

Expert Series

Progress Dynamics. Developer's Guide

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, 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: 4477
Item Number: 103613;V2.1B

Acknowledgements

Much of the material in this book is derived from functional specifications for the various product areas it covers, from the previous edition of this book, or from earlier framework documentation. Many developers and documenters created the original material, without which this book would not have been possible. This edition was co-authored by John Sadd and Hilton-James Barnes.

Contents

Preface	xvii
Purpose	xvii
Audience	xvii
Organization of this manual	xviii
Typographical conventions	xx
Syntax notation	xxi
Progress messages	xxv
1. Overview	1-1
1.1 Progress Dynamics framework	1-2
1.1.1 Progress Dynamics and the ADM	1-2
1.1.2 Benefits of the Repository	1-3
1.1.3 Client support	1-4
1.2 Prescriptive development approach	1-5
1.2.1 Standard components	1-6
1.2.2 Standard code structure	1-7
1.2.3 Goals	1-8
1.3 Summary	1-8
2. Database Design Principles in Progress Dynamics	2-1
2.1 Schema naming conventions	2-3
2.1.1 Field naming conventions	2-3
2.1.2 Table naming conventions	2-4
2.1.3 Dynamic object naming conventions	2-6
2.1.4 Naming length limits	2-6

2.2	Normalization and table design	2-7
2.2.1	Avoid array fields.	2-8
2.2.2	Reduce field numbers	2-9
2.2.3	Avoid constraints that lead to larger transaction scopes.	2-10
2.2.4	Consider distributed database access	2-10
2.3	Indexing guidelines	2-11
2.4	Validation and other schema information	2-12
2.5	Object IDs and site numbers in Progress Dynamics	2-13
2.5.1	Object IDs	2-13
2.5.2	Site numbers	2-15
2.6	Using <i>ERwin</i> with Progress Dynamics	2-16
2.7	Conclusion	2-18
3.	Progress Dynamics Integration with the AppBuilder	3-1
3.1	Starting the AppBuilder	3-3
3.2	Progress Dynamics AppBuilder UI enhancements	3-3
3.2.1	Main window enhancements.	3-4
3.2.2	File menu enhancements	3-5
3.2.3	Build menu	3-9
3.2.4	Compile menu enhancements	3-11
3.2.5	Tools menu	3-11
3.3	Progress Dynamics Administration window	3-12
3.4	Progress Dynamics Development window	3-14
3.4.1	Administration and Development File menu	3-16
3.5	Design windows for Repository objects	3-19
3.5.1	Options disabled for dynamic object design windows.	3-19
3.5.2	Repository object types the AppBuilder edits	3-19
3.5.3	Object names and filename extensions	3-20
3.6	Creating new Repository objects	3-21
3.6.1	Creating a single Repository application object	3-21
3.6.2	Creating many application objects at once	3-23
3.7	Opening objects for editing	3-23
3.7.1	Opening a Repository object.	3-23
3.7.2	Opening files not in the Repository	3-24
3.8	Saving an object to the Repository	3-24
3.9	Adding a file to the Repository	3-26
3.10	Saving a dynamic object as static	3-27
3.10.1	Replacing a dynamic object with a static object	3-28
3.10.2	Handling triggers	3-28
3.11	Editing properties for Repository objects	3-28
3.11.1	Property sheet example	3-29
3.12	Running objects	3-29
3.12.1	Running objects with the Dynamic Launcher	3-30
3.13	Closing Repository objects	3-31

3.14	Object palette and object template information stored in Repository . . .	3–31
3.14.1	Creating and modifying template objects	3–32
3.14.2	Creating and modifying Palette objects	3–34
3.14.3	New and changed attributes	3–36
3.15	Summary	3–41
4.	Preparing to Build Application Objects	4–1
4.1	Obtaining and assigning site numbers	4–2
4.1.1	Obtaining a site number	4–2
4.1.2	Assigning a site number to your database	4–7
4.2	Defining products and product modules	4–8
4.2.1	Assigning product and product module names	4–8
4.2.2	Assigning login companies to products	4–11
4.2.3	Creating product modules	4–11
4.3	Creating Repository data for tables and fields	4–12
4.3.1	Entity import	4–13
4.3.2	Entity record fields	4–16
4.3.3	Entity maintenance	4–18
4.3.4	Entity mnemonic	4–21
4.3.5	Maintaining Entity Display fields.	4–24
5.	Using the Object Generator.	5–1
5.1	Introduction – field, field container, and window objects	5–2
5.1.1	Generating field objects	5–3
5.1.2	Generating field container objects	5–3
5.1.3	Generating window objects	5–3
5.2	Field container object basics	5–3
5.2.1	SmartDataObject basics	5–3
5.2.2	SmartDataBrowser basics	5–8
5.2.3	SmartDataViewer basics.	5–9
5.3	Using the Object Generator	5–10
5.3.1	Object preferences.	5–12
5.3.2	Objects page	5–13
5.3.3	SCM page	5–13
5.3.4	Data Objects page	5–13
5.3.5	The DataFields page	5–21
5.3.6	Browsers page	5–23
5.3.7	Viewers page	5–24
5.3.8	Logging page	5–26
5.3.9	Generating objects.	5–26

6.	Using the AppBuilder in Progress Dynamics.....	6-1
6.1	Creating and editing individual objects	6-2
6.1.1	Creating and editing SDOs	6-2
6.1.2	Creating and editing browsers	6-5
6.1.3	Creating and editing viewers.....	6-7
6.1.4	Changing an object's product module.....	6-8
6.2	Building viewers	6-9
6.3	The Dynamic property sheet	6-13
6.3.1	User interface	6-13
6.3.2	Description fields.....	6-14
6.3.3	Tab folders	6-15
6.4	Migrating static objects to Progress Dynamics	6-17
6.4.1	Individual object migration.....	6-17
6.4.2	Batch object migration.....	6-18
7.	Building Progress Dynamics Lookups and Combos.....	7-1
7.1	Adding dynamic combos and lookups to viewers	7-3
7.1.1	Defining and using dynamic combos	7-4
7.1.2	Saving the combo	7-13
7.1.3	Runtime example	7-13
7.1.4	Defining and using dynamic lookups.....	7-13
7.1.5	Using the SmartDataField Maintenance tool.....	7-19
7.1.6	Running a window with a lookup.....	7-22
7.2	Viewer tabbing options	7-25
7.3	Combo and lookup performance and caching considerations	7-27
7.4	Subclassing dynamic combos and lookups	7-28
8.	Using the Progress Dynamics Container Builder	8-1
8.1	Introduction	8-2
8.2	Relative positioning in the containers	8-3
8.3	Launching the Container Builder	8-5
8.3.1	Launching the Container Builder from the Build menu	8-5
8.3.2	Creating containers using the AppBuilder.....	8-8
8.4	Using the Container Builder	8-10
8.4.1	Creating a new container	8-10
8.4.2	Assigning objects to the container	8-11
8.5	Advanced container features	8-17
8.5.1	Container properties	8-17
8.5.2	Page maintenance	8-18
8.5.3	Links maintenance	8-19
8.5.4	Object menu structure	8-26
8.5.5	Object Initialization sequencer	8-26
8.5.6	Run	8-27

8.6	Standard toolbar objects	8-27
8.6.1	FolderPageTop toolbar	8-27
8.6.2	ObjcTop toolbar	8-28
8.6.3	FolderTop toolbar	8-29
8.6.4	Browser toolbar	8-29
8.6.5	BrowseToolbarNoUpdate toolbar	8-31
8.6.6	FolderTopNoSDO toolbar	8-31
8.6.7	DynToolbar	8-31
8.6.8	Other toolbars	8-31
8.7	Preferences	8-32
8.8	Customization option for dynamic viewers and browsers	8-33
8.8.1	Customization priority	8-34
8.8.2	Synchronization	8-35
8.8.3	Adding and removing widgets	8-35
8.8.4	Supported attributes	8-35
8.9	Dynamic SmartFrames	8-36
9.	Building Progress Dynamics TreeView Windows.....	9-1
9.1	Introduction	9-2
9.1.1	Components of the dynamic TreeView window	9-3
9.2	Building a TreeView window	9-4
9.2.1	Defining TreeView nodes	9-5
9.2.2	Defining the TreeView window	9-11
9.3	Setting up structured nodes	9-13
9.4	Using an extract program for a node	9-17
9.5	Creating a filter viewer for a TreeView window	9-21
9.6	Use Rows To Batch with large numbers of nodes	9-26
9.7	Running the TreeView window	9-26
9.8	Building a menu structure TreeView window	9-27
9.9	Summary	9-30
10.	Building Basic Business Logic in a Progress Dynamics Application	10-1
10.1	Writing distributed applications	10-3
10.1.1	Client access to the database	10-4
10.1.2	Record locking and transaction scoping	10-5
10.1.3	Minimizing AppServer calls from the client	10-6
10.2	The Progress Dynamics SmartDataObject	10-6
10.2.1	SmartDataObject basics	10-7
10.2.2	Setting instance properties in SDOs	10-8
10.2.3	Data management in the SDO	10-13
10.2.4	Determining the proper batch size	10-18
10.3	Strategies for SmartDataObject query definition	10-21

10.4	Standard validation procedures for SDOs	10-26
10.4.1	Client-side data validation	10-26
10.4.2	Server-side validation	10-29
10.4.3	New Progress Dynamics validation procedures	10-32
10.5	The SDO logic procedure	10-36
10.5.1	Calculated field support.	10-40
10.6	Message handling in Progress Dynamics	10-42
10.6.1	Retrieving and formatting error messages with aferrortxt.i	10-44
10.6.2	Error message checking include file (checkerr.i)	10-47
10.6.3	The Progress Dynamics message dialog box.	10-51
10.7	Summary	10-54
11.	Building Advanced Business Logic in a Progress Dynamics Application . . .	11-1
11.1	Building SmartDataObjects against temp-tables	11-2
11.1.1	Temp-table basics.	11-2
11.1.2	Populating the SDO's temp-table	11-7
11.1.3	Defining an SDO with a temp-table like a database table.	11-10
11.2	Database triggers and Progress Dynamics	11-12
11.3	Running business logic procedures in Progress Dynamics	11-13
11.3.1	Creating PLIPs	11-17
11.3.2	Transaction Scoping within procedures	11-20
11.3.3	Using PLIPs versus SDO logic procedures.	11-22
11.4	SBO overview	11-24
11.4.1	Container Builder support	11-24
11.4.2	Viewers	11-25
11.4.3	Data logic procedure	11-25
11.4.4	API	11-26
11.5	Building and using SmartBusinessObjects	11-26
11.5.1	Guidelines for using SBOs	11-27
11.5.2	SBO files and properties	11-29
11.5.3	Data relationships in the SBO	11-31
11.5.4	Brokering SDO data in an SBO	11-32
11.5.5	Data management in the SBO	11-34
11.5.6	Updates through the SBO.	11-37
11.5.7	Other SBO issues	11-40
11.6	Example: creating an SBO	11-42
11.6.1	Creating a dynamic SBO.	11-42
11.6.2	Creating a static SBO	11-44
11.7	Defining business logic for SBOs	11-50
11.8	Conclusion	11-56

12.	Using the Toolbar and Menu Designer	12-1
12.1	Overview of the Toolbar and Menu Designer	12-2
12.1.1	Defining menu and toolbar items	12-5
12.1.2	Defining categories	12-6
12.1.3	Creating a category	12-7
12.1.4	Defining items	12-8
12.1.5	Creating items	12-11
12.1.6	Copying nodes	12-18
12.1.7	Editing nodes	12-18
12.1.8	Deleting nodes	12-19
12.2	Defining toolbar bands	12-19
12.2.1	Adding items to bands	12-21
12.2.2	Displaying band items and adding items on the fly	12-21
12.2.3	Band object associations	12-21
12.2.4	Creating bands	12-26
12.3	Defining SmartToolbar objects	12-29
12.3.1	SmartToolbar bands	12-30
12.3.2	Menu bar options	12-30
12.3.3	Defining the example SmartToolbar object	12-31
12.4	Menu translations	12-32
12.4.1	Adding a toolbar to an application window	12-34
12.4.2	Adding your toolbar to a dynamic window	12-34
12.4.3	Defining a custom super procedure	12-36
12.4.4	Attaching a custom super procedure to all windows	12-39
12.4.5	Editing toolbar properties	12-39
13.	Defining Progress Dynamics Application Security	13-1
13.1	Introduction	13-2
13.1.1	Security model	13-2
13.1.2	Security user types	13-3
13.1.3	Security structures	13-4
13.1.4	Security allocations	13-5
13.1.5	Resolving security	13-6
13.2	Using security documentation	13-6
13.3	Using the Security Control tool	13-8
13.4	Menu security	13-11
13.4.1	Creating menu security allocations	13-12
13.5	Menu item security	13-16
13.5.1	Creating menu item security allocations	13-16
13.6	Container security	13-18
13.6.1	Creating container security allocations	13-19
13.7	Action security	13-20
13.7.1	Creating action security structures	13-20
13.7.2	Restricting access to folder tabs	13-22

13.8	Field security	13-24
13.8.1	Creating field security structures	13-25
13.8.2	Creating field security allocations	13-27
13.9	Data range security	13-28
13.9.1	Creating data range security structures	13-29
13.9.2	Creating data range security allocations	13-31
13.9.3	Programming with data range security	13-33
13.10	Data security	13-36
13.10.1	Creating data security allocations	13-37
13.11	Login company security	13-39
13.11.1	Creating and maintaining login company security structures ..	13-40
13.11.2	Defining security allocations for login companies	13-40
13.12	Searching with the Security Enquiry tool	13-42
Index	Index-1	

Figures

Figure 1–1:	Progress Dynamics run-time architecture	1–5
Figure 2–1:	ERwin Attribute Editor showing object ID definitions	2–16
Figure 2–2:	ERwin Table Trigger Editor	2–17
Figure 2–3:	Expanded Template Code editor	2–17
Figure 2–4:	4GL code generated by ERwin macros	2–18
Figure 3–1:	Progress Dynamics enabled AppBuilder main window	3–4
Figure 3–2:	AppBuilder New dialog box	3–6
Figure 3–3:	Open Object dialog box	3–7
Figure 3–4:	Save As dialog box	3–8
Figure 3–5:	Save As Dynamic Object dialog box	3–8
Figure 3–6:	Register in Repository dialog box	3–9
Figure 3–7:	Progress Dynamics AppBuilder Compile menu	3–11
Figure 3–8:	Progress Dynamics Administration window	3–12
Figure 3–9:	Progress Dynamics Development window	3–14
Figure 3–10:	Suspended User dialog box	3–16
Figure 3–11:	Progress Dynamics Preferences dialog box	3–17
Figure 3–12:	Static SDO Dynamic Properties window	3–29
Figure 4–1:	Progress Dynamics Site Number Allocation web site	4–3
Figure 5–1:	Interaction between the server and client proxy DynSDO	5–5
Figure 5–2:	Business logic procedure	5–7
Figure 5–3:	Object Generator with multiple tables selected	5–10
Figure 5–4:	The Data Objects page of the Object Generator	5–13
Figure 5–5:	Object Generator showing existing SDOs	5–16
Figure 5–6:	DataFields page	5–21
Figure 5–7:	Browsers page fields	5–23
Figure 5–8:	Viewers page fields	5–25
Figure 5–9:	Logging page	5–26
Figure 6–1:	Dynamic property sheet	6–14
Figure 6–2:	Static SmartObject to Dynamic Object Migration Utility window	6–18
Figure 6–3:	Advanced Migration Settings window	6–19
Figure 7–1:	Example of run time Customer Update window	7–13
Figure 7–2:	SmartDataViewer example	7–19
Figure 7–3:	Order Update window	7–22
Figure 7–4:	Customer Lookup	7–23
Figure 7–5:	Customer Lookup window - Filter tab	7–24
Figure 8–1:	Container Builder - open	8–5
Figure 8–2:	Container Builder - new mode	8–6
Figure 8–3:	Container Builder - existing container	8–7
Figure 8–4:	New object window	8–8
Figure 8–5:	Object design widget	8–9
Figure 8–6:	Container Builder launched from the AppBuilder	8–9
Figure 8–7:	Object instances details	8–11
Figure 8–8:	Layout grid	8–12

Figure 8-9:	Advanced layout grid features	8-13
Figure 8-10:	Object type selectors	8-14
Figure 8-11:	Add Quick-Link tool	8-15
Figure 8-12:	Add Quick-Link tool disabled	8-15
Figure 8-13:	Add Quick-Link tool enabled	8-15
Figure 8-14:	Dynamic Properties window	8-17
Figure 8-15:	Page Maintenance window	8-18
Figure 8-16:	Insert / replace warning	8-19
Figure 8-17:	Links Maintenance window	8-20
Figure 8-18:	Object Locator window	8-24
Figure 8-19:	Foreign Fields Mapping dialog	8-25
Figure 8-20:	Object Menu Structures window	8-26
Figure 8-21:	Object Initialization sequencer	8-26
Figure 8-22:	FolderPageTop toolbar	8-27
Figure 8-23:	File Menu that duplicates toolbar functions	8-28
Figure 8-24:	ObjcTop toolbar	8-28
Figure 8-25:	FolderTop toolbar	8-29
Figure 8-26:	Browser toolbar	8-29
Figure 8-27:	Filter window	8-29
Figure 8-28:	Print Preview Example	8-30
Figure 8-29:	Audit Control	8-31
Figure 8-30:	Container Builder Preferences - General tab	8-32
Figure 8-31:	Container Builder Preferences - Grid tab	8-32
Figure 9-1:	Folder toolbar linked to current displayed logical object	9-3
Figure 9-2:	Completed example TreeView	9-5
Figure 9-3:	Tree Node Maintenance frame	9-7
Figure 9-4:	Examples of TreeView styles	9-11
Figure 9-5:	Repository gsm_node table	9-14
Figure 9-6:	Tree Node Control window Details tab	9-15
Figure 9-7:	Tree Node Control window Structure tab.	9-16
Figure 9-8:	New dialog box	9-18
Figure 9-9:	loadData procedure	9-18
Figure 9-10:	Example SmartDataViewer	9-22
Figure 10-1:	Reading the first batch of rows into the SDO	10-13
Figure 10-2:	Reading the second batch of rows into the SDO	10-14
Figure 10-3:	Modifying customer 5 In the client SDO	10-15
Figure 10-4:	Adding row 9 and deleting row 6 after updating row 5	10-16
Figure 10-5:	SmartDataObject on client and server	10-38
Figure 10-6:	SmartDataObject with business logic procedure	10-39
Figure 10-7:	Dynamic SDO with business logic procedure	10-40
Figure 10-8:	Standard Message dialog box: Message Summary tab	10-51
Figure 10-9:	Standard Message dialog box: Message Detail tab	10-51
Figure 10-10:	Standard Message dialog box: System Information tab	10-52
Figure 10-11:	Sample Message dialog box: Message Summary tab	10-53
Figure 10-12:	Unformatted message display	10-54

