- getTranslation 6–75
- getUserProperty function 5–55
- Group assign link 3–15
- gsc_object_type table 8–7

H

- Handles 5–7
 - basic managers 6–40
 - procedure 6–14
 - static 6–40
- hasActiveAudit function 3–50
- hasActiveComment function 3–50
- Hide action rule 3–42
- HideObject procedure 5–20
- HidePage procedure 5–52
- Hiding buttons and menu items 3–28

I

- IN TARGET-PROCEDURE 1–25
- Include files 1–41, 8–17
- Index information 5–11
- Init page procedure 5–52
- Initialization properties 5–25
- InitializeObject procedure 3–30, 5–19, 6–56
- Instance
 - attribute value 3–25
 - browser attributes 1–49
 - description 8–14
 - name 8–14
 - objects 3–23
 - property list function 5–54
- Instance ID 8–59
 - retrieving 8–57
- Instance level
 - attribute value 8–33
 - setting properties 3–18
- Instances, contained 8–59
- Integrating applications 6–31
- isICFRunning function 6–39
- Item link 3–46

J

- Joins, customization of SmartObject table
 - 8–44

K

- Keys
 - cache object record 8–52
 - SmartObject table 8–44
 - unique 8–5

L

- launchContainer procedure 6–5, 6–28
- launchExternalProcedure 6–37
- launchFolderWindow procedure 5–24
- launchProcedure 6–28
- Layout table 8–37
 - structure 8–38
- layout_position field 8–14
- layout_supported field 8–7
- Level, constant 3–24
- Link
 - editor 3–12
 - modifying activation 3–13
 - repository 8–51
- Linked object values 3–33
- LinkHandles function 5–31
- LinkProperty function 5–31
- Links
 - addLink 5–27
 - container 5–29
 - data 3–14
 - groupAssign 3–15
 - item 3–46
 - management 5–27
 - object 3–3
 - properties 5–32
 - SmartObject 5–27
 - source/target 8–35
 - support methods 5–29
- Localization manager 6–72
- Logic
 - business 1–35
 - client-side API 1–50
 - defined in super procedures 1–11
- Logical service maintenance window 6–7

- LogicalObjectName 5–4

- Lookup type 3–26

- lookup_value field 8–22

M

- Maintenance windows, logical/physical service 6–8

Manager

- access from viewers 7–38
- accessing from SDO 7–34
- API 6–3
- architecture 6–4, 6–14
- client 6–16
- client/server 6–3
- code 6–15
 - server 6–21
- creating field edit 7–17
- defining 7–20
- handles 6–14
- new 7–5
- procedure name 7–12
- Progress Dynamics 7–1
- registering 7–31
- server 6–16
- structuring on server 6–16
- templates 7–3
- testing 7–40

Manager type

- control window 6–6
- creating 7–8
- defining 6–6
- editing 6–41
- maintenance 7–9

- Managers, overview 6–3

- Manual, organization of xv

- Mapping customization 8–44

Master

- attribute value 3–24
- browser attributes 1–49
- objects 3–23

- Master level, property setting 3–17
- Menu items, disabling and hiding 3–28
- Message
 - control 7–37
 - table 6–63
- Messages, defining and using 6–63
- Methods
 - modifyNewRecord 5–49
 - startup/shutdown 5–24
- MinHeight attribute 3–10
- MinWidth attribute 3–11
- ModifyDisabledActions procedure 3–31
- Modifying objects 8–41
- ModifyList property 5–56
- ModifyNewRecord method 5–49
- multiTranslation API call 6–76

N

- Naming conventions 1–20
- NewRecord property 3–49
- NO-PROXY setting 7–3

O

- Object
 - class 8–6
 - description 8–10
 - filename 8–10
 - hierarchy 8–6
- Object classes 2–1
- Object diagram 8–6
- Object Generator, modifying 1–14

- Object IDs 8–4
- Object instance 8–33
 - level 8–28
 - table 8–13
- Object record 8–49
 - key 8–50
- Object type
 - code 8–7
 - control 3–23
 - defining 2–12
 - description 8–7
 - level 8–26
 - maintenance window 3–27
 - master level 8–26
 - sub-class 2–12
 - table 8–6
- ObjectEnabled property 5–4
- ObjectMode property 3–50
- Objects 8–1
 - adding to test window 3–5
 - DataField 8–2
 - dynamic generation 8–4
 - handles 5–7
 - identification 3–39
 - in repository 8–5
 - instantiated with prepareInstance 8–59
 - linked 3–33
 - modifying 8–41
 - names 5–3
 - order maintenance 3–8
 - relationships with super procedures 1–26
 - static/dynamic 1–9, 8–7
 - static/logical 8–9
 - types 5–3
 - widget 8–2
- Objects, changing type 2–23
- Objects, extending 2–1
- Objects, heirarchy 2–2
- Order maintenance objects 3–8
- Overrides, parameters 3–33