Figure 11-1:	Test Window for Temp-Table	11-8
Figure 11-2:	Updated Test Window for Temp-Table	11-8
Figure 11-3:	All values confirmation message box	11-12
Figure 11-4:	Standard objectDescription procedure	11-19
Figure 11-5:	Example data link hierarchy	11-32
Figure 11-6:	SmartToolbar properties dialog box	11-33
Figure 11-7:	Data management example	11-35
Figure 11-8:	Example window – Pages 0 and 1	11-48
Figure 11-9:	Example window: OrderLines	11-49
Figure 11-10:	ADM2 message	11-51
Figure 11-11:	ADM2 error message	11-52
Figure 12-1:	Progress Dynamics Toolbar and Menu Designer	12-2
Figure 12-2:	Search Nodes dialog box	12-3
Figure 12-3:	Search Nodes dialog box: Items tab folder	12-4
Figure 12-4:	Search Nodes dialog box – Bands tab folder	12-5
Figure 12-5:	Toolbar and Menu Designer - Category tab folder	12-6
Figure 12-6:	Toolbar and Menu Designer - Item tab folder	12-8
Figure 12-7:	Example toolbar	12-11
Figure 12-8:	File menu	12-12
Figure 12-9:	Edit menu	12-12
Figure 12-10:	Search menu	12-13
Figure 12-11:	Modules menu	12-13
Figure 12-12:	Progress Dynamics Administration window	12-13
Figure 12-13:	Defining toolbar bands	12-20
Figure 12-14:	Toolbar and Menu Designer – Band Object tab	12-22
Figure 12-15:	File band	12-23
Figure 12-16:	Merging bands: results	12-23
Figure 12-17:	Toolbar and Menu Designer – Band Item tab	12-24
Figure 12-18:	Progress Dynamics Administration window example	12-25
Figure 12-19:	Toolbar and Menu Designer – SmartToolbar Tab	12-29
Figure 12-20:	Menu Item Translation main window	12-33
Figure 13-1:	Security Control tool	13-9
Figure 13-2:	Example Customer Selection window	13-34
Figure 13-3:	Login with restricted company	13-42
Figure 13-4:	User Authentication Failure message	13-42

Tables

Table 2–1:	Values for third character in Progress Dynamics table prefixes	2–5
Table 2–2:	ERwin naming length limits	2–7
Table 3–1:	AppBuilder main window Progress Dynamics-specific menu	3–4
Table 3–2:	Progress Dynamics-specific options on the File menu	3–5
Table 3–3:	Build menu options	3–10
Table 3–4:	Compile menu options	3–11
Table 3–5:	Tools menu options	3–12
Table 3–6:	Progress Dynamics Administration window	3–12
Table 3–7:	Development window menus	3–14
Table 3–8:	Administration and Development File menu options	3–16
Table 3–9:	Preferences dialog box options	3–17
Table 3–10:	Options disabled for Progress Dynamics object Repository design . . .	3–19
Table 3–11:	Creating new objects	3–22
Table 3–12:	Dynamic Launcher options	3–30
Table 3–13:	Changed attribute names	3–37
Table 3–14:	Template attributes	3–38
Table 3–15:	Palette attributes	3–39
Table 5–1:	Object Generator preferences	5–12
Table 5–2:	Data Objects page general fields	5–14
Table 5–3:	Settings for data objects	5–14
Table 5–4:	Settings for DataLogic procedure	5–18
Table 5–5:	Browsers page fields	5–24
Table 5–6:	Viewers page fields	5–25
Table 6–1:	Target field options	6–16
Table 6–2:	Filtering options	6–17
Table 8–1:	FolderPageTop toolbar links	8–27
Table 10–1:	Supported arguments	10–44
Table 10–2:	Named arguments for checkerr.i	10–47
Table 10–3:	Additional arguments for checkerr.i	10–48
Table 10–4:	Question arguments	10–49
Table 10–5:	NO-RETURN variables	10–50
Table 11–1:	launch.i named arguments	11–14
Table 11–2:	Variable values available	11–16
Table 12–1:	Toolbar Designer menu bar	12–30
Table 13–1:	Security Control tool documentation	13–7
Table 13–2:	Security Control tool parts	13–9
Table 13–3:	User names and login company security	13–11
Table 13–4:	Data security by security model	13–39

Preface

Purpose

This guide provides comprehensive information on how to develop applications using the Progress Dynamics® framework.

This guide is written by developers experienced in the use of Progress Dynamics, so that application designers and developers have access to the level of serious technical material they need to design, build, and deploy enterprise applications using the whole range of Progress® technologies. The principal author of this Developer's Guide is John Sadd, Progress Engineering Fellow, ADM Architect, and author of a number of white papers on the use of the ADM2.

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 guide is organized in the following manner:

[Chapter 1, “Overview”](#)

Describes the Progress Dynamics framework, how it relates to the Progress Application Development Model (ADM), and the Progress Dynamics prescriptive approach to development.

[Chapter 2, “Database Design Principles in Progress Dynamics”](#)

Describes some basic principles of database design that are important to understand to use the Progress Dynamics framework successfully. Some of these are fairly universal, but some are specific aspects of the framework.

[Chapter 3, “Progress Dynamics Integration with the AppBuilder”](#)

Describes how Progress Dynamics menus and tools are fully integrated into the Progress AppBuilder interface.

[Chapter 4, “Preparing to Build Application Objects”](#)

Describes the use of the entity import and maintenance tools. The Repository includes tables where information is stored about each table (or “entity”) in the application database (and for that matter, in the Repository database itself). This information assists the framework in generating default objects, understanding the keys to be used to identify database records, and other operations. If possible, you should set up the entity tables for your application before you generate application components.

[Chapter 5, “Using the Object Generator”](#)

Describes the Object Generator tool, which creates a SmartDataObject™ for each table or combination of tables you specify in your database. These objects handle database queries to retrieve data from the database, send it to the client, and make updates to the data back on the server, applying any validation logic you have defined. In addition, the Object Generator can create a default dynamic Browser and/or a dynamic Viewer for each SDO, so that you can quickly begin to assemble useful table maintenance windows with almost no development work.

[Chapter 6, “Using the AppBuilder in Progress Dynamics,”](#)

Describes how to use the AppBuilder to create and edit objects and focuses on editing object properties using the new Dynamic Properties Sheet. It also shows how objects can be migrated from a static version to Progress Dynamics.

Chapter 7, “Building Progress Dynamics Lookups and Combos”

Describes how to create static (procedural) Progress SmartDataViewers™ to display and update fields from a record using the AppBuilder. You can customize both static Viewers (in the AppBuilder) and dynamic Viewers (using the Progress Dynamics Repository Object Maintenance tool) to adjust positioning and format of fields, to add other visual objects, and in particular to add dynamic Lookups or Combos to a Viewer. These very powerful Progress SmartDataField™ objects allow the application user to select a valid value for a field that is a foreign key for some other table from a list of possible values generated directly from the application database.

Chapter 8, “Using the Progress Dynamics Container Builder”

Describes the tools you can use to build layout templates for windows and pages of tab folders. It also describes how to build dynamic windows from those layouts and how to assemble your individual application objects into complete windows.

Chapter 9, “Building Progress Dynamics TreeView Windows”

Describes how to create Progress Dynamics windows with a TreeView layout. It details how to use property sheets to define nodes of the tree and the overall tree structure. The nodes are represented in an ActiveX-based TreeView object on the left side of the window. The dynamic windows launched from the selection of the nodes appear as frames on the right-hand side. You can use a TreeView to represent either hierarchical data or a menu structure.

Chapter 10, “Building Basic Business Logic in a Progress Dynamics Application”

Reviews principles of writing a distributed application. It also describes validation logic at the level of a single table, and the Progress Dynamics tools for defining and using application messages.

Chapter 11, “Building Advanced Business Logic in a Progress Dynamics Application”

Covers more complex business logic, including how to organize multiple related tables into a larger transaction, and the use of stand-alone business logic procedures that can execute any logic you need to have on the server.

Chapter 12, “Using the Toolbar and Menu Designer”

Describes how to use the powerful Toolbar and Menu Designer that lets you design custom toolbars and menus for use throughout your application. Here you can specify not just the organization and appearance of your toolbars but also the exact action, parameters, and other attributes of each button and menu item.

Chapter 13, “Defining Progress Dynamics Application Security”

Describes how you can apply security to application windows, menu items, folder tabs, database tables, individual records or ranges of data, and so forth. Progress Dynamics provides comprehensive support for defining and maintaining users and their passwords, and for applying security restrictions to your application based on User ID, or based on the Login Company or other organizational entity under which a user logs in.

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**).

Overview

This chapter describes the Progress Dynamics® framework, how it relates to the Progress® Application Development Model (ADM), and how to use best practices to get the most out of Progress Dynamics.

It includes the following sections:

- [Progress Dynamics framework](#)
- [Prescriptive development approach](#)
- [Summary](#)

1.1 Progress Dynamics framework

Progress Dynamics is a comprehensive, repository-based framework for Progress 4GL developers building new distributed applications or migrating existing applications to take advantage of new Progress technologies.

A Progress Dynamics application has many different types of components. Some of these components provide support for the user interface, such as:

- Windows
- Tab Folders
- Menubars and toolbars
- Browsers (display query results and allow record selection)
- Viewers or frames (display or update a single record)

Other components are procedural objects that define the business logic of an application.

Most Progress Dynamics components are data-driven objects, created at run time from data in the Progress Dynamics Object Repository. Progress Dynamics labels these objects as *dynamic*. A Progress Dynamics application can also include procedural components from the Progress 4GL and ADM SmartObjects. Progress Dynamics labels these objects *static*, since they are based on static code files (.p, .w, and so on).

1.1.1 Progress Dynamics and the ADM

Progress Dynamics is partly based on the Version 9 Application Development Model (ADM) and Progress SmartObjects™, which provide a basis for defining and combining standard components. The ADM defines the concepts of:

- Templates for objects, so that many objects of a type can have similar characteristics and behavior
- Properties or attributes for objects, which together define all the settable characteristics of a particular object
- Links that allow objects to combine in standard ways and communicate with each other using named Progress events
- Standard behavior for objects, implemented in Progress 4GL procedures that become “super procedures” of an object, providing levels of functional inheritance

Progress Dynamics extends the Version 9 SmartObjects, while retaining compatibility with Version 9 SmartObject applications. The framework also provides supporting services that are completely independent of the ADM, such as Session Management to coordinate all the parts of a distributed application. It also provides tools that let you define free-form procedural logic, perhaps adapted directly from an existing application, and combine Progress Dynamics components with existing applications.

1.1.2 Benefits of the Repository

The Progress Dynamics Object Repository stores data for several purposes. Because Progress Dynamics represents most application objects as data stored in the Repository, you do not need to create, compile, deploy, and maintain source code for them. A dynamic window can contain a dynamic toolbar instance, a dynamic browser to display multiple records from a query, and a dynamic viewer for displaying and updating fields from a single record. Progress Dynamics represents all these objects as data in Repository tables. Progress Dynamics realizes them at run time with standard 4GL procedures that each know how to create objects of a specific type: window, toolbar, browser, or viewer. Each object type has a single one of these *driver* procedures that instantiate and manage every instance of that type.

Managing an application in this way greatly reduces the amount of Progress 4GL code in the application. This fact results in a smaller application footprint and less r-code to deploy to client workstations running the application. You have greater flexibility in how you use and maintain applications. For example, altering a single driver procedure can change the way an entire aspect of an application looks or behaves without needing to modify or even recompile many existing procedures. Finally, an abstract, repository-based definition of an application supports its deployment to a variety of client platforms.

NOTE: When you develop your application, there are many good reasons to place your code in a new repository and keep it separate from the main Repository, which contains all the framework code. The Progress Dynamics documentation distinguishes the framework Repository from other repositories with a capital “R.”

Cascaded attribute values

Object inheritance, and thus performance, have been improved by streamlining the implementation of cascaded attribute values. One performance improvement is the elimination of the overhead of cascading attribute values when creating objects and object instances. Another performance improvement comes as a result of the reduced volume of data, particularly in deployment.

1.1.3 Client support

Progress Dynamics Version 2 supports 4GL clients, including the standard Progress run-time client and Progress WebClient™. With the WebClient, users can download a fully functional, no-cost Progress client executable over the Internet. They can run the client part of your application from anywhere in the world without the need to distribute or maintain Progress run-time installations or the application itself. Progress Dynamics also supports native Web browser-based access to the same abstract application definition.

This support allows part or all of an application to be rendered in a browser without any Progress run-time client. The application still accesses the same back-end business logic on server machines connected to the database. Most of the client part of the application is data, not procedural code. As a result, other client platforms (such as handheld display devices or platforms not yet anticipated) might be driven from the same data definition, without change to the business logic or basic application definition.

Progress Dynamics runs in a distributed environment, with the visual portion of the application running in either a 4GL client session or some other client type, without a local database connection. Business logic runs on one or more Progress AppServer™ sessions where the Repository database and the application database are located, maximizing the efficiency of database access. Access to the AppServer is always stateless, so that a small pool of AppServer sessions can support a large number of clients.

To support this stateless AppServer access, Progress Dynamics uses Manager procedures to handle various aspects of the application and its environment. Each of these procedures runs as both a client-side manager and a server-side manager. The server-side manager maintains one or more Repository database tables on the server and provides data to client sessions as needed. Data is cached on the client for maximum efficiency and, when necessary, returned to the Repository database on the server. This provides persistent storage for all data relating to the running of the application.

Figure 1–1 illustrates this architecture.

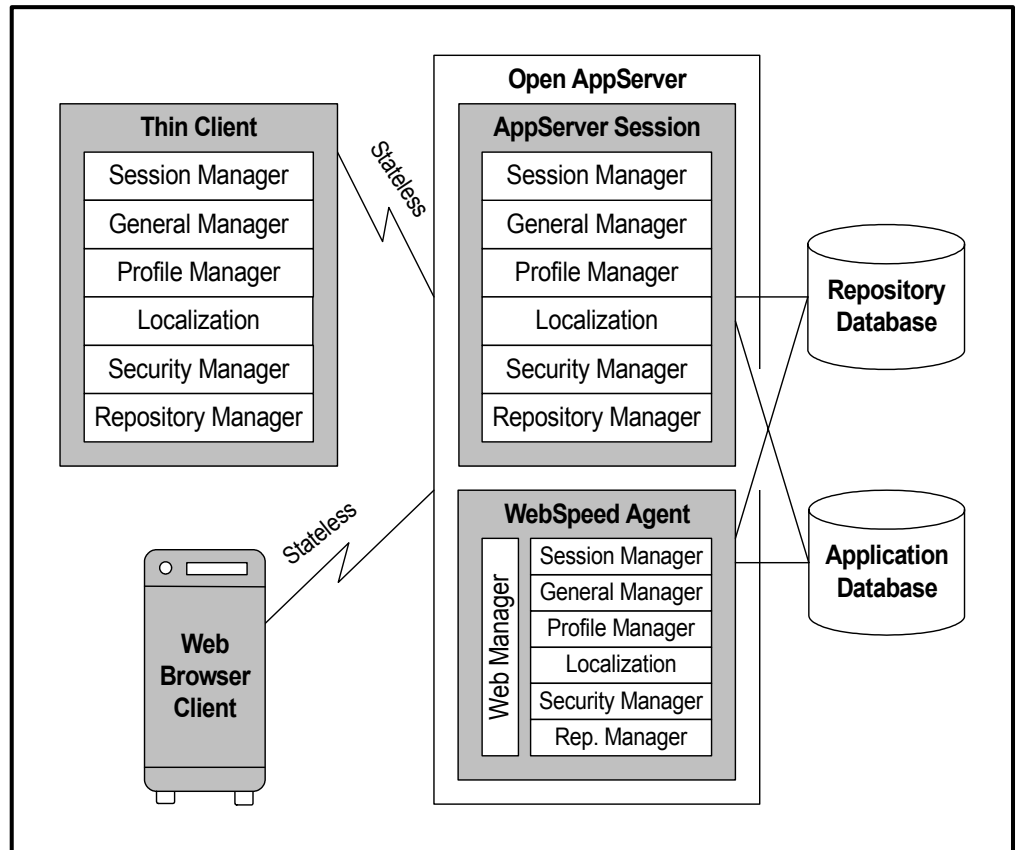


Figure 1–1: Progress Dynamics run-time architecture

1.2 Prescriptive development approach

Progress Dynamics provides an abundance of advantages when you follow its prescriptive approach for developing applications in the Progress 4GL. This prescriptive approach speeds development by allowing you to architect and code your application in a manner that takes advantage of all the out-of-the-box features the framework provides. At the same time, Progress Dynamics provides flexibility where it counts the most. By following the development approach detailed in this guide, you'll be sticking with the prescription and discovering the most efficient ways to customize applications built with the framework.

The framework provides:

- **Standard component types with a wealth of built-in behavior** — You can combine the components in many ways to build a variety of applications. You can extend these components by customizing them or by building new ones with a similar structure. Because these objects are already written for you and because many of the objects are data-driven, you have less code to write while building your application.
- **A standard structure for code that you do have to write** — You have less work deciding how to organize and place your code so it runs when it should, handles messages and error conditions properly, and is easily maintained.

The following sections describe these ideas in more depth.

1.2.1 Standard components

Progress Dynamics strives to provide you with standard predefined solutions to requirements found in most serious applications. These solutions spare you from having to solve the same business and technical problems most applications need to address. These solutions include tools to:

- Define users and security
- Build user interface components in standard styles
- Communicate data and program requests between a stand-alone client and one or more AppServer sessions
- Define and manage the physical and logical configuration of a distributed application
- Define and properly log or display application messages
- Translate a user interface into multiple languages

Every Progress application for an enterprise needs these supporting features. If you start with the Progress Dynamics framework, you can focus immediately on your application's particular business problems without reinventing the supporting features.

These standard features, and in particular the Repository that supports them, make Progress Dynamics-based applications truly "Future Proof™." An application you design, develop, and deploy today can migrate to other platforms, including unanticipated ones, without major rearchitecting.

For example, in the future you might be able to migrate a 4GL WebClient user interface working with the Progress Dynamics Managers and application business logic on the back-end to any browser-based front-end with little extra effort.

1.2.2 Standard code structure

The Progress Dynamics framework provides building blocks that help you identify where application-specific logic should go and how to manage it. For example, on the user interface (UI) side of the application, when you identify common ways you need to extend or override the default UI behavior, the framework provides efficient mechanisms to implement these customizations.

The framework also provides standard events and hooks and guidelines on using them. Progress Dynamics guidelines tell you where and how to:

- Create code to enable and disable fields and other UI objects when specific events occur
- Make special requests of the AppServer
- Add custom logic to the UI to respond to user actions

On the server side of the application, where the bulk of the actual business logic resides, the framework provides several forms that logic can take. The forms are structured so you can understand where to put different parts of the business logic so it:

- Reliably executes at the appropriate times
- Interacts properly with the client side of the application whether the data from the client is valid or not, and whether operations on the server side succeed or fail

The framework is as flexible as possible and should not restrict you from creating distinctive applications to satisfy your particular needs. On the User Interface side, dynamic layouts support a wide variety of objects combinations in a window and relationships between application windows. Dynamic layouts also support standard features such as resizability and access to common useful functions like data filtering and data export. With the Toolbar and Menu Designer, you can create the overall visual organization you need for the actions that tie your application's parts together.

Your application can freely incorporate custom-built static (procedural) objects on the client side, to give the exact look and behavior you need. You can easily combine static objects in your application with fully dynamic screens built out of Repository data. You can create dynamic toolbars and menus for procedural windows. Both dynamic and static windows can contain a mix of dynamic and procedural objects, providing a maximum of flexibility while taking advantage of the data-driven nature of the Progress Dynamics Repository wherever possible.

Another way these standard code structures aid you is through the Progress Dynamics Managers. The Managers, and other components that support for many standard services, are independent procedures with their own APIs. You can customize the Security Manager, Session Manager, Configuration Manager, Connection Manager, and other such components. You can specialize each Manager's services as needed within an application, by adding calls to the API or by overriding or extending the 4GL source code itself.

1.2.3 Goals

The goals of the Progress Dynamics prescription are two-fold:

- The framework provides the best possible head start for Progress application developers looking to bring competitive and complete applications to market quickly. It supplies an understandable structure within which you can create these custom applications.
- The framework provides a common body of work that can contain the contributions of many developers. If developers from all over work together to share commonly useful code and ideas in all areas of application development, then all developers can move forward more quickly and effectively. It provides a far more effective development solution than allowing developers working in isolation to solve the same problems over and over again.

In short, maximizing the standard behavior available to all applications, creating a structure which easily supports writing and organizing custom code, and providing a community for improving the base functionality, are the means for creating a large and successful body of applications.

1.3 Summary

This chapter gives you a rough idea of the scope of the Progress Dynamics framework and the intended scope of this guide.

In addition to the information in this guide, which includes many examples, see [Getting Started with Progress Dynamics](#). It provides a comprehensive, realistic example of a useful application built in the Progress Dynamics framework. It also introduces you to the tools you use to build a Progress Dynamics application. For more advanced development tasks and programming examples, refer to the [Progress Dynamics Programming Handbook](#).

Database Design Principles in Progress Dynamics

With Progress Dynamics™, you do not need to redesign your database before building an application for that database. Redesign may be required for business reasons, but dynamics itself can work with legacy designs if required. Progress Dynamics facilitates rearchitecting existing applications to make them more distributed with n-tier architecture. The goal is creating an application that runs in a graphical distributed environment with a number of different interfaces, while retaining much of your existing business logic. The rearchitected application should also work with existing 4GL reports and other programs that execute on the server. The framework is designed, as much as possible, not to require you to redesign or modify your existing database to support a gradual migration strategy.

However, there are Progress Dynamics design conventions that can provide substantial benefits if you are defining a new database, whether for a brand new application or to modernize an existing database. If you are in a position to make extensions to an existing database, you can make use of these conventions to provide guidance on how to extend it. For example, there are fields you might add to the database tables to make your data more accessible to the framework.

There are design principles that are applicable to most databases. This chapter does not cover all principles of relational database design. Instead, it discusses the aspects of relational design that are especially important to success with Progress Dynamics. Reviewing these principles might help you determine whether redesigning a legacy database is a better course than struggling to bring it forward to a new generation of applications.

This chapter describes some basic principles of database design you should understand to use the Progress Dynamics framework successfully. Some of these are fairly universal, but some have to do with specific aspects of the framework. It includes the following sections:

- [Schema naming conventions](#)
- [Normalization and table design](#)
- [Indexing guidelines](#)
- [Validation and other schema information](#)
- [Object IDs and site numbers in Progress Dynamics](#)
- [Using ERwin with Progress Dynamics](#)
- [Conclusion](#)

2.1 Schema naming conventions

This section describes the conventions, which are only guidelines, you should follow when you name an *entity* (database table) or a data field (database field). Here are some conventions for entities and fields:

- Choose an entity or field name (table name or data field name) that is meaningful, and adequately describes the entity or field being defined. Avoid general terms such as *category* and *type*. These cause confusion when used to refer to different kinds of information on different tables in a database. Even if your current database only uses a term in one place, consider that you might integrate other databases into your application in the future. These generic names run the greatest risk of a conflict with future Progress keywords. You can avoid problems by expanding such names into a more specific, meaningful term (for example, “*user_category*” or “*document_type*”).
- Make your entity or field names singular, (for example, “*customer*” not “*customers*”).
- Avoid hyphens as separators. The hyphen is not a valid separator in SQL, Java, and some other languages. At some point, you might need to access your database from these languages, either directly, through one of the Progress DataServers, or through the Progress Open AppServer. Instead, use an underscore as the separator between the “words” of an entity or field name or use mixed-case letters. Because Progress keywords never contain underscores, using at least one underscore in entity or field names guarantees that the name will never clash with any keyword. In addition, underscores help with ERwin integration, as described in a later section.

2.1.1 Field naming conventions

Use the same name for fields in different tables if they can be used as join fields, such as *Customer.CustNum* and *Order.CustNum*. If the fields cannot legitimately be used as join fields between two tables, then make the field names different, even if the information is from the same domain of values (such as “*ShipToNumber*” and “*BillToNumber*”). This convention simplifies query syntax to take advantage of the Progress *<table> OF <table>* construct. It also enables modeling tools such as ERwin, and other tools that need to make associations between tables, to make the correct determinations on how tables can be related.

Progress Dynamics can take advantage of table and field names that are a meaningful string of words. The framework can replace the standard delimiter (whether underscore, capitalization, or something else) with a space to create a meaningful name for a message text or label. For example, if your naming convention uses underscores as delimiters between parts of a name, you can specify that delimiter in the entity import. Then, ERwin’s macros converts the field name “*customer_name*” to “*customer name*” for use in messages and labels. Entity import needs it to determine other things like table codes, description, fields, and so on.

If you use the *ERwin* modeling tool to define the database and generate the Progress schema, there is a macrocode supplied with Progress Dynamics that defines an initial label for every field. The label is the field name with the delimiters replaced by spaces. Observing the naming convention can save a lot of time completing the definition of all the entities and fields of a new database schema.

2.1.2 Table naming conventions

Table names should be unique across all databases used by an application. The unique names enable any table used by the application to be called just by its table name. Progress Dynamics takes advantage of this convention to carry out common operations in a standard way for any database table.

NOTE: Progress uses the table code to associate comments with a record or to log auditing information for a record.

Progress Dynamics presumes that the dump name of every table is unique. Since the dump name can now be any length, this is not particularly an issue. You could use the table name as the dump name. However by convention, Progress Dynamics uses the dump name as an identifier that is displayed in various places. A dump name of five to eight characters is considered standard. You might see this reflected in default display formats.

Of course, you cannot easily guarantee that a table name will always be unique in any set of databases that is used by an application. By convention, the Progress Dynamics Repository database prefixes every table name with a three-letter code followed by an underscore. This convention helps define the nature of the information in the table. It also helps create a name space that makes table name conflicts unlikely. The first two letters of the name are an abbreviation of the application or “module” name for which the table is principally used. In the Progress Dynamics Repository, for example, tables that form part of the framework repository use the “ry” prefix. The third character in the prefix identifies the nature of the data in the table, how volatile it is.

The possible values of the third character are shown in [Table 2–1](#).

Table 2–1: Values for third character in Progress Dynamics table prefixes

Character	Data or table type	Description
c	constant	This data effectively is used as system parameters and value lists and normally does not change.
m	master	These tables contain (usually limited) sets of values referred to by other tables, such as country or sales rep codes. The values are expected to change relatively infrequently.
t	transactional	These tables contain values subject to frequent change during application execution.
r	raw	This data is not validated and not used in Progress Dynamics.

You do not need to observe this particular convention for your tables. However, it can be a useful way of organizing the data in your databases. You can understand, at least basically, a table's use just by looking at its name.

Each table has a unique abbreviation. The abbreviation starts with the table name's three-letter prefix and adds at least two letters from the table name. For example, the `gsc_entity_mnemonic` table's abbreviation starts with "gsc" prefix and adds "em" for "entity mnemonic." To increase the number of possible unique abbreviations, the framework now allows up to eight characters (based on the dump name of the table).

By convention, procedures and logical objects names begin with the abbreviation of the principal table operated on by the procedure or logical object. In this way, you can meaningfully organize and sort an application's procedures and objects based on their names. You do not need to observe this naming convention to use Progress Dynamics successfully. However, it is a useful model for helpfully and consistently naming physical and logical objects.

The Progress Dynamics Repository database serves as an example of these naming conventions. One table maintains attribute groups which identify related attributes that should be managed or visualized together. This table is part of the Repository (ry) module and maintains constant (c) data—values that should not change once they are defined within the framework. Therefore, the table name prefix is "ryc", the full table name is `ryc_attribute_group`, and its abbreviation is "rycag" ("ag" for "attribute group").

Using a complete, meaningful phrase for the table name makes it easy to turn the name into a description that you can insert into messages or reports. By removing the prefix and the underscores, Progress Dynamics refers to this table as the “attribute group” table. The same convention applies to field names and the default labels generated for them.

You can use the abbreviation at the beginning of every file name or object name that maintains this table. The SDO for the table with all fields might be `rycagfull.o.w`. You might name the primary Progress SmartDataViewer™ for the table `rycagview.w`. Any other Viewers would have names beginning `rycag` and ending in `v`, or whatever convention is appropriate to your application.

2.1.3 Dynamic object naming conventions

Dynamic objects use the same naming conventions as tables. From the previous example, the default dynamic Progress SmartDataBrowser™ with all table fields displayed is named “`rycagfullb`”. Other Browsers would have names ending in `b`. This convention automatically groups procedures and objects pertaining to a particular table in a sorted listing of filenames or object names. The standard final letter makes it easy to identify the type of object.

NOTE: This convention is the reverse of the standard ADM2 convention of beginning filenames with a letter or more that identifies the type. Either convention can work; using a consistent standard is the important goal.

Progress Dynamics keeps track of both static (physical) and dynamic (logical) objects in its Repository. For a dynamic object, the Repository stores all the data needed to define and generate the object at run time. The object’s name (such as `rycagfullb`) is simply a key for locating the object; it does not correspond to any physical source file. Even though procedural objects are run as Progress r-code, they are also registered in the Repository. The default Repository name for these objects is the filename of the procedural object with its extension, product module, and relative path.

The relative pathname and filename extension of a procedural object are stored in different fields from the object name. There is no significant difference between the names of static and dynamic objects. The product module name is included in the filename by default.

2.1.4 Naming length limits

There are no strict limits beyond the normal Progress limits on lengths of table names and field names in Dynamics. But if you observe some sensible conventions, it is easier to use your database within a standard framework. For example, the tools that display these entity or field names expect a reasonable maximum length and provide an appropriate display format. A good limit for table names and field names is 32 characters.

The “Using ERwin with Progress Dynamics” section later in this chapter gives a brief look at the ERwin modeling tool template that you can use to generate a new database schema. That template includes many domain definitions for particular types of fields. These field type definitions include data type, format, and other standard characteristics. When you categorize your database fields in this way, you avoid having fields with no consistent pattern of sizes and formats. Instead, you have a manageable number of discrete subtypes that are used for different purposes. This allows utilities that autogenerate parts of your application, such as application screens, to work more easily and effectively.

Table 2–2 provides a few sample guidelines based on standards from the ERwin template.

Table 2–2: ERwin naming length limits

Type of text	Maximum number of characters
Codes (abbreviations that might not be very meaningful to users but which uniquely identify an object in a compact way)	10
Short descriptions of the type that could be displayed next to a coded value (in a Lookup for example) to make it more intelligible	35
Names	32
Editor for small text blocks	500
Editor for large text	3000

2.2 Normalization and table design

A Dynamics application will always benefit for well-designed tables. Keep in mind the basic principles of normalization when designing your tables:

- A table column contains only one value. For example, a Name column in an Employee table will be easier to work with if it were separated into two columns: FirstName and LastName.
- A table column avoids duplication and repetition of data. The Employee table should not have a Name column if it also has a First and Last column.
- A table column does not store data that does not describe the table. All data bears a relationship to the concept captured by the table name. An Employee table that also contained data on Vendors will confuse your designs.

- A table column does not contain redundant data. The Employee table should not contain three Joe Smith records if only one Joe Smith works at the company.
- A table column is not necessary if it can be derived from other columns. For example, an Employee table may only need to store Zip Code for U.S. addresses, if City and State can be derived from a Zip Code lookup table.

When you begin developing an application using well-designed, normalized database tables, you get immediate benefits:

- Straightforward relationships makes it easy to reuse standard code.
- If the dataset for a particular task can be defined with simple, meaningful queries, the overall development will be less.
- Eliminating excess data allows you to easily update and manage the data.

All in all, good table design means less procedural code and less customization of standard code blocks.

NOTE: The choice between using database triggers or other procedures for logic of this kind, and other aspects of organizing your application logic, is discussed in [Chapter 11, “Building Advanced Business Logic in a Progress Dynamics Application.”](#)

The following sections describe a few other important table design guidelines.

2.2.1 Avoid array fields

One particular characteristic of the Progress database not shared by SQL, or by most other modern database types, is the array field. It is best never to use arrays in your database definition. In a relational database, the use of array fields is not recommended. You might try using array fields to increase performance by reducing the number of related records needed to calculate some information. However, you are likely miscalculating the actual cost of using this technique. Consider that array fields are inherently non-normalized—they attempt to represent a one-to-many relationship in a single record. Arriving at the appropriate extent for an array is often an arbitrary decision. If the original extent turns out later to be insufficient, you must make a schema change, and probably change your application code, to deal with this issue.

Arrays can lead to large records that span multiple database blocks. This increases the number of reads required to retrieve a single record and reduces any performance gained by using them. Transporting such large records across a network requires increased bandwidth and decreases performance. Array values cannot be efficiently indexed, cannot be used as a join field, and cannot be accessed using SQL syntax (for example, from a reporting tool that will want to report on the data those array fields hold). You cannot filter array values; that is, you cannot write a single query statement to retrieve records where one of the values in an array field matches a filtering condition. These are all good reasons to avoid arrays.

Given the problems you might encounter, you should use arrays only for some very particular purpose on the server-side of your application. If an array is necessary, use it for data that is only accessed in your business logic. The data cannot be used in any way outside the server-side procedure where it is manipulated. Standard Progress Dynamics components cannot display array data. You cannot manipulate the data on the client, sort or filter on it, or report on it.

If you define a SmartDataObject™ for a table with an array field, the array is expanded into a series of discrete fields for each element. Usually, this does not result in the kind of client-side representation that you want for the data. The statement defining the temp-table that passes data between the client and server might easily exceed the maximum length of a Progress 4GL statement. That would effectively make the table unusable with Progress Dynamics.

NOTE: The framework converts imported array fields into fields with ordinal numbers appended. Address[] becomes Address1, Address2, Address3, and so on. This can lead to conflicts if you already have similarly named fields in your database.

2.2.2 Reduce field numbers

A tables with too many fields can cause some of the same problems seen with array fields:

- Excessive record length
- Inefficient data access
- Overflow of statement limits in the standard temp-table and a standard query created by the framework to manage the data

You can break up a large number of related fields into logical subgroups that often are more efficient to handle. How many fields are too much is hard to pinpoint, but even a hundred fields in a single table can cause problems. If there is a subgroup of fields that might be optional for a particular master record, it is better to place them on a separate table. Rather than having empty elements in an over-large master record, you can simplify the database by not defining a record for these fields when they are not used. Grouping fields in this way makes managing the data easier. You can use the Progress SmartDataObject™ and the SmartBusinessObject™, the standard Progress Dynamics mechanisms for managing sets of related tables, to handle Adds, Deletes, and Updates on a one-to-zero-or-one basis to support such cases.

2.2.3 Avoid constraints that lead to larger transaction scopes.

Do not to impose unnecessary constraints on data that make coding more difficult. For example, by allowing an Order to have no OrderLines, a user can add the Order header first and then add the lines in separate transactions. Built-in referential integrity constraints would force the user to add the lines in the same transaction. Often, a flag on the Order indicating whether it has any lines is better than forcing the user to enter the first line.

2.2.4 Consider distributed database access

When designing a new database, considering the implications of distributed database access is critical. It is no longer practical (or even possible) to have the direct record-by-record control over locks and scoping that older Progress applications generally have. Records are read from the database in batches. They are then passed in a temp-table to a client running on a different machine. Changes are made without any direct connection to the database. Then, those changes are passed back to the server to be updated in a server-side transaction. This mechanism is not particular to Progress Dynamics or to SmartObjects, but simply a consequence of managing data in a distributed environment.

Because of that mechanism, it is neither desirable nor possible to hold record locks for an entire client-side update operation. A few additional database fields combined with standard application logic can make it much easier to manage transactions in this environment. For example, put an Updating flag field in a header record for an order. Then, have the order update logic in the standard beginTransactionValidate procedure of the SDO or SBO first check this flag before it begins executing the business logic associated with an order. When the update logic sets the flag, it can effectively “reserve” all the related records associated with an update without using actual transaction locking.

If the update logic checks this flag before allowing an update to begin, then two users cannot update the same set of related records concurrently. Alternatively, you can set such a flag in a separate transaction when one or more records are read for an update transaction. This effectively locks out other users from starting an update without using conventional record locks.

Such additional fields are a useful part of any new database design. They can also be a useful, or even necessary, addition to an existing database.

2.3 Indexing guidelines

When designing a database, there are indexing guidelines that you should consider. First, no table should be defined without at least one index. This is generally a good policy. Not observing this causes many performance problems. However, Progress Dynamics has particular problems dealing with tables that have no index.

Try not to make a key that has a real meaning the only unique identifier on a table. There should be a key whose values cannot be forced to change. Any name or number whose value is not a completely arbitrary and never-changing sequence value may be a problematic choice for a key value. Certainly, a value that has a good probability of changing is a poor candidate for a key.

For example, textbooks on relational theory and SQL (not to mention Progress) routinely use examples where a meaningful key value is used as the sole join field for two tables, such as Customer.SalesRep and SalesRep.SalesRep. The Sales Rep's initials are stored as the value for these fields. However, this definition is inherently denormalized because the Sales Rep's initials are stored in two different tables and might change (as the result of marriage, for instance).

Every table should have a unique identifier. That unique identifier should be an unchanging sequence value that has no external meaning at all. All relationships to the table are then based on this unique identifier. Having alternate keys that are multi-component is fine, as long as there is another "primary" way of relating entities to one another.

Progress Dynamics supports a specific mechanism for defining keys called Object IDs, which are discussed in the ["Object IDs and site numbers in Progress Dynamics"](#) section later in this chapter.

Cascading data items (such as the SalesRep initials) down from parent to child is sometimes necessary, such as for performance reasons, but this is really a form of denormalization and should be avoided. You might want to provide a meaningful piece of information, the ID of the Sales Rep in this example, in the child table to avoid having to retrieve the associated parent record. But, putting the key there might not accomplish what you want. In this example, you might want to display the actual Sales Rep's Name, rather than initials, when displaying a Customer record. To get the name, you have to join to the SalesRep table anyway. If you use an arbitrary numeric sequence number to join the two tables, you avoid writing code for cascading changes.

Other points to consider about indexes and defining foreign key relationships are:

- Always put an index on a Foreign Key field that involves only that field, Customer.SalesRep, or more properly, Customer.SalesRepSequence in the example. This allows you to efficiently determine relationships, such as Customers for a SalesRep, at the start of the index at the least.
- Never use the RECID or ROWID of a record to relate records. RECIDs and ROWIDs are not preserved across database dumps and reloads. In fact, with support for multiple storage areas, they are no longer unique across the database.