P

- Page object table 8–40
- Page table 8–39
- Pages
 - defined 8–39
 - enable/disable 8–39
- Paging
 - methods/properties 5–51, 5–53
- Palette, adding to 2–26
- Partitioning, internal/external 6–20
- pcContainerMode 6–30
- pcInstanceAttributes 6–30
- pcLogicalName 6–29
- pcObjectFileName 6–29
- pcPhysicalName 6–29
- pcProcedureType 6–31
- pcRunAttribute 6–30
- Performance issues 1–4
- Persistent procedures 6–28, 8–34
- phObjectProcedure 6–30
- phParentProcedure 6–30
- phParentWindow 6–30
- phProcedureHandle 6–31
- Physical object name 5–3
- Physical service maintenance window 6–8
- Physical types 6–43
- physical_smartobject_obj field 8–13
- PLIP 1–35, 1–41, 8–9
- plOnceOnly 6–29
- Popup menu
 - creating 6–60
 - defining event procedures 6–62
 - disabling 6–49
 - menu drop 6–71
 - translating labels 6–73
- Prepare query function 5–22
- prepareInstance function 8–59
- Procedure handle 5–6
- Procedures
 - addLink 5–27
 - adm-create-object 5–14
 - askQuestion 6–65
 - assignPageProperty 5–51
 - cacheFieldEdits 7–21
 - clearClientCache 7–28
 - collectChanges 5–50
 - constructObject 5–15
 - dataAvailable 3–34, 5–21
 - deletePage 5–52
 - destroyObject 5–24
 - disableFields 3–35, 5–21
 - displayFields 5–23
 - displayObject 5–20
 - enableFields 3–35, 5–21, 7–38
 - enableObject/disableObject 5–19
 - eventMenuItemChoose 6–63
 - exitObject 5–24
 - fetch 5–23
 - fieldEditData 7–30
 - fieldEditValidate 7–35
 - getEntityDescription 6–21
 - hidePage 5–52
 - icfstart.p 7–16
 - initializeObject 3–30, 3–39, 5–19, 6–56, 7–34
 - initPages 5–52
 - launchContainer 6–28
 - launchProcedure 6–28
 - modifyDisabledActions 3–31
 - modifyListProperty 5–56
 - plipSetup 7–16
 - removeLink 5–30
 - reset 3–37

- rowDisplay 1–47
- rowObjectValidate 7–40
- selectPage 5–53
- sendRows 5–23
- setProfileData 6–68
- showMessages 6–69
- viewObject/hideObject 5–20
- viewPage 5–52

Product module object 8–8

Profile

- changing 6–47
- creating types and codes 6–48
- data 6–57
- defining types 6–9
- manager 6–47, 6–57

Progress 4GL 8–34

Progress Dynamics Architecture 8–4

Properties

- action rules 3–48
- browser, new 6–49
- data visualization 3–50
- DataModified 3–49
- defining 6–26
- DisabledActions 3–28
- disableOnInit 5–25
- editable 3–49
- FolderWindowToLaunch 5–26
- foreign fields 3–8
- ForeignFields 3–8
- Hidden Actions 3–28
- managing 5–54
- NewRecord 3–49
- objectInitialized 5–25
- ObjectMode 3–50
- OpenOnInit 5–25
- page 5–53
- query position 3–44
- QueryPosition 3–48
- RecordState 3–48
- RowObjectState 3–48
- setting at instance level 3–18
- setting at master level 3–17
- setUser 5–55
- tableIOType 3–29
- visual 3–17

Property

- action rule 3–47
- management 6–25
- setting values 6–26
- values, client/server 6–27

Property include files 8–17

PropertyType function 5–55

ProTools 1–27

Prototype generator (ProtoGen) 1–27

PUBLISH statement 1–25

- parameters 3–33
- redirect 3–41

Q

Queries

- custom 5–33
- filter 5–33
- filtering/sorting 5–45
- handle 5–7
- management 5–33
- refresh 5–43