2.4 Validation and other schema information

You should avoid using schema field validation because it is not executed in Progress Dynamics applications. The Progress Dynamics Object Generator eliminates the inheritance of schema validation from a generated SDO's temp-table definition. If the framework did not have this behavior, references to a CAN-FIND in the field validation expression would be compiled into any Viewer or other frame-based component built against that SDO. The resulting Viewer would not run without the database connected on the client, which is unacceptable in a distributed deployment.

Compiling field validation into the user interface can cause other problems. In particular, the standard Progress frame behavior does not allow you to leave a field whose value does not pass the validation, even if, for example, you are generating a LEAVE event by pressing a button. Generally, step-by-step field validation is not considered appropriate for GUI applications. A user expects to enter data more flexibly and have control over when to correct an invalid value.

In addition, dynamically generated Viewers or other UI components, including those of a non-Progress interface, cannot use the schema validation. Therefore, you cannot count on it as the primary mechanism for verifying correctness of your data. You should build field validation into the back-end business logic where it always executes.

You can use Lookups and Combos to enforce the kind of validation done by a CAN-FIND or a validation expression that defines a list of valid values. Lookups and Combos provide autocompletion and a list of values from which the user can select. This is preferable to simply flagging an invalid entry and forcing correction before the user can leave the field.

On the other hand, specifying in the schema the appropriate values for the Label, Column-Label, Format, and VIEW-AS phrase for a field, is very useful. This determines the default look of any Viewer or Browser you build. For a dynamic Viewer, it gives you the proper representation of all your fields without any coding or customization.

2.5 Object IDs and site numbers in Progress Dynamics

This section introduces you to Progress Dynamics' Object IDs. The Object ID is central to the Progress Dynamics Repository structure. It can also be valuable in your application database.

2.5.1 Object IDs

Every Progress Dynamics table has a unique identifier called an *Object ID*. The field has a DECIMAL data type. The field's name is the table name, without the three-character table-type prefix, and ending with the suffix “_obj”. For example, the Object ID field for the ryc_attribute_group table is attribute_group_obj. The Object ID values are assigned using a globally unique site number, as described later in the “[Site numbers](#)” section.

Because the Object IDs are assigned using globally unique site numbers, every record in every table in every Progress Dynamics-managed database can have a common key value that is guaranteed to be unique worldwide, like a RECID. Using this mechanism, you can easily locate and refer to any record in any table in a database using a standard mechanism. You only need to know the table name (or its abbreviation, its abbreviated name, or dump name) and its Object ID. The framework does this to support a number of standard operations that can apply to any table, including auditing and record-specific data security.

The unique site numbers allow data from different Repository databases to be combined without fear of collisions between Object ID values. Application tables can also be safely combined as long as there are no other unique key values that could conflict between databases. You can use this technique to merge dynamic application components from different sources into a single application or suite of applications.

For example, you develop a new feature that includes one or more dynamic (data-driven) components, such as a table maintenance window, and you want to submit it for inclusion in Progress Dynamics. Your Repository database must have a unique site number. Otherwise, your Repository data could not reliably be deployed to the central Repository database. You would encounter the same issue if you wanted to integrate your application with a Progress Dynamics application from another vendor.

For this reason, **it is imperative that you obtain a unique site number** for your development database and for each distinct, deployed database before undertaking any work that might require sharing Repository or application data between databases, for cooperative development or as part of the deployment of your application. Also note that the site ID becomes part of the object ID.

Because the Object ID is useful for comments, auditing, and similar tasks, you should provide Object IDs for all the tables in any database you create. Even if you have an existing database, you might want to consider adding Object IDs to the tables where the size of existing deployed databases permits.

The standard naming convention for Object IDs provides other benefits. For example, the Object ID is always used to define foreign key relationships. If the table name is “xyz_table_name,” its Object ID field is “table_name_obj.” If there is a foreign key relationship to another table, then the table_name_obj field also appears in that table. It is easy to build tools that recognize and take advantage of this relationship.

Having the Object ID field appear on the foreign table yields another benefit. The framework searches for fields from the foreign table, in addition to a foreign key field, that are appropriate to retrieve and display. These are fields that describe the foreign key field. The framework finds these fields by searching for names containing strings from a list of meaningful descriptive field names. The list below shows the default values:

- reference
- code
- last_name
- short (for “short description”)
- id
- desc
- name

The Object Generator uses this technique to identify useful fields from related tables to include in the SDO field list. If all of the following are true:

1. You are building an SDO for a table.
2. There is a foreign key relationship of this type to another table.
3. That table contains a field name that matches one of the strings that describe the foreign key field.

Then:

1. The other table is added to the SDO's query.
2. The descriptive name field from the other table is added as a read-only field to the SDO field list.

If you are building a new database using the naming convention, you can take advantage of features such as this without additional effort. For more information, see [Chapter 5, "Using the Object Generator."](#)

2.5.2 Site numbers

The framework assigns a value to this field for every table with a standard database trigger. The value is derived from two (or potentially more) database sequences, whose values are generated by a trigger or by adding your own code to a trigger, are combined to form the high-order and low-order portions of the whole-number portion of the field's value. The values of two sequences are combined because a single integer might not be sufficient over time to store all possible values for all database records.

The decimal portion of the value is meant to be a unique site number to identify a particular user organization. The decimal portion is set to the reverse of the site number, so that trailing zeros in a site number do not lose their significance. For example, site number 710 yields Object ID keys ending in .017, and site number 7100 yields key values ending in .0017. This is necessary because .710 and .7100 are the same decimal value. A site number allocation service exists on the Progress Web site (<http://www.progress.com/dynamics/sitenumber>). The service allows application developers to reserve a range of site numbers for all their customers. Thus, every Progress Dynamics database anywhere in the world can have a unique site number.

NOTE: See [Chapter 4, "Preparing to Build Application Objects,"](#) for more information on the site number application.

2.6 Using ERwin with Progress Dynamics

ERwin is an entity-relationship modeling tool. You can use it to design a new database and then generate the Progress schema for the database. Several important capabilities of Progress Dynamics take advantage of features in ERwin. Progress Dynamics provides a customized ERwin template to act as the foundation for using these features.

This section is an overview of using ERwin together with Progress Dynamics to design a database. Note that ERwin is an entirely independent product and is not sold or supported by Progress Software Corporation, although Progress does provide templates and macrocode.

The customized template includes custom data types and formats for Progress Dynamics. You can use them to generate Progress field definitions with a particular format and a standard data type. Using the many data domains in the template, you can set a field's format, default value, or other characteristics simply by picking a domain.

For example, [Figure 2-1](#) shows that Object ID fields are defined as type “o_obj”. This convention provides a decimal data type definition of the right kind, and also associates a database trigger with the field to assign the Object ID field.

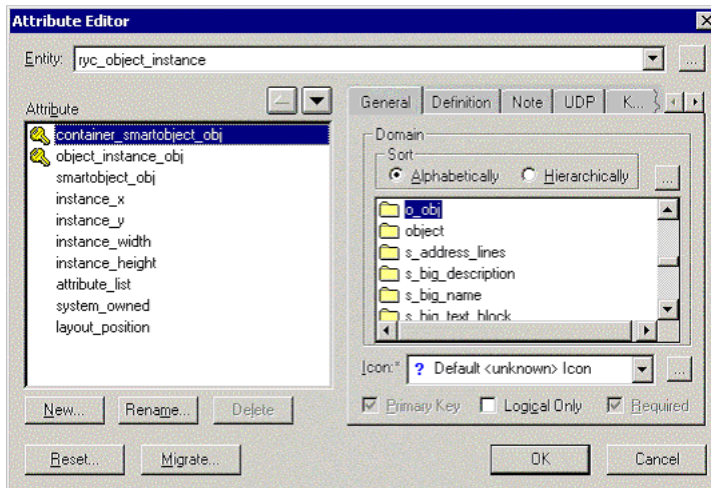


Figure 2-1: ERwin Attribute Editor showing object ID definitions

Progress Dynamics also takes advantage of ERwin macrocode to generate 4GL database trigger procedures for your Progress database, as illustrated in [Figure 2–2](#).

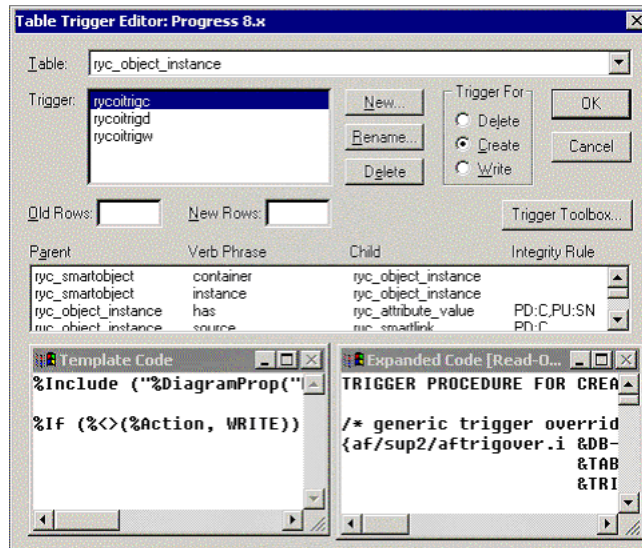


Figure 2–2: ERwin Table Trigger Editor

[Figure 2–3](#) shows the expanded Template Code editor for the Create trigger example. Note the references to macros that expand into Progress 4GL code to enforce referential integrity checks and other standard operations.

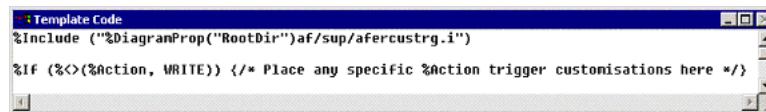


Figure 2–3: Expanded Template Code editor

Figure 2–4 shows the 4GL code generated by these macros. The ease of generating this code is a tremendous benefit in creating your application database and its supporting triggers.

```

Procedure - Untitled:1
File Edit Search Compile Help

TRIGGER PROCEDURE FOR CREATE OF ryc_object_instance .

/* generic trigger override include file to disable trigger if required */
(a/sup2/aftrigover.i <DB-NAME      = "RYDB"
      <TABLE-NAME    = "ryc_object_instance"
      <TRIGGER-TYPE  = "CREATE")

/* Created automatically using ERwin ICF Trigger template af/sup/afercustry.i
*/

<SCOPED-DEFINE TRIGGER_TABLE ryc_object_instance
<SCOPED-DEFINE TRIGGER_FLN rycoi
<SCOPED-DEFINE TRIGGER_OBJ object_instance_obj

DEFINE BUFFER lb_table FOR ryc_object_instance.      /* Used for recursive relationships */
DEFINE BUFFER lbl_table FOR ryc_object_instance.     /* Used for lock upgrades */

DEFINE BUFFER o_ryc_object_instance FOR ryc_object_instance.

/* Standard top of CREATE trigger code */
(a/sup/aftrigtopc.i)

ASSIGN ryc_object_instance.(<TRIGGER_OBJ) = mip-next-obj().

/* Update Audit Log */
IF CAN-FIND(FIRST gsc_entity_anemonic
      WHERE gsc_entity_anemonic.entity_anemonic = 'rycoi':U
      AND gsc_entity_anemonic.auditing_enabled = YES) THEN
  RUN af/app/afauditlqp.p (INPUT "CREATE":U, INPUT BUFFER ryc_object_instance:HANDLE,
      INPUT BUFFER o_ryc_object_instance:HANDLE).

/* Standard bottom of CREATE trigger code */
(a/sup/aftrigendc.i)

/* Place any specific CREATE trigger customisations here */

```

Figure 2–4: 4GL code generated by ERwin macros

The ERwin customizations in Progress Dynamics also generate default field labels automatically. If you construct field names to be a meaningful sequence of words, delimited by underscores or another delimiter you can specify, then a default label is generated with the delimiters replaced by spaces.

These benefits save time when generating a new database schema using ERwin for a Progress Dynamics application.

2.7 Conclusion

This chapter has shown some of the benefits of observing the Progress Dynamics conventions if you design a database for a new application. You do not need to observe these conventions or modify your existing database schema to use Progress Dynamics successfully. However, observing at least some of these conventions enables you to make better use of Progress Dynamics and, in particular, to define your application components more quickly.

Progress Dynamics Integration with the AppBuilder

The Progress AppBuilder remains the starting point for Progress application development, even with the addition of all the new tools provided with the Progress Dynamics framework. To support development both with and without Progress Dynamics, the AppBuilder operates in two different modes. In its standard mode, its menus appear as they do in Progress running without Progress Dynamics. When you run the AppBuilder with Progress Dynamics, the menus of the AppBuilder change to provide access to all of the new tools that are part of the Progress Dynamics framework. This chapter provides an overview of all the Progress Dynamics tools accessible from the AppBuilder. It describes some of the tools and points to other chapters for detailed discussions of most others.

This chapter includes the following sections:

- [Starting the AppBuilder](#)
- [Progress Dynamics AppBuilder UI enhancements](#)
- [Progress Dynamics Administration window](#)
- [Progress Dynamics Development window](#)
- [Design windows for Repository objects](#)
- [Creating new Repository objects](#)
- [Opening objects for editing](#)
- [Saving an object to the Repository](#)

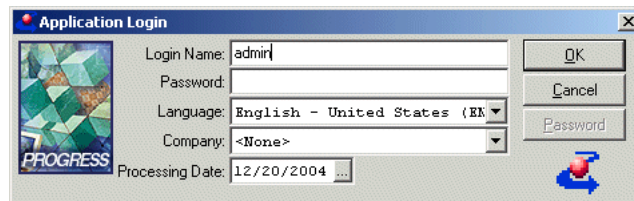
- [Adding a file to the Repository](#)
- [Editing properties for Repository objects](#)
- [Running objects](#)
- [Closing Repository objects](#)
- [Object palette and object template information stored in Repository](#)
- [Summary](#)

3.1 Starting the AppBuilder

See [Progress Dynamics Installation Guide](#) for details on how to install and configure Progress Dynamics.

Follow these steps to start the AppBuilder:

- 1 ♦ If the Progress Dynamics databases are not already auto-started, from the Windows Start Menu choose **Start→Progress Dynamics→Start Progress Dynamics DB Servers**.
- 2 ♦ From the Windows Start Menu, choose **Start→Progress Dynamics→Progress Dynamics Development**. The Application Login window appears:



This standard login window (which you can customize for your application or your organization) allows the user to:

- Choose a language to operate in
 - Specify a Login Company or organizational entity, which is used to apply appropriate customizations and security settings (the default is taken from the user profile)
- 3 ♦ Enter the login information in the Login window, then choose **OK**. The Application Login window registers you as a Progress Dynamics user.

The AppBuilder main window appears and its status bar updates to show your full Progress Dynamics user name as well as the current company you selected in the Login window. You can now start Progress Dynamics development using the AppBuilder.

3.2 Progress Dynamics AppBuilder UI enhancements

This section describes the changes made to the AppBuilder user interface that help you develop Progress Dynamics applications. It describes enhancements made to the AppBuilder main window as well as the new Progress Dynamics Administration and Progress Dynamics Development windows.

3.2.1 Main window enhancements

The AppBuilder main window provides access to Progress Dynamics development activities, as shown in [Figure 3–1](#). The window is wider to accommodate additional menus, toolbar icons, and additional Progress Dynamics-related status bar items for current user and company.

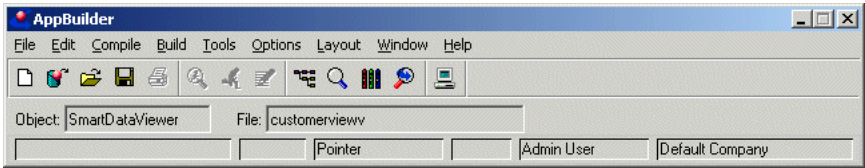




Figure 3–1: Progress Dynamics enabled AppBuilder main window

The special options described in [Table 3–1](#) are available in the AppBuilder main window specifically for Progress Dynamics development.

Table 3–1: AppBuilder main window Progress Dynamics-specific menu

Menu	Description
Build menu	Provides access to Progress Dynamics development features for creating and maintaining application objects.
Open Objects icon 	Allows you to browse objects in the Repository and open them for editing in the AppBuilder. If you select an item in the browse and right-click, you can choose to remove the item from the Repository or display its properties.
Dynamic Properties icon 	Displays the Dynamics Properties sheet for the selected object in the design window.
Status bar addition	Displays the Progress Dynamics user name and company you specified in the Login window.

The following sections describe the new menus and menu options for the Progress Dynamics AppBuilder main window.

3.2.2 File menu enhancements

Table 3–2 describes the new options on the File menu that support Progress Dynamics development.

Table 3–2: Progress Dynamics-specific options on the File menu (1 of 2)

Menu	Description
Open Object	Runs the Open Object dialog box where you can choose a Repository-based object to open in the AppBuilder.
Open Associated Procedure	If the currently selected object has a custom super procedure or data logic procedure associated with it, opens that procedure.
Open File	Runs the Open File dialog box where you can specify a file system object to open in the AppBuilder. This is the same menu item as the Open menu item in the standard AppBuilder.
Save As	Save As works as you would expect, except with dynamic containers. The correct way to save a dynamic container under a new name is to use the Container Builder to create a new container with the existing container as its template.
Save Static Object As Dynamic	<p>When you select an eligible static object, this option appears on the File menu. Allows you to create a new dynamic object from the static object. You specify the Product Module and the option exists to include Repository Modules, modules of the framework.</p> <p>NOTE: If you have a StaticDataViewer (SDV) that contains a SmartDataField (SDF), and you want to save the SDV as a dynamic object, you must first register the SDF in the ICFDB repository. If you register the SDF before you save the static viewer as a dynamic viewer, the SDF is referenced properly by the new dynamic viewer. However, if the SDF is not registered before you try to migrate the viewer from static to dynamic, you get an error message requesting that you register the SDF and the dynamic version of the viewer is not created.</p>
Save Dynamic Object As Static	When you select eligible dynamic object, this option appears on the File menu. Allows you to create a new static object from the dynamic object. This only applies to dynamic viewers and dynamic SDOs.
Register in Repository	Displays the Register in Repository dialog box you can use to add an AppBuilder-created file to the Repository as a static object.

Table 3–2: Progress Dynamics-specific options on the File menu (2 of 2)

Menu	Description
Re-Login	Displays the Login window for you to login as a different user or with a different company or with a different language.
Session Reset	Use the Session Reset dialog box to stop or start the Progress Dynamics Managers, which clears all client-side caching, and connect or disconnect the AppServer.

File→New

When you choose **File→New**, the New dialog box appears, as shown in [Figure 3–2](#).

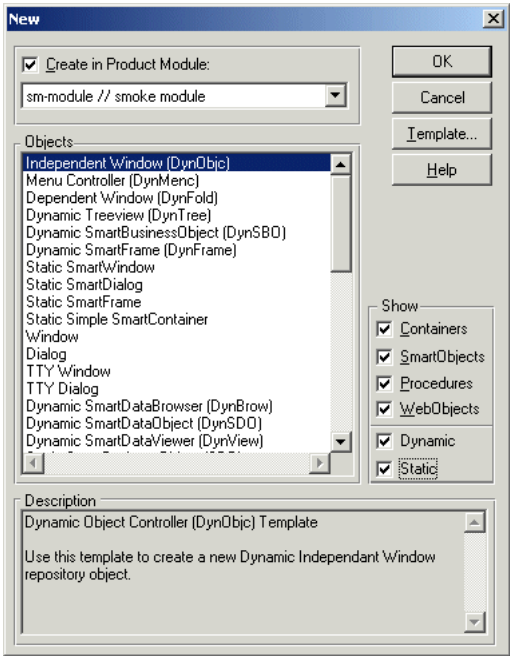


Figure 3–2: AppBuilder New dialog box

Click the Template button to temporarily treat an AppBuilder procedure file as a template and place it in the Objects list for the current AppBuilder session.

The objects displayed for each Show filter type are determined by AppBuilder Custom (.cst) files. See the [“Object palette and object template information stored in Repository”](#) section for more details.

File→Open Object

The Open Object dialog box, shown in [Figure 3–3](#), provides access to opening Repository objects for editing in the AppBuilder. It is limited to displaying only the object types described in the “[Repository object types the AppBuilder edits](#)” section.

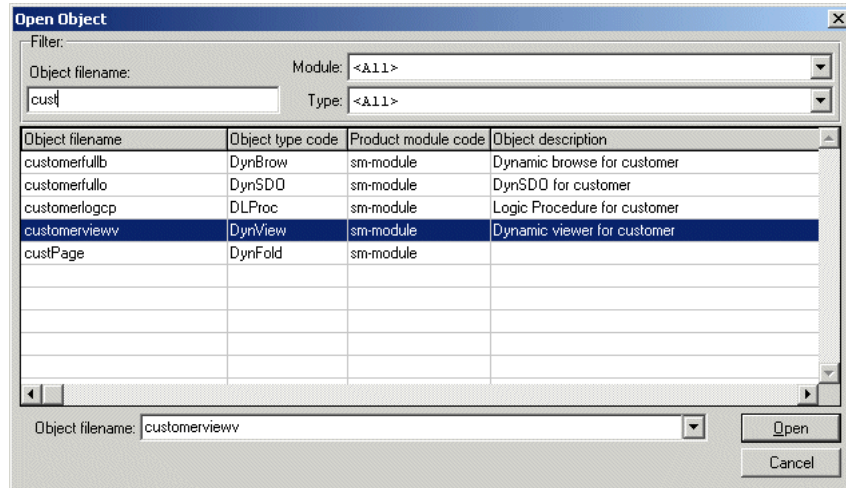


Figure 3–3: Open Object dialog box

When you select an object in the browse and right-click, you have the option to:

- Open the object
- Remove the object from the Repository
- View the property sheet for the object

File→Save Dynamic Object as Static

This option invokes the default **Save As** dialog, as shown in [Figure 3–4](#), but is only enabled for dynamic objects. This means that it allows you to save a copy of a current dynamic object as a static object with a new name. You can also choose to register the new object in the Registry or choose deployment options for the object. Later sections in this chapter cover these topics.

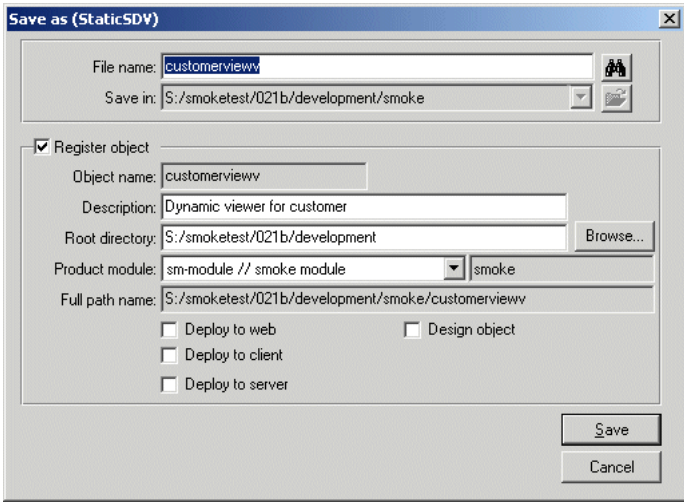


Figure 3–4: Save As dialog box

File→Save Static Object as Dynamic

On selecting the **Save Static Object as Dynamic** option, a dialog box opens, requiring you to specify the Product Module as shown in [Figure 3–5](#).

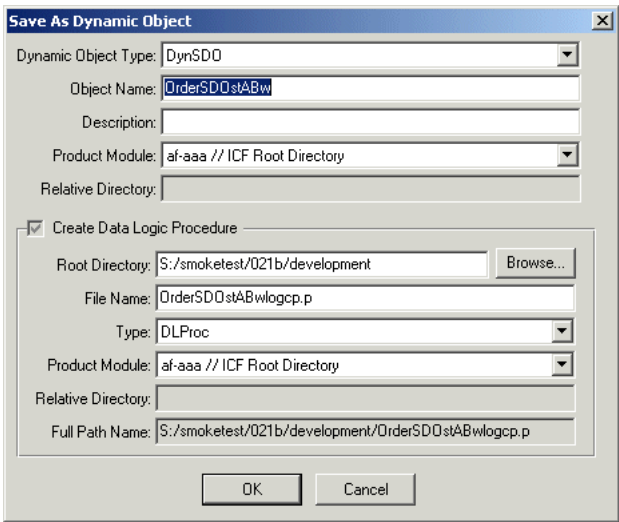


Figure 3–5: Save As Dynamic Object dialog box

You can also specify the name and particulars of a custom super procedure for the dynamic object.

NOTES:

- The object name defaults to the name of the static object, the only difference being the period between the name and extension is removed.
- The Product Module defaults to the last product module used when the last dynamic object was saved, but can be changed.

File→Register in Repository

To add the procedure to the Repository as a static object, open an existing static AppBuilder procedure and select **File→Register in Repository**. The dialog box shown in [Figure 3–6](#) appears.

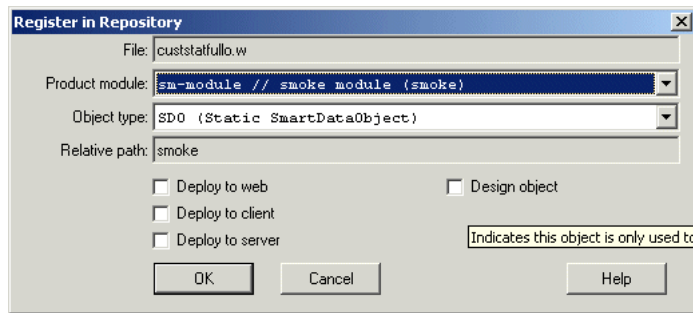


Figure 3–6: Register in Repository dialog box

The check boxes at the bottom allow for marking the object for deployment. If any of the check boxes are selected, the object will be deployed to the specific destinations.

After you complete the dialog box entries and choose **OK**, the framework adds the file to the Repository as a static object.

3.2.3 Build menu

The AppBuilder's Build menu provides access to Progress Dynamics development features for creating and maintaining dynamic application objects. The Build menu in the AppBuilder main window contains the options described in [Table 3–3](#).

Table 3–3: Build menu options

Menu	Description
Object Generator	Runs the tool you can use to generate Progress SmartDataObjects™, logic procedures, dynamic SmartDataBrowsers, dynamic SmartDataViewers, and Data Fields (described in Chapter 5, “Using the Object Generator”).
Container Builder	Runs the Container Builder tool, which you can use to create containers and container templates for windows and pages of tab folders (described in Chapter 8, “Using the Progress Dynamics Container Builder”).
Toolbar and Menu Designer	Runs the Toolbar/Menu Designer tool, which you can use to build custom menus and toolbars for your application (described in Chapter 12, “Using the Toolbar and Menu Designer.”
Repository Maintenance	Runs the Repository Maintenance tool you can use to edit object and instance attributes. This is a low-level tool and usually should be avoided. Everything you can do here can be done with more user-friendly tools in the AppBuilder environment.
SmartDataField Maintenance	Runs the Progress SmartDataField™ Maintenance property sheet where you can define Dynamic Lookups and Combos. These objects are described in Chapter 7, “Building Progress Dynamics Lookups and Combos.”
Tree Node Control	Runs the tool where you can create individual TreeView nodes for a dynamic TreeView window. You then run the Dynamic TreeView Builder to assemble these into a finished TreeView window.
Dynamic TreeView Builder	Runs the Dynamic TreeView Builder where you can create, generate, and edit dynamic tree view and menu objects. The tree view is a special window layout and Progress Dynamics object with which you can create a single window with many pages, navigated by means of a tree view control on the side of the window. (The Dynamic TreeView Builder is described in Chapter 9, “Building Progress Dynamics TreeView Windows.”)
Set Site Number	Lets you specify the Repository site number information. Every Progress Dynamics user organization should have a unique site number or range of site numbers allocated to them. Getting and using a site number is described in Chapter 4, “Preparing to Build Application Objects.”

3.2.4 Compile menu enhancements

Figure 3–7 shows the enhanced AppBuilder Compile menu.

Run	F2
Check Syntax	Shift+F2
Debug	Shift+F4
Dynamic Launcher...	Ctrl+F2
Clear Repository Cache...	
Code Preview	F5
Close Character Run Window	

Figure 3–7: Progress Dynamics AppBuilder Compile menu

Table 3–4 shows the new options on the Compile menu support Progress Dynamics development.

Table 3–4: Compile menu options

Menu	Description
Dynamic Launcher	Displays the Dynamic Launcher window where you enter the name of a Repository object and choose Run to run the object. The Dynamic Launch utility is described in Chapter 8, “Using the Progress Dynamics Container Builder.”
Clear Repository Cache	<p>Clears client-side caching of object Repository data to ensure that you are running the latest versions of dynamic objects. The Progress Dynamics Manager procedures retrieve Repository data on the server and send it to the client as requested. Generally this data is cached in the client session so that it does not need to be constantly retrieved.</p> <p>NOTE: This option does not clear translations cache, security cache, or toolbar and menu cache.</p>

3.2.5 Tools menu

Table 3–5 describes the options on the Tools menu that support Progress Dynamics development.

Table 3–5: Tools menu options

Menu	Description
Administration	Runs the Progress Dynamics Administration window.
Development	Runs the Progress Dynamics Development window.

These windows are described in the following sections.

3.3 Progress Dynamics Administration window

The Progress Dynamics Administration window, shown in [Figure 3–8](#), provides access to Progress Dynamics tools that let system administrators setup and maintain the application environment.



Figure 3–8: Progress Dynamics Administration window

[Table 3–6](#) provides a brief overview of the Progress Dynamics Administration window menus. These functions are covered in detail in later chapters of this guide.

NOTE: Some of the frequently used functions are also available from the AppBuilder main window or the Progress Dynamics Development window.

Table 3–6: Progress Dynamics Administration window (1 of 3)

Menu	Description
File	Described in detail in the “Administration and Development File menu” section.
Application	Contains a number of miscellaneous maintenance functions (such as Translation, Status, Multi Media Type) described in later chapters.

Table 3–6: Progress Dynamics Administration window*(2 of 3)*

Menu	Description
Deployment	Includes a number of deployment-related options, including tools for creating, exporting, and importing <i>Deployment Datasets</i> —sets of database records in the Repository or in your application database that need to be deployed with the application. For more information on deployment, see the Progress Dynamics Administration Guide .
Security	Contains functions you can use to define: <ul style="list-style-type: none"> • Users and categories of users • Login Companies or other organizational entities to which you can apply security • Security controls or tokens (such as menu items, buttons, and folder tabs), ranges of data, and database fields to which security can be applied. For more information, see Chapter 13, “Defining Progress Dynamics Application Security.”
Session	Contains functions you can use to manage data used by all of the parts of Session management, including Configuration information and definition of Logical and Physical Services used by the Connection Manager , etc. For more information on session management, see the Progress Dynamics Administration Guide .
System	Provides access to a number of functions you can use to set up your system, including: <ul style="list-style-type: none"> • Setting a site number. • Importing and managing entity data describing your database tables to the Repository. • Managing online help. • Managing messages. • Manage filter sets. Most of these topics are discussed in Chapter 4, “Preparing to Build Application Objects.”
Transaction	Provides access to Audit Control, where you can view auditing information at the database record level.

Table 3–6: Progress Dynamics Administration window (3 of 3)

Menu	Description
Links	Provides access to other Progress Dynamics windows.
Window	Lets you alternate the setting for whether invoking the same function more than once will bring up multiple copies of the supporting window or only one. It also provides links to other windows in the session.
Help	Accesses the online help, which provides information about the Progress Dynamics Administration window user interface.

3.4 Progress Dynamics Development window

The other independent Progress Dynamics window you can bring up from the AppBuilder Tools Menu is the Development window, shown in [Figure 3–9](#).



Figure 3–9: Progress Dynamics Development window

This window provides access to some ancillary Progress Dynamics development operations. Generally you would not expect to invoke this menu frequently for new application development.

[Table 3–7](#) describes the menus on the Development window.

Table 3–7: Development window menus (1 of 2)

Menu	Description
File	Described in detail in the “Administration and Development File menu” section.
Attributes	Provides access to functions that let you define new attribute names and groups, as well as individual attribute value records. (Although, typically you maintain individual attribute value records with other high-level tools.)

Table 3–7: Development window menus*(2 of 2)*

Menu	Description
Objects	Provides access to: <ul style="list-style-type: none"> • Maintenance of the Object Types that describe the basic characteristics of each type of application object (such as DynBrow). • The list of standard Progress SmartLinks recognized by the framework. • The list of basic container types used in creating dynamic windows. • The utility that lets you maintain runtime customizations for your applications. • The utility that lets you replace all instances of an object (in existing containers) with a new object.
Build	Duplicates functions you can get at directly from the AppBuilder or Administration menus.
SCM	Provides the functions that manage your integration with Progress Dynamics and your SCM tool. Documentation on the Progress Dynamics integration with RoundTable can be found at http://psdn.progress.com/library/progress_dynamics/index.ssp .
Links	Provides access to other Progress Dynamics windows.
Window	Lets you alternate the setting for whether invoking the same function more than once will bring up multiple copies of the supporting window or only one.
Help	Accesses the online help, which provides information about the Progress Dynamics Administration window user interface.

3.4.1 Administration and Development File menu

The Progress Dynamics Administration and Development windows have the same File menu options described in [Table 3–8](#).

Table 3–8: Administration and Development File menu options

Menu	Description
Re-Logon	Displays the Login window so you can change your login information: User name, Language, or Company.
Suspend	Hides all development windows until you enter your password in the Suspended User dialog box shown in Figure 3–10 .
Preferences	Displays the Progress Dynamics Preferences dialog box, shown in Figure 3–11 , where you can set your individual user preferences.
Print Setup	(Administration only) Invokes the operating systems printer setup window.
Translate	Displays the tool that lets you translate all of the text items in the current application window. This Translate menu item is a standard part of every Progress Dynamics File menu until you specifically disable it. For more information on this translation tool, see the Progress Dynamics Administration Guide
Map Help Context	Displays a dialog box that lets you manage help. You can associate help files and contexts within the file with particular application objects.
Desktop	Displays a submenu duplicating the buttons in the Administration Toolbar, which provides access to a number of useful desktop tools, such as text editors, e-mail, Internet browsers, and so on.
Exit	Exit Progress Dynamics.

[Figure 3–10](#) shows the Suspended User dialog box.

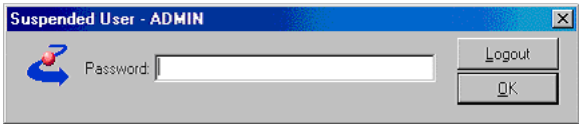


Figure 3–10: Suspended User dialog box

Figure 3–11 shows the Progress Dynamics Preferences dialog box.

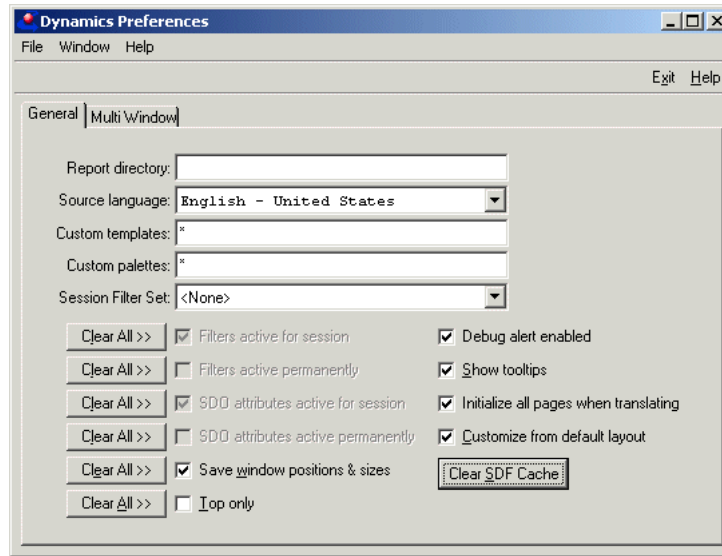


Figure 3–11: Progress Dynamics Preferences dialog box

Table 3–9 describes the options available in the Preferences dialog box.

Table 3–9: Preferences dialog box options

(1 of 2)

Element	Description
Report directory	Not used.
Source language	Specify the preferred language.
Custom templates	Specify any custom templates you wanted loaded during your session. The default is to load all custom templates in the directory.
Custom palettes	Specify any custom palettes you wanted added to the AppBuilder Palette window during your session. The default is to load all custom templates in the directory.
Session filter set	Specify any filter sets you wanted loaded during your session. The default is to load all custom templates in the directory.

Table 3–9: Preferences dialog box options

(2 of 2)

Element	Description
Filters active for session / permanently	When you filter an SDO query using the Filter button on any Progress Dynamics or application window, the framework stores that filter setting and brings that window up with the same filter the next time you access it. You can set here whether you want filter settings to be saved for the duration of the session, permanently (in the Repository database), or not at all. You can also clear current filter settings here.
SDO attributes active for session / permanently	Use these settings to specify whether you want SDO attributes, such as the RowsToBatch for a query, to be saved for the duration of the session, permanently (in the Repository database), or not at all.
Save window positions & sizes	Select to instruct Progress Dynamics to save the position and size of every window you open, so that it comes up the same size and in the same place the next time you open it.
Top only	Forces Progress Dynamics to display the Administration window on top of all other windows.
Debug alert enabled	Sets the Progress session DEBUG-ALERT flag, which adds a Help button to every alert box that comes up in your application. Choosing the Help button displays a Progress 4GL stack trace and lets you invoke the Progress Debugger.
Show tooltips	Specifies to display ToolTips. Clear this option if you do not want ToolTips to display for buttons and fields.
Initialize all pages when translating	Forces Dynamics to initialize all pages it displays when the Translation Manager is active.
Customize from default layout	When you have a custom layout opened in the AppBuilder for a dynamic viewer, and you open a second custom layout, you are presented with a choice (radio set) on how to save the second layout. Should the second layout be stored as differences against the first custom layout, or as differences against the master layout. This preference specifies the default value for that radio-set.
Clear SDF Cache	Click to clear any data currently cached for SmartDataFields.
Multi-Window tab	Lets you set whether you want a separate window to be brought up for each record you select in a Browser that invokes a record maintenance window. Also allows you to force this setting to be permanent or for this session only.

3.5 Design windows for Repository objects

You can open almost all objects directly in the AppBuilder. However, container objects, including SBOs, are maintained by the Container Builder. Open this tool to open the design window for a container.

3.5.1 Options disabled for dynamic object design windows

Table 3–10 lists the AppBuilder options disabled when the AppBuilder's current design window is for a dynamic Repository object.