Query

- properties 5–51
- reposition 5–39

Query selection, assign/remove 5–36

QueryObject property 5–4

QueryPosition property 3–44, 3–48

QueryRowObject property 1–47, 5–8

R

Record identifiers 8–50, 8–52, 8–59

RecordState property 3–48

Referential integrity manager 6–13

Refreshing database query 5–43

registerCacheItem function 4–15

Remove links procedures 5–30

Rendering procedure 2–20

Rendering, thin 2–21

Reposition database query 5–39

Repository

- adding tables 7–17

- database 6–23

- design manager 6–12

- maintenance tool 1–16, 2–13

- manager 6–12

- Object IDs 8–4

- object maintenance tool 3–17

- standard tables 7–18

- storing translated text 6–73

Repository manager 8–45, 8–48

- API 8–53

Request manager 6–13

required_db_list field 8–11

Reset procedures 3–37

Resize

- horizontal 3–10, 5–17

- modifying attributes 3–10

- vertical 3–10, 5–17

Result codes 8–42

Retrieving object information 8–57

Row object

- properties 5–51

- query filtering/sorting 5–45

- table properties 5–51

rowDisplay procedure 1–47

RowIdent 5–41

RowObjectState property 3–48

Rules, action defined for menu/toolbar 3–42

RUN SUPER statement, when not to use
3–41

run_when option 8–12

S

Schema triggers 6–13

SCM integration 2–25

Scoping variables 3–40

Section editor 7–23

Security 8–9

- adding check 6–80

- disable/enable 8–7

- restrictions 6–80

Security manager 6–11, 6–80

security_smartobject_obj field 8–13

Select page procedure 5–53

Server

- calls to manager 7–24

- code 7–5

Service

- connecting 6–46

- disconnecting 6–46

- interacting 6–46

- registering 6–45

Service type manager

- registering 6–45

Session

- defining types 6–7

- manager 6–5, 6–23

- type 7–10

 - properties 6–8

 - type and parameter 6–42

set function 8–20

Set query functions 5–34

- setOpen query 5–34, 5–35
- setProfileData procedure 6–68
- setPropertyList function 6–26
- Setting properties 3–19
- setUser property 5–55
- Shared cache 4–5
- showMessages procedure 6–69
- Shutdown
 - methods 5–24
 - unsaved changes message 8–10
- Signature function 5–56
- SmartDataObjects 8–50
 - accessing Manager 7–34
- SmartLink tables 8–34, 8–36
- SmartLink type table 8–35
- SmartLinks 3–3
 - supported 8–35
- SmartObject table 8–8, 8–15
 - joining customizations 8–44
- SmartObjects 8–7
 - initialization 5–19
 - super procedures 1–6, 1–7
- Startup
 - session type 7–15
 - Dynamics 7–11
 - methods 5–24
- Statements
 - PUBLISH redirect 3–41
 - RUN SUPER, do not use 3–41
- Static handle 6–40
- Static object 8–7
- static_object field 8–10

- Storing attributes 8–59
- Sub-class object type 2–12
- Subclassing 2–1
- Subclassing in the middle 2–8, 2–9
- subclassing off bottom 2–8
- SUBSCRIBE statement 1–25
- Super procedures
 - client code 1–5
 - creating 1–12
 - custom 1–6, 3–30
 - registering 3–30
 - defined 2–11
 - defining 1–8
 - principles 1–23, 1–24, 1–25, 1–29
 - relationships with objects 1–26
 - scoping variables 3–40
 - SmartObjects 1–7
- Super procedures, instantiation order 2–14
- Support functions 1–48, 6–12
- Synching instances 4–12
- System support 6–12
- system_owned field 8–10, 8–13

T

- TableIOType property 3–29
- Tables
 - gsc_object_type 8–7
 - object type 8–6
 - SmartObject 8–8
- Target page function 5–53
- TARGET-PROCEDURE 1–32, 1–33
- template_smartobject field 8–10
- Templates Manager 7–3

Temp-table 6–3
 cache_object page 8–50
 cache_ObjectLink 8–51
 cient cache 8–49
 object instances 8–51
 property values 6–25
 UI events 8–51

Text translation 6–72

Thin rendering 2–21

tokenSecurityCheck 6–82

Toolbar adding to test window 3–4

Tools customization 8–47

translatePhrase function 6–74

Translation
 control 6–73
 text 6–72
 text strings 6–73

Types physical 6–43

U

User interface
 events 1–17
 naming conventions 1–20
 supporting procedures 1–21
 testing 1–22
 viewer 1–16
 generation by web browser 6–13

 manager 6–13
 web-based 6–5

User interface events 1–13

User profile types and codes 6–48

V

Value check 3–38

View page procedure 5–52

Viewer
 customer maintenance 3–7
 property sheet 3–20

ViewObject procedure 5–20

Visual properties, modifying 3–17

W

Web browser 6–5

WHERE clause 5–33
 customize with add query 5–35

Widget 1–10

Widget functions 1–48

Widget objects 8–2

Writing code, basic principles 1–35