Table 3–10: Options disabled for Progress Dynamics object Repository design

AppBuilder UI location	Disabled options
AppBuilder main window	Toolbar icons: Print, Procedure Settings, Run, Edit Code
File menu	Register in Repository, Print, Save as Object
Edit menu	Copy to File, Insert from File
Compile menu	Run, Check Syntax, Debug, and Code Preview
Tools menu	Procedure Settings
Window menu	Code Section Editor
Help menu	All options are enabled (none disabled)
Object palette	No special behavior

3.5.2 Repository object types the AppBuilder edits

The AppBuilder can open and edit in its design windows the following Repository object types:

- All static object types
- Dynamic SmartDataObjects (DynSDO)
- Dynamic SmartDataBrowser (DynBrow)

- Dynamic SmartDataViewer (DynView)
- Dynamic containers (Independent windows)

The AppBuilder does not directly open or edit any other types of objects, including Toolbars and Menu objects, and all other dynamic object types. You can open dynamic containers, dynamic treeviews, and dynamic SBOs from the AppBuilder, but AppBuilder launches the appropriate tool for handling that object. To edit these objects, use other Progress Dynamics tools and utilities accessible from the AppBuilder.

NOTE: If you add any object to the Repository, the AppBuilder will display them in the Open Object dialog box, and you can open them using the Open Object dialog box for editing in the associated property sheets or maintenance tools launched from the AppBuilder.

3.5.3 Object names and filename extensions

At this point it is worth noting that the full pathname of a static object can be represented in the Repository in as many as three separate parts. The relative pathname to the file is stored in the `object_path` field in the `ryc_smartobject` table. The object name is stored in the `object_filename` field. The filename extension is stored in a separate field called `object_extension`, and the `object_filename` field holds only the simple filename without any extension. For procedural objects created, the filename extension is stored along with the filename in the `object_filename` field. Progress Dynamics handles both object name forms properly and consistently.

In the case of a dynamic object, which has no physical source file, there is no `object_path` or `object_extension`, and the `object_filename` (somewhat misleadingly named in this case, since it is just a logical name and not an actual filename) is the name given to the object, which for dynamic objects never has an extension.

The naming convention, in which the `object_filename` is always just the simple name of the object, whether it is procedural or dynamic, promotes consistency. It also makes it easier to convert static objects, such as static Viewers and SDOs, to dynamic objects of the same type, without a change to this important `object_filename` key field. For now, it is important to understand that you can store procedural objects with or without the extension in the filename, though for newly created objects it is always stored separately. Framework code that relies on the name is prepared to deal with both forms to preserve compatibility, but you should not rely on the presence of the extension in your work.

Note also that this convention does mean, for example, that you cannot have a window (.w) and a procedure (.p) file with the same base name in your application. But this restriction really existed already, because both of these files in the same directory would compile to a single r-code (.r) file. The names must be unique without considering the directory, so they would not be allowed in different directories either. This restriction is also necessary to support RoundTable.

3.6 Creating new Repository objects

You can use the AppBuilder and the tools it organizes to create one or many application objects in the Repository. The following sections summarize these processes. They are described in detail in [Chapter 5, “Using the Object Generator.”](#)

3.6.1 Creating a single Repository application object

Follow these steps to create a single Repository application object in the AppBuilder:

- 1 ♦ From the AppBuilder window, choose **File→New** or from the AppBuilder Tool Bar, choose the **New** button. The New dialog box appears. (See the [“File menu enhancements”](#) section earlier in this chapter for details.)
- 2 ♦ Select the **Create in Product Module** option if you want to create a new Repository object and select the **Product Module**. If you uncheck the **Create in Product Module** option for a static object, the AppBuilder will not create the object as a Repository object. You can later add the file to the Repository as a static object using the AppBuilder’s Add to Repository option on the File menu.
- 3 ♦ Select the type of object you want to create from the **Objects** selection list.
- 4 ♦ Choose **OK**. The new object opens in an AppBuilder design window.
- 5 ♦ If you selected to create a dynamic object, see the [“Additional steps by object type”](#) section.

Additional steps by object type

If the new application object is a dynamic object, AppBuilder launches a tool or wizard to help you define the object. [Table 3–11](#) shows what happens with the most common objects.

Table 3–11: Creating new objects*(1 of 2)*

Object type	AppBuilder action
Independent window	Opens the Container Builder.
Menu controller	Opens the Container Builder.
Dependent window	Opens the Container Builder.
Dynamic treeview	Opens the Dynamic Treeview Builder.
Dynamic SmartBusinessObject	Opens the Container Builder.
Dynamic SmartFrame	Opens the Container Builder.
Dynamic SmartDataBrowser	Opens a wizard.
Dynamic SmartDataObject	Opens a wizard.
Dynamic SmartDataViewer	Opens a wizard.
Static SmartWindow	Opens a wizard
Static SmartDialog	Opens a wizard
Static SmartFrame	Opens a wizard
Window	Opens in the AppBuilder.
Dialog	Opens in the AppBuilder.
Static SmartBusinessObject	Opens a wizard.
Static SmartDataBrowser	Opens a wizard.
Static SmartDataObject	Opens a wizard.

Table 3–11: Creating new objects*(2 of 2)*

Object type	AppBuilder action
Static SmartDataViewer	Opens a wizard.
Static SmartDataField	Opens a wizard.

3.6.2 Creating many application objects at once

You can create many application objects at one time using the AppBuilder's **Build→Object Generator** option. This option is described in detail in [Chapter 5, “Using the Object Generator.”](#)

3.7 Opening objects for editing

The following sections describe the different options the AppBuilder provides for opening and editing objects.

3.7.1 Opening a Repository object

To open a Repository object in the AppBuilder, follow these steps:

- 1 ♦ From the AppBuilder main window, choose **File→Open Object** or choose the **Open Object** button on the toolbar.

The Open Object dialog box appears. (See the [“File menu enhancements”](#) section for details.)
- 2 ♦ Change the current Product Module if the object resides in another Product Module.
- 3 ♦ Change the object type to display only objects of a specific type.
- 4 ♦ Enter the logical name of the object you want to open in the **Object filename** field or select the object from the objects displayed. The **Object filename** field at the top of the dialog box will search through the list of objects for matches as you type. This allows you to focus likely matches in the browse and select the correct object when it appears in the browse. The **Object filename** combo field at the bottom allows you to select an object from a list of recently opened objects.
- 5 ♦ Choose **OK**.

The selected object opens in an AppBuilder design window.

3.7.2 Opening files not in the Repository

Follow these steps to open and edit files that are not registered as objects in the Repository:

- 1 ♦ From the AppBuilder main window, choose **File→Open File** or choose the **Open** button on the toolbar.

The **Open** dialog box appears.

- 2 ♦ Change the current folder if the file to open is located in a different path than what is displayed.
- 3 ♦ Change the file type to display only files of a specific type.
- 4 ♦ Type the name of the file you want to open in the **File Name** field or select the file from the list of files displayed.
- 5 ♦ Choose **Open**.

The selected file opens for editing in an AppBuilder design window. The AppBuilder handles the file as a non-Repository object even if the file was previously created or added to the Repository as a static object.

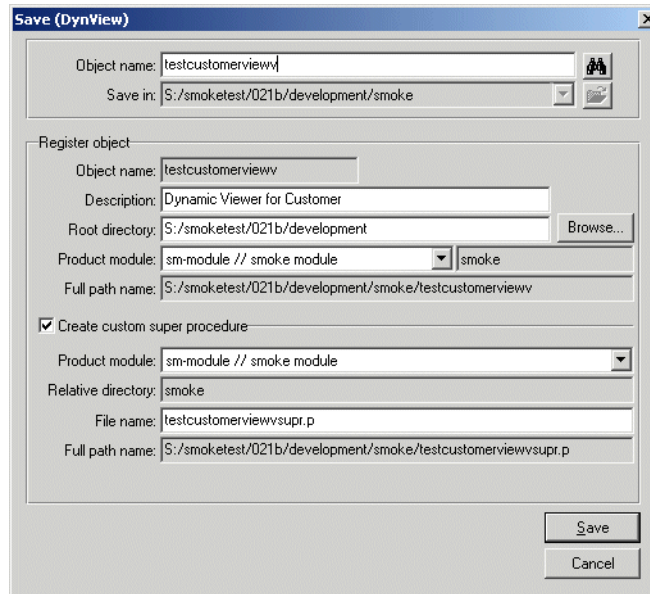
3.8 Saving an object to the Repository

To save an object to the Repository, follow these steps:

- 1 ♦ Click in the Repository object design window to make it the current AppBuilder design object.
- 2 ♦ From the AppBuilder main window, choose **File→Save**, **File→Save Dynamic Object as Static** or **File→Save Static Object as Dynamic**.

Save invokes the normal **Save** dialog with options to let you register the object and specify deployment options.

See the “[Saving a dynamic object as static](#)” section below for more information on this option.



- 3 ♦ For **Static**, type the filename as well as the extension you want to use to save the object to the file system, then choose **OK**. The AppBuilder saves a physical file in the file system for that object.
- 4 ♦ You can also specify a custom super procedure to go with this dynamic object, or a data logic procedures for SDOs and SBOs.
- 5 ♦ Choose **OK**.

If the object is an existing static Repository object, the object is saved back to the file system in the file from which you opened it.

When a static object is saved to the Repository, the whole object is not actually stored there. Only certain properties about the object are registered in the Repository, such as its name and object type. The actual source of the object is still saved to the file system in the location you specified. The location you specify to save a static object file, using the Save As dialog box, can be different from the default path for the object’s Product Module.

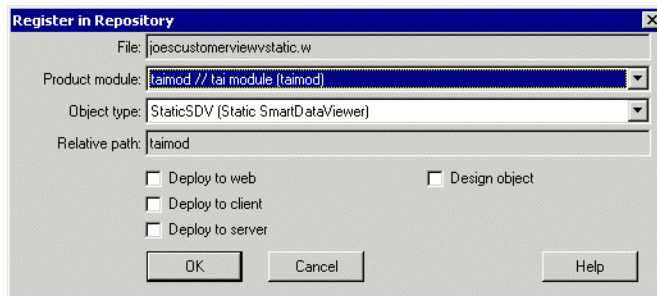
3.9 Adding a file to the Repository

You can use the Register in Repository option to add AppBuilder-created files to the Repository as static objects. For example, you might have an existing Static SmartDataObject™ that you want to add to the Repository so you can build other Repository objects from it.

To add an AppBuilder static file to the Repository as a static object, follow these steps:

- 1 ♦ Choose **File**→**Open File**. If the file is already open, click in its design window to make it the current design object.
- 2 ♦ Choose **File**→**Register in Repository**.

The **Add to Repository** dialog box appears.



- 3 ♦ Select a **Product Module** in which to save the file in the Repository.
- 4 ♦ Select an object type for the static file you are adding.
- 5 ♦ (Optional) Categorize your object. Select one or more of the deployment options or select Design object if this is not intended to be a deployed object. Not selecting an option means the object will be included in all deployments.
- 6 ♦ Choose **OK** to save the static file as a static object in the Repository.

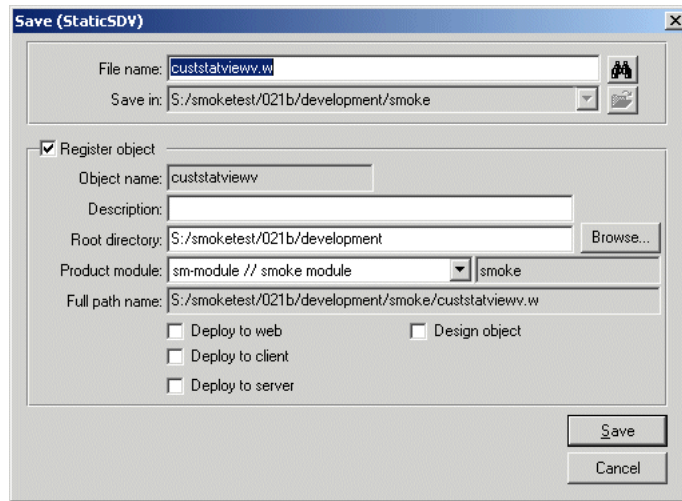
The AppBuilder enables the Register in Repository option for static files that have already been saved to the file system and that are open in the AppBuilder as non-Repository files. Otherwise, the option is not enabled for use.

3.10 Saving a dynamic object as static

When you open a dynamic viewer or a dynamic SDO, the **File** menu enables an additional command: **Save Dynamic Object As Static**. This option allows you to save either a dynamic viewer or dynamic SDO (and no other dynamic object) as a static object file.

- 1 ♦ Select the dynamic viewer or dynamic SDO in the AppBuilder.
- 2 ♦ Select **File**→**Save Dynamic Object as Static**.

The **Save As** dialog appears.



- 3 ♦ Enter a different name for the new static object. You need a new name if you plan to register this static object in the Repository and keep the dynamic object in the Repository.
- 4 ♦ If you want the item registered in the Repository, check Register in Repository.
- 5 ♦ Click **Save**.
- 6 ♦ Select a **Product Module** in which to save the file in the Repository.
- 7 ♦ (Optional) Select one or more of the deployment options or select Design object if this is not intended to be a deployed object. Not selecting an option means the object will be included in all deployments.
- 8 ♦ Choose **OK** to save the static file as a static object in the Repository. All existing dynamic events will be written out as static triggers executed the same way as the dynamic events.

3.10.1 Replacing a dynamic object with a static object

Suppose you decide that a dynamic object in your application should be a static object. Because of the registration in the Repository, you'll need to save the dynamic object as static, give the file a name different from the dynamic version, and replace the dynamic object with the new static object in your containers,

3.10.2 Handling triggers

If the dynamic object contains UI events, this command creates triggers in the new static code and RUN or PUBLISH statements in the trigger and the AppBuilder writes out run or publish code for the specified event.

If the dynamic object has a super procedure with event code, you must move that code manually into the new static code file.

3.11 Editing properties for Repository objects

The Dynamic Property Sheet integrates with other Progress Dynamics tools, allowing properties or attributes to be assigned to or defined for dynamic objects. This tool also addresses the requirement to customize objects. This tool allows customization result codes to be specified.

It functions similarly to an ActiveX property sheet, with a grid-like interface that allows the editing of attributes and their values in one row per attribute. It consists of a non-modal window with a list of allowable attributes or properties that may be edited for a specified object.

If you open a Repository object and then its Dynamic Property sheet in AppBuilder, you are looking at the master attributes for that object. Any changes here affect all instances of the object. When you access the Dynamic Property sheet from the Container Builder for an object instance, you are viewing the properties of only the selected instance.

To edit the **master** properties of a Repository object open in the AppBuilder, follow these steps:

- 1 ♦ Click in the dynamic object design window to make it the current AppBuilder design object.
- 2 ♦ Choose the **Dynamic Properties** icon on the toolbar of the AppBuilder window.

The Dynamic Properties sheet dialog box displays for the design window you selected.

- 3 ♦ Make the necessary changes to the object's properties in the Property Sheet dialog box. (Choose **Help** if you need more information about the options on this dialog box.)
- 4 ♦ Save the object in the AppBuilder.

NOTE: If a static object is registered in the Repository, you can apply attributes using the Dynamic Property sheet. For a viewer, you can only apply attributes to the viewer, and not to items in the viewer.

3.11.1 Property sheet example

Figure 3–12 shows an example of the Static SDO object's Property Sheet window.

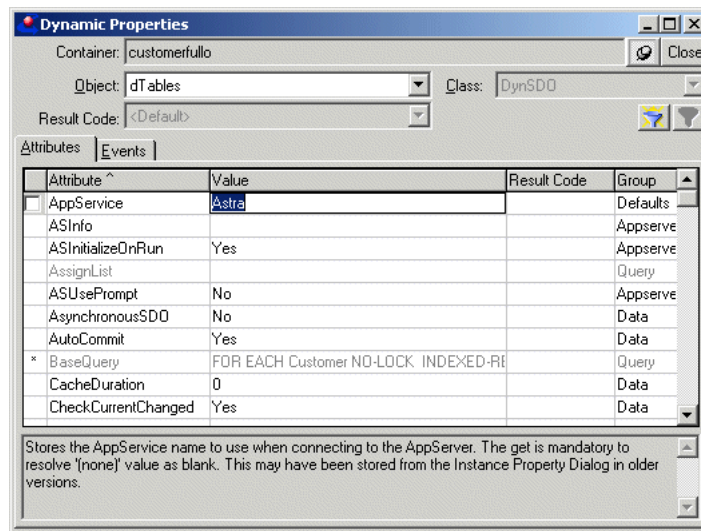


Figure 3–12: Static SDO Dynamic Properties window

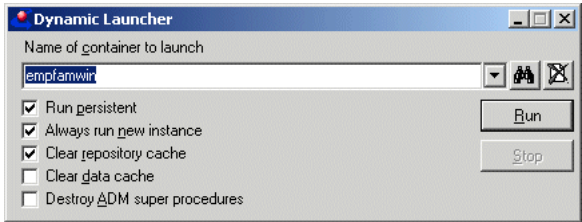
3.12 Running objects

You can run dynamic containers from the Container Builder (open the container and click the Preview Container button on the toolbar). You can run both static and dynamic containers from the AppBuilder (**Compile**→**Run** or click the **Run** button on the toolbar), from the Container Builder, or from the Dynamic Launcher (type the object name and click **Run**).

3.12.1 Running objects with the Dynamic Launcher

To run a dynamic container object from the AppBuilder, follow these steps:

- 1 ♦ Choose **Compile→Dynamic Launcher**. The Dynamic Launcher dialog box appears:



- 2 ♦ Enter the name of the container object to launch, then choose **Run**.

The Dynamic Launcher runs the container object you enter.

NOTE: You can run a static container window using the Dynamic Launcher if the static object exists in the Repository.

The Dynamic Launcher also provides the options shown in [Table 3–12](#).

Table 3–12: Dynamic Launcher options (1 of 2)

Option	Description
Run persistent	When enabled, the selected object runs and stays open while you remain in the development environment.
Always run new instance	When this option is disabled, each time you run a particular object, the current instance of the object will be closed before the object runs again. If the option is enabled, each launch of a particular object results in a new instance of the object.
Clear repository cache	Clears client-side caching of object Repository data to ensure that you are running the latest versions of dynamic objects. The Progress Dynamics Manager procedures retrieve Repository data on the server and send it to the client as requested. Generally this data is cached in the client session so that it does not need to be constantly retrieved.

Table 3–12: Dynamic Launcher options*(2 of 2)*

Option	Description
Clear data cache	Before attempting to run, clear all information in the data cache used by SDOs, dynamic combos, and dynamic lookups.
Destroy ADM super procedures	Check this when you want to refresh the super procedures running in your session.

3.13 Closing Repository objects

To close a design window object in the AppBuilder, choose **File→Close** or **File→Close All** or choose the **X** in the object’s design window. If you have made changes that you have not saved, the AppBuilder prompts you to save the changes before it closes the object.

3.14 Object palette and object template information stored in Repository

The AppBuilder custom files (.cst files) are static text files used to construct the AppBuilder Object Palette and to define a list of new object types based on pre-existing object templates. The template definition is currently used for both static and dynamic objects. However, the Palette currently is of limited use with dynamic objects.

Progress Dynamics stores that information in the Repository. The information from the .cst files is now stored in the new “template” and “palette” classes of Repository objects. This step creates opportunities for greater flexibility and extensibility at design time. If you have been using the .cst files that ship with Progress Dynamics, this change should be transparent. You might notice some previously available custom widgets missing from the pop-up menus of certain objects on the Object Palette.

At startup, the AppBuilder checks two new session parameters, stored as tags in the `icfconfig.xml` file, to decide which templates and palettes to load. Before you can use the Repository data to replace the `.cst` files, you need to regenerate your configuration file or add the following parameters manually:

- **IDETemplate** — Indicates the master template objects to load. It is a string of master template objects. The default list is 'templateContainer,templateSmartObject,templateProcedure,templateWebObject'.
- **IDEPalette** — Indicates the master palette objects to load. It is a string of master palette objects. The default value is 'PaletteDynamics'.

NOTE: If you remove these session parameters from `icfconfig.xml` file, the AppBuilder loads static `.cst` files as in previous versions. This enables you to continue using any customized `.cst` files that you have. You must remove both of the session parameters. You cannot choose to use the Repository for one function and a `.cst` file for the other function.

If you currently use customized `.cst` files, you can re-create your environment by creating new template and palette objects with the Repository Object Maintenance (ROM) tool. The following sections describe how to do this.

NOTE: See the “Customizing the AppBuilder” appendix of the *Progress AppBuilder Developer's Guide* for more information about customized `.cst` files.

3.14.1 Creating and modifying template objects

Always create your own template and palette objects. Do not modify the default template or palette objects. Any changes you make to the default objects might be overwritten by changes Progress makes in future releases.

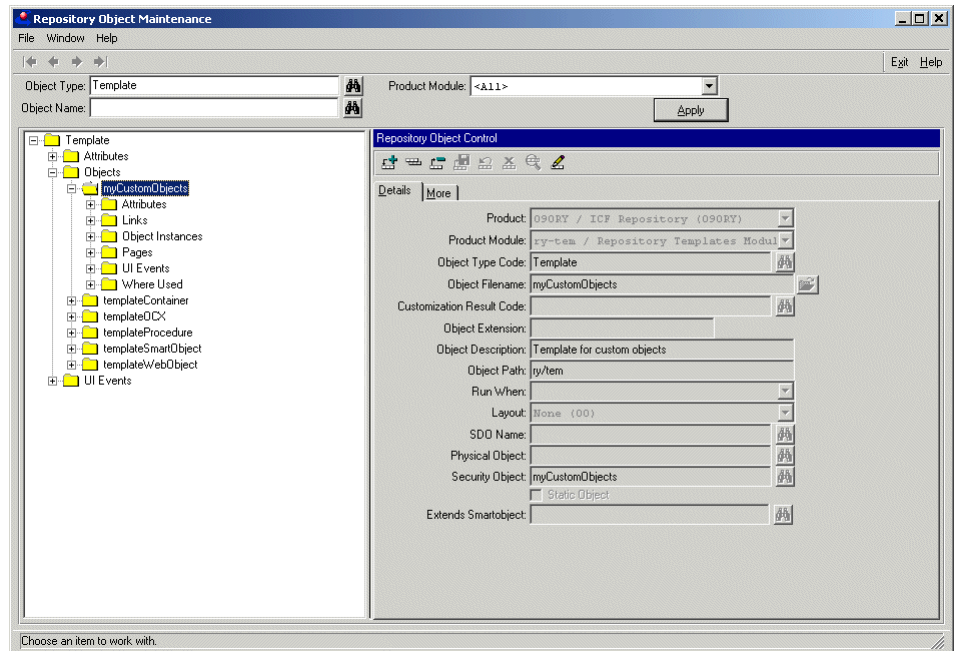
Follow these steps to create a template object with the ROM to add a custom object to the New dialog box:

- 1 ♦ Create a static SmartDataViewer template.
- 2 ♦ Save it as **mySmartViewer.w** in the `adm2/custom` directory and register it in the Repository.
- 3 ♦ Open the ROM and type **template** in the **Object Type** field.
- 4 ♦ Expand the treeview to show the contents of the **Template→Objects** node.

- 5 ♦ Create a new Object using the following values:

Field	Value
Product	MyProduct
Product module	MyProductModule
Object type code	SmartViewer
Object filename	myCustomObjects
Layout	None (00)
Template SmartObject toggle box	Selected

- 6 ♦ Expand the **MyCustomObject** node:



- 7 ♦ Create a new **Object Instance** named **mySmartViewer**. Specify the Object Type as SmartDataViewer and Object Filename as myCustomObjects.

- 8 ♦ Add the following attributes for **mySmartViewer**:
 - **TemplateFile** — adm2/custom/mySmartViewer.w
 - **TemplateGroup** — SmartObject
 - **TemplateLabel** — My&CustomSmartViewer
 - **TemplateOrder** — 7
- 9 ♦ (IMPORTANT) Add the new object (MyCustomObjects) 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 (for example, *,**MyCustomObjects**).
- 10 ♦ Choose **Menu→Use Custom** from the Object Palette's menu. The Use Custom dialog box appears. The buttons on the dialog box are disabled when you are working from the Repository and enabled when you are working from the static .cst files.
- 11 ♦ Choose **OK** to reload the information from the Repository.
- 12 ♦ Choose the **New** icon on the AppBuilder main window. The MyCustomSmartViewer object is available when the New dialog box appears.

3.14.2 Creating and modifying Palette objects

Always create your own template and palette objects. Do not modify the default template or palette objects. Any changes you make to the default objects will be overwritten by changes Progress makes in future releases.

Follow these steps to create a palette object with the ROM to add a custom icon to the Object Palette:

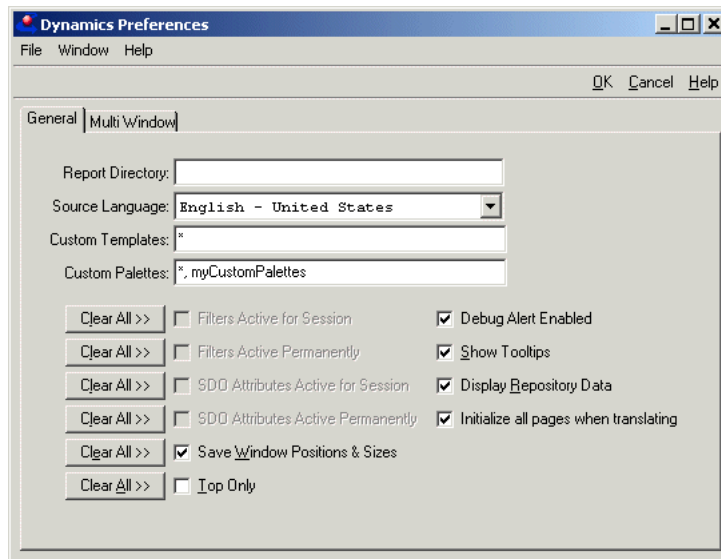
- 1 ♦ Create a SmartDataField drop-down calendar.
- 2 ♦ Save it as **dropdowncal.w** in the adm2/custom directory and register it in the Repository.
- 3 ♦ Open the ROM and type **palette** in the **Object Type** field.
- 4 ♦ Expand the treeview to show the contents of the **Palette→Objects** node.

- 5 ♦ Create a new Object using the following values:

Field	Value
Product	MyProduct
Product module	MyProductModule
Object type code	staticSDF
Object filename	myCustomPalettes
Layout	None (00)

- 6 ♦ Expand the **myCustomPalettes** node.
- 7 ♦ Create a new **Object Instance** named **dropdownncal**.
- 8 ♦ Add the following attributes for **dropdownncal**:
- **PaletteNewTemplate** — adm2/custom/dropdownncal.w
 - **PaletteType** — SmartDataField
 - **PaletteLabel** — Dropdown Calendar
 - **PaletteOrder** — 5
- 9 ♦ Choose **File→Preferences** on the Administration window. The Dynamics Preferences dialog box appears.

- 10 ♦ Type *****, **myCustomPalettes** in the **Custom Palettes** field:



- 11 ♦ Choose **OK**.
- 12 ♦ Choose **Menu**→**Use Custom...** from the Object Palette's menu. The Use Custom dialog box appears.
- 13 ♦ Choose **OK** to reload the information from the Repository.
- 14 ♦ Right-click the **SmartDataField** icon. The new Dropdown Calendar object is available on the pop-up menu.

3.14.3 New and changed attributes

Importing information from the .cst files into the Repository required the creation of new attributes for the template and palette objects. You can also extend widgets when you collect them in a palette, just as you could in a static .cst file. For an example of how this works, look at the standard paletteButtons palette that contains the buCancel, buDone, and buOK buttons. However, this required that some attributes used in the .cst files have their names changed to match attribute names in the Repository.

Attribute name changes

Some of the attribute names used in the .cst files had to be changed to match the Repository. [Table 3–13](#) lists the names of the attributes that have changed.

Table 3–13: Changed attribute names

(1 of 2)

.cst attribute name	Repository attribute name
3-D	THREE-D
CANCEL-BTN	CANCEL-BUTTON
COLOR	DCOLOR
COLUMN-SEARCHING	ALLOW-COLUMN-SEARCHING
DEFAULT-BTN	DEFAULT
ENABLE	ENABLED
FLAT	FLAT-BUTTON
HEIGHT	HEIGHT-CHARS
HEIGHT-P	HEIGHT-PIXELS
IMAGE-DOWN	IMAGE-DOWN-FILE
IMAGE-INSENSITIVE	IMAGE-INSENSITIVE-FILE
INITIAL-VALUE	INITIALVALUE
LOCK-COLUMNS	NUM-LOCKED-COLUMNS
MIN-HEIGHT	MinHeight
MIN-WIDTH	MinWidth
NAME	NameDefault
NO-AUTO-VALIDATE	AUTO-VALIDATE
NO-LABELS	LABELS
NO-TAB-STOP	TAB-STOP

Table 3–13: Changed attribute names (2 of 2)

.cst attribute name	Repository attribute name
ROW-HEIGHT	ROW-HEIGHT-CHARS
SCROLLBAR-H	SCROLLBAR-HORIZONTAL
SCROLLBAR-V	SCROLLBAR-VERTICAL
SHOW-POPUP	ShowPopup
WIDTH	WIDTH-CHARS
WIDTH-P	WIDTH-PIXELS

Template attributes

Template objects have several attributes to store template information. These attributes are defined on the Base and DynamicObject classes in order to extend to all possible templates.

[Table 3–14](#) lists the template attributes.

Table 3–14: Template attributes (1 of 2)

Attribute	Description
templateFile	The relative path and filename of the static object used as the template at design time (Required).
templateGroup	The possible values are Container, SmartObject, Procedure, and WebObject. It defaults to Container (Required).
templateLabel	The label displayed in the New dialog box and the pop-up menu for the Palette icon. It defaults to the instance name.
templateDescription	The description associated with the object. The description in the New dialog box is based on the templateFile.
templateOrder	Used to determine the order of templates in the New dialog box. All templates are ordered within the group. The AppBuilder automatically orders templates by Container, SmartObject, Procedure, and then WebObject.

Table 3–14: Template attributes*(2 of 2)*

Attribute	Description
templateImageFile	Used in design mode for the image that appears in the design window (dynamic objects only). This can also be defined at the class level.
templatePropertySheet	Used for dynamic containers to associate the static property sheet. This allows for future template objects to use other property sheet mechanisms besides the Container Builder (dynamic objects only).
dynamicObject	Logical value to specify that the template is dynamic (required if true).

Palette attributes

Palette objects have several attributes to store palette-specific information. These attributes are defined on the Base, Progress Widget, and DynamicWidget classes in order to extend to all possible palette widget and object types.

[Table 3–15](#) lists the palette attributes.

Table 3–15: Palette attributes*(1 of 3)*

Attribute	Description
paletteisDefault	If Yes, the item is displayed on the palette as an icon. Only one item within a group can be the default. (Required for at least one instance.)
paletteType	The type of widget supported in the AppBuilder. If creating a custom widget, the name must be spelled exactly as the widget, such as, Button, Radio-set, or Editor (required).
paletteLabel	Label used in the drop-down menu. If not specified, the Instance name is used.
paletteTooltip	ToolTip used for the palette icon. The default is the paletteLabel.
paletteTriggerEvent	For widgets in static objects, defines the Event associated with the widget (pipe delimited).
paletteTriggerCode	Defines the code for the triggerEvent (pipe delimited).

Table 3–15: Palette attributes*(2 of 3)*

Attribute	Description
paletteImageUp	The up image to use in the palette. (Required if paletteisDefault is true.)
paletteImageDown	The down image to use in the palette. (Required if paletteisDefault is true.)
paletteNewTemplate	Used for new SmartObjects to indicate the rendering object file. (Required for SmartObjects if paletteDirectoryList, paletteFilter, and paletteTitle are not specified.)
paletteRenderer	<p>Used to inform AppBuilder of the static file to render when dropping a smart object from the palette onto a container. This attribute is akin to the USE option in the .cst file and is specified on an instance of a 'palette' object.</p> <p>The framework previously used the value of attribute PalleteNewTemplate, which should only be used when creating a new object from the palette. If this new attribute is not specified, the framework will use the PaletteNewTemplate value as before.</p>
paletteDirectoryList	Used in the Choose Object dialog box to specify a directory. This must be specified if the next two parameters are also specified.
paletteFilter	Used in the Choose Object dialog box to filter files.
paletteTitle	Choose Object dialog box title.
paletteControl	To be defined for the OCX control object only to specify the control codes.
paletteDBConnect	If Yes, specifies the DB must be connected before using this item.

Table 3–15: Palette attributes*(3 of 3)*

Attribute	Description
paletteEditOnDrop	Indicates whether to automatically display the property sheet when the object is dropped onto the appBuilder design window.
paletteOrder	<p>The order of the palette item in the palette. If no order is specified, the icon is added to the end.</p> <p>If the item is displayed on the palette directly (paletteisDefault is true), this integer value indicates the actual position of the icon in the palette relative to the basic widget types. Value 1 would refer to the position to the right of the OCX icon. You must specify a valid position.</p>

3.15 Summary

This chapter has provided an overview of the additional tools provided through the AppBuilder if you are running Progress Dynamics. As noted in the text, these tools are described in more detail in later chapters of this guide and in other documentation.

Preparing to Build Application Objects

The biggest single step in actually creating your Progress Dynamics-based application is generating your base application objects. The Object Generator creates SmartDataObjects (SDOs) to manage queries and to stream data between server and client, logic procedures to provide a place to enter table-specific business logic, Data Fields to provide object capabilities to database fields, dynamic Browsers to display that data in client applications, and dynamic Viewers to display and update individual records. The application you get from the Object Generator will not be complete by any means, but it will give you a big head start in creating at least the data maintenance windows for your application tables. The Object Generator is described in detail in [Chapter 5, “Using the Object Generator.”](#)

Before you get to that step, however, you must do some setup work to prepare your environment and to define a structure for your application, as explained in the sections of this chapter:

- [Obtaining and assigning site numbers](#)
- [Defining products and product modules](#)
- [Creating Repository data for tables and fields](#)

4.1 Obtaining and assigning site numbers

The first step you should take before starting serious application development is to obtain a unique site number for each copy of your development database. When you get to the point of deploying your application to customer sites, you will also want to be able to assign a unique site number to each distinct deployment site. The reason for this is to assure that the special Object ID keys, used by the Repository database (and by application databases designed using the same principle), are unique even between different copies of that database. Now take a moment to examine how the Object IDs are constructed and what purpose their uniqueness serves.

In essence, the Progress Dynamics Repository database uses a decimal Object ID as the unique key field for every table in the Repository. As discussed in the “Object IDs and Site Numbers” section in [Chapter 2, “Database Design Principles in Progress Dynamics,”](#) you might want to make use of this same technique in your own application database if you are designing a new database or are in a position to make changes to your existing database schema. The site number becomes part of every Object ID key value, such that every Object ID value for every Progress Dynamics Repository or application database anywhere has a value that is guaranteed to be absolutely unique. This makes it possible for you to deploy dynamic application components that are defined strictly as data, without concern that the key values from one database that are used to associate related records can have conflicting values with existing records in the database those objects are deployed to. The same holds for any application data that you might need to deploy from one site to another.

4.1.1 Obtaining a site number

To allow everyone in the Progress community who is using Progress Dynamics to build and deploy an application, Progress offers the ability to obtain a unique site number or range of site numbers. The <http://www.progress.com/dynamics/sitenumber> Web site supports a Web application to register framework users and sites and assign site numbers. This application maintains a central database where registered user sites and their site numbers are stored.

Figure 4–1 shows the entry screen for the site number application.

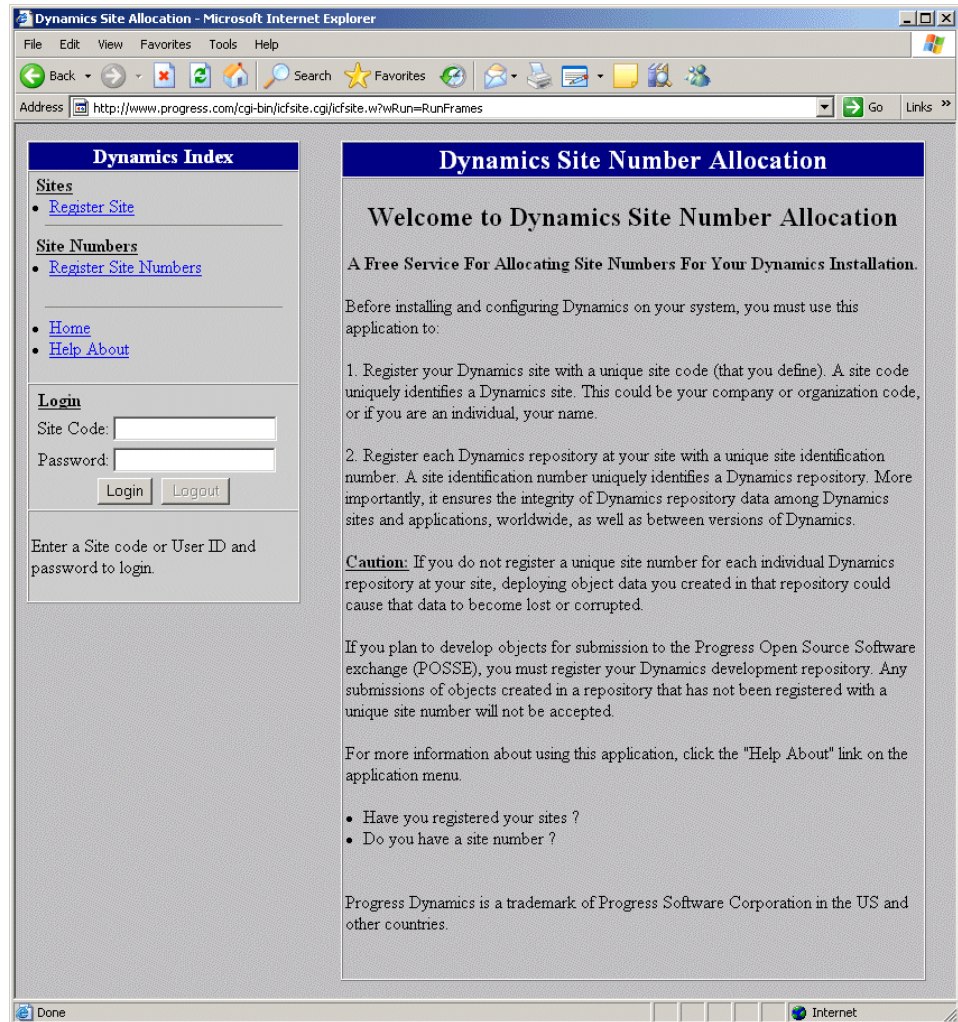


Figure 4–1: Progress Dynamics Site Number Allocation web site

Follow these steps to obtain a site number:

- 1 ♦ To register your site with the application, select the **Register Site** link in the left frame.

The Site Creation page appears:

The screenshot shows a Microsoft Internet Explorer window titled "Dynamics Site Allocation - Microsoft Internet Explorer". The address bar displays "http://www.progress.com/cgi-bin/icf/site.cgi/icf/site.w?wRun=RunFrames". The page is divided into two main sections: "Dynamics Index" on the left and "Dynamics Site Control" on the right.

Dynamics Index

- Sites
 - [Register Site](#)
- Site Numbers
 - [Register Site Numbers](#)
- [Home](#)
- [Help About](#)

Login

Site Code:
Password:

Enter a Site code or User ID and password to login.

Dynamics Site Control

Register Site

Site Code*:
E-mail address:
New Password*:
Confirm Password*:
Description:

• Enter your site information and press the Register button.
• Required fields are marked with a red asterisk (*).

- 2 ♦ Enter a site code, a unique character string (without spaces) to identify your site in a way that is meaningful to you. A one-word abbreviation of your organization name, or your own name if you are assigning the site number for your own use, is appropriate. The application informs you if the **Site Code** you choose already exists in the database.
- 3 ♦ Enter the e-mail address where you can be reached. Part of the purpose of registering your sites is to enable Progress to contact you regarding a notification for all active Progress Dynamics developers. Your e-mail address will not be public information.
- 4 ♦ Enter a password for your site code. The password prevents anyone else in the community from modifying the information or site numbers associated with your site code.

- 5 ♦ Enter a site description, typically a longer identifier for your organization or your full name.
- 6 ♦ Choose **Register** to enter your information into the site number database.

Now that you have registered your site, follow these steps to assign one or more site numbers to it:

- 1 ♦ From the left Index frame, select the **Register Site Numbers** link:

- 2 ♦ Enter your **site code** in the Owning Site Code field. This is the site code that you entered earlier to identify your site.

- 3 ♦ If your site code is part of a larger organization that is also registered in the site number application, indicate this by filling in the Allocated Site Code with the **site code** for the controlling organization. This field helps organize sites in the application.
- 4 ♦ If you have already been using a site number in your Progress Dynamics Repository database and you want to keep using it, enter that number for the site number. Otherwise leave this field blank and you will be assigned the next available starting site number. If your site number is already being used by another registered site, you will be notified and you will have to select another site number.

NOTE: There is no inherent problem in having Object ID records in your database with different site numbers. This just means that the fractional part of those Object IDs will not all be the same. Your Repository and your application will still function properly. An issue will arise only if you need to deploy object definitions or other data from your copy of the Repository database to another Repository database that could possibly have the same site number (such as a deployment to an application site), or if you deploy data from your application database to another database that could be using the same site number. To avoid these issues, you must get your site number assigned correctly before you start doing work that could actually lead to data being deployed.

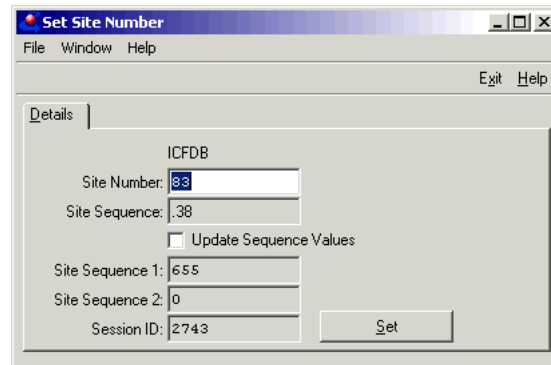
- 5 ♦ Enter the site count, the number of site numbers you want to reserve. You will want a distinct site number for each different development Repository database within your organization, and for each deployed set of databases (the Repository plus your application database) that you anticipate. You can allocate up to 100 site numbers to a site code. The total possible number of site numbers is limited only by the nine-digit format of the fractional portion of the Object ID field. This is more than adequate, but it protects against anyone allocating millions of site numbers inappropriately. It also allows large blocks of site numbers to be allocated when they need to be matched up with product registration numbers or some other key.
- 6 ♦ Choose **Register** to save your site number information. A list of your assigned site numbers displays at the bottom of the page.

4.1.2 Assigning a site number to your database

Once you have obtained a site number for your Repository database, follow these steps to enter it in your database:

- 1 ♦ From the AppBuilder main window, select **Build→Set Site Number**. (You can also select this option from the Progress Dynamics Administration window's System menu.)

The **Set Site Number** dialog box appears:



- 2 ♦ Enter your site number in the field provided for the Progress Dynamics Repository Database (**ICFDB**).

The site number is stored as the fractional part of each Object ID, to the right of the decimal point or comma. To ensure that trailing zeroes in the site number do not lose their significance (for example, so that site number 70 and site number 700 become distinct values, which would not be the case for .70 and .700), the site number is reversed before it is applied to the key. The **Site Sequence** is the actual decimal fraction that becomes part of every Object ID key.

The whole number portion of the Object ID is constructed from two database sequences, forming the high-order and low-order portions of a large integer value, to assure support for a sufficiently large number. The number must accommodate the largest possible number of records in your Repository database, since every record in every table will have a unique ID number. You can adjust the starting values for those sequences in the **Set Site Number** dialog box. **Site Sequence 1** is the low-order portion of the Object ID key, and **Site Sequence 2** is the high-order portion. When the low-order **Site Sequence 1** reaches its maximum value and rolls over back to 1, the trigger procedure that assigns **Object IDs** automatically increments the high-order **Site Sequence 2**.

NOTE: Progress Dynamics does not automatically assign object IDs to records in an application database. You have to implement your own logic to do this based on the principles that Progress Dynamics uses.

There is also a unique Session ID assigned to each database for each session. This is used to form a key for context data that might be stored for the session by the Session Manager. This Session ID number is also displayed, and you can modify it if necessary (normally it will not be) by selecting the Update Sequence Values toggle box.

4.2 Defining products and product modules

Progress Dynamics lets you segment your application into **Product Modules** so that you can organize your application objects based on the part of the overall application in which they are used. This section describes how to set up and use Product Modules. Product Modules can represent any type of application organization you like. For example, they could correspond to different products and sub-products that you sell and deploy independently of one another, or they could be some other type of hierarchical organization that is meaningful to you. Setting up some sort of organizational structure helps you to:

- Keep track of where your application components are located.
- Determine what you need to deploy with your application.
- Distribute and manage development work within your organization.
- Locate objects quickly when assembling and maintaining parts of your application.

This first release of Progress Dynamics uses a strict two-level structure to organize components. The top level is called “Products” and the second level “Product Modules.”

Every object you create in the Repository, whether it is entirely dynamic or whether it is a procedural object registered in the Repository, must have a Product Module assigned to it. Some kinds of objects might be very general-purpose and not really part of any specific application Module (for example, window layouts that you create in the Layout Builder). For this reason, you might want to set up one or more modules that identify those general-purpose objects, such as a “Design” module.

4.2.1 Assigning product and product module names

There is a convention whereby the framework assigns procedural objects you create to a directory structure corresponding to the Product Module names. Within the Products for the Repository itself, a hyphen within a Module name indicates the concatenation of a Product and Product Module. For example, there is an RY Product for objects that support Repository maintenance (the SDOs, dynamic windows and Browsers, Viewers, and so forth that make up the maintenance windows for the utilities under the Administration and Development menus, for example).

Within the RY Product, there are a number of different Modules for different types of objects, for example, ry-app for AppServer-related procedures, ry-obj for general objects such as SDOs, ry-inc for include files, ry-uib for static container procedures, etc. This particular organization is done more by the type of object than its purpose, but any organization will do. The significance of the name is that procedures in the ry-app Module are found in the ry\app directory relative to the top-level src directory for the framework. If this kind of organization is appropriate for you, then you can give your Product Modules names in this same way. In any case, you will specify the actual relative pathname for objects when you define the Product Module, as described below.

NOTE: The relative pathname is stored in the object_path field in the ryc_smartobject Repository table. There is no equivalent for dynamic objects.

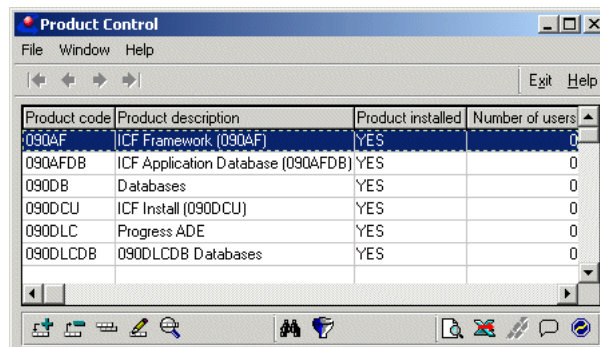
In almost every screen in the Repository maintenance tools, you can select the **Product Module** from a drop-down list as a way of filtering objects when you are searching for something. Keep this in mind as you organize your application. Between the subdirectory structure for procedural objects, and the drop-down filtering for all kinds of objects, using **Product Modules** can help you locate and keep track of objects more efficiently.

Another important point to keep in mind is that you can define security structures, such as access restrictions on an action, data range or field, for a specific **Product Module** or for all **Product Modules**. Thus you can easily restrict access, for example, to all Add functions within a product module for a particular user. This can be another effective use of the **Product Module** to organize your application.

To define Products and Product Modules, follow these steps:

- 1 ♦ From the Administration window, select **Application→Product Control**.

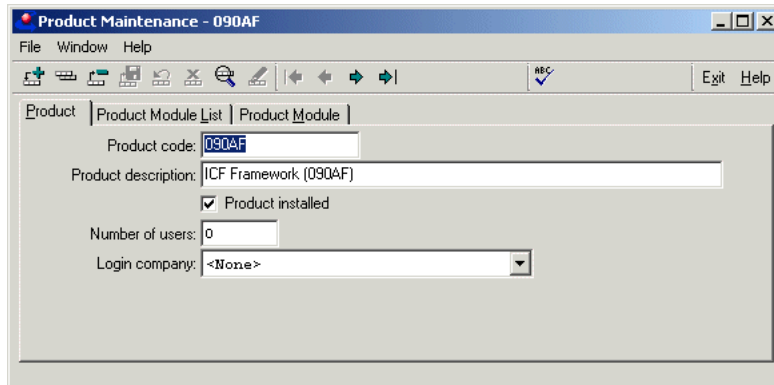
The Product Control window appears with a list of all existing **Products**. Initially, the products defined are for the Repository itself:



The screenshot shows a window titled "Product Control" with a menu bar (File, Window, Help) and a toolbar. Below the toolbar is a table with four columns: Product code, Product description, Product installed, and Number of users. The table contains six rows of data, all of which are installed (YES) and have zero users.

Product code	Product description	Product installed	Number of users
090AF	ICF Framework (090AF)	YES	0
090AFDB	ICF Application Database (090AFDB)	YES	0
090DB	Databases	YES	0
090DCU	ICF Install (090DCU)	YES	0
090DLC	Progress ADE	YES	0
090DLCDB	090DLCDB Databases	YES	0

- 2 ♦ Choose the **Add** button in the bottom toolbar to bring up the Product Maintenance window so that you can create a new Product:



- 3 ♦ In the Product Code field, enter a **one-word name** for the product.
- 4 ♦ In the Product Description field, enter a **longer description** of the Product.

The framework does not currently use the Product Installed and Number of Users fields. The intent of the Product Installed field is to adjust what application components such as menu structures are displayed to a user, depending on whether that product is installed at that user site. Likewise, the intent of the **Number of Users** field is to apply some sort of licensing restrictions to a site; counting users at this point is your responsibility if you want to make use of this field.

You should set Product Installed to **Yes** and leave the Number of Users at **0**. However, these settings are stored as fields in the `gsc_product` table in the Repository database, and if you would like to make use of these fields as a control mechanism for your application, set these fields to values that are meaningful for you.

- 5 ♦ Choose **Save**.

4.2.2 Assigning login companies to products

The Login Company is a concept that is described in more detail in [Chapter 13, “Defining Progress Dynamics Application Security.”](#) You can define different organizational entities that make use of an application or products in an application. These entities might actually be Companies in some sense of the word or they might be any other kind of organization (Departments or Roles are alternative terms). When a user logs in to an application, one of the fields the login dialog box prompts for is the Login Company. When you create your own application, you can rename the label for this field to something more meaningful to you, or you can eliminate it altogether.

The user can either log on in the context of a particular Company, or can select **<All>**, meaning that the logon is for no particular Company. The **Product Maintenance** tab lets you associate a Product with a particular **Login Company**. The **Login Company** field is just for reference.

When you get to the point of defining security restrictions, you can associate a restriction with a user (meaning “this user cannot access this application function”), with a login company (meaning “users logged in under this company cannot access this function”), or with a combination of the two (meaning “this particular user cannot access this function when logged in under this particular company”). You can define Login Companies using the Login Company Control option under the Administration Security menu.

You might want to associate some Products with a Login Company when you come to design your security structures. Initially, you will probably want to leave the Login Company field in the Product Maintenance window at the default setting of **<All>**.

4.2.3 Creating product modules

After you have saved your new Product, follow these steps to add Modules to your Product:

- 1 ♦ Select the **Product Module** tab:

The screenshot shows a window titled "Product Maintenance - 090AF" with a menu bar (File, Window, Help) and a toolbar. The "Product Module" tab is selected. The form contains the following fields and controls:

- Product module code:** Text field containing "af-aaa".
- Product module description:** Text field containing "ICF Root Directory".
- Product module installed:** A checked checkbox.
- Number of users:** Text field containing "0".
- Relative path:** Empty text field.

Buttons for "REC" (with a checkmark), "Exit", and "Help" are located in the top right corner of the window.

- 2 ♦ Choose the **Add** button, then fill in the fields.
- 3 ♦ In the Relative Path field, specify the **relative pathname** for procedural objects saved in this Product Module. This string should be a pathname relative to the session working directory. Do not use a slash at the beginning or at the end. Use forward slashes to delimit subdirectories.
- 4 ♦ Save the Product Module definition.
- 5 ♦ Repeat [Step 2](#) through [Step 4](#) to add as many Modules as you need.
- 6 ♦ To view a summary of all the Product Modules for the current Product, select the **Product Module List** tab.

4.3 Creating Repository data for tables and fields

The next task you must perform as you prepare your environment is to create data in the Repository for all the “entities” (tables and fields) in your application database. This information is used to let tools such as the Object Generator use the structure of your data, your naming conventions, and other information that makes it more successful in creating objects automatically.

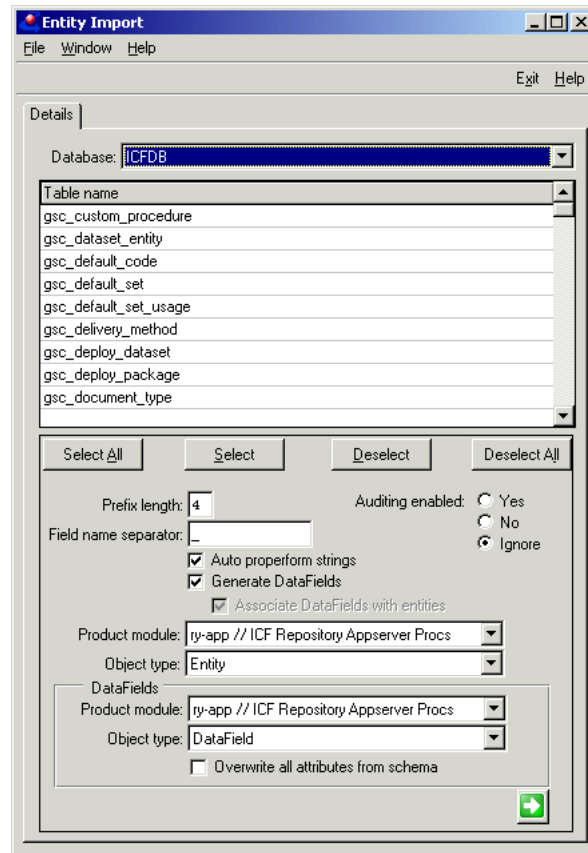
You can specify the key fields that form a unique key for a table, so that Comments and Auditing records can be created for any database record. You can also identify what fields should be made part of objects such as the dynamic Browsers and Viewers the Object Generator creates.

Entity Import is the process of reading through the database schema information for all the tables in your application database, or for selected tables, and generating the starting values for this kind of information in the Repository table `gsc_entity_mnemonic`. You can then go into the Entity Maintenance utility to modify values for specific tables and fields as needed.

4.3.1 Entity import

Follow these steps to go through the Entity Import process:

- 1 ♦ From the Administration window, select **System**→**Entity Import**. The Entity Import dialog box appears:



- 2 ♦ Select the application database you want to import entities from.
- 3 ♦ Select one or more tables from that database.

As with any multiple-select browser, you can use the **CTRL** key together with your mouse to add individual entries to the list. The tables are highlighted when you select them. To select a series of entries, click on the first one, then press **SHIFT** and click on the last one. Use **Select All** to select all rows, and **Deselect All** to undo that action.

- 4 ♦ Enter the **prefix length** in the Prefix Length field.

Chapter 2, “Database Design Principles in Progress Dynamics,” discusses the prefix the Progress Dynamics Repository uses—a naming convention involving a four-character prefix for all table names. This prefix consists of two letters to identify the organizational structure (representing the **Product**), a letter to indicate the relative volatility of the data (C for constant versus T for transactional, for example) and an underscore. The framework uses its understanding of this naming convention in various ways. For example, the Object ID field for a table is always given a name consisting of the table name, without its prefix, plus the suffix `_obj`. So the Object ID field for the `gsc_object_type` table, for example, is named `object_type_obj`. You can use this information to identify potential join fields, descriptive fields for a table, and so forth.

The Prefix Length field in this dialog box tells the framework what, if any, standard prefix you use for your table names. If you design a new database and use the Progress Dynamics naming conventions, then you should enter a prefix length of 4. If you use no such standard prefix, then enter 0. If you use some other prefix length, enter its number.

- 5 ♦ Enter your **field name separator**, if there is one., in the Field Name Separator field.

The Repository database has a naming convention of creating table and field names that are meaningful strings of words, such that the name is self-explanatory insofar as this is possible in a few characters. The standard separator between words is an underscore. Reasonable choices are underscore, hyphen, or the word “upper” to indicate that words are separated using mixed case, as in “CustomerName”. If you use no consistent separator in your database, leave blank.

NOTE: Keep in mind that for new database design, you are strongly encouraged **not** to use hyphens in table and field names, since this character is not permitted by SQL databases, which will limit your ability to use DataServers for your database or to use third-party tools that are SQL-based to access your data. Other languages such as Java also do not recognize hyphens as valid name delimiters, which might affect your ability to access your data from other tools and non-Progress products. You will notice in the current screen shot that the field does not yet allow you to enter “upper” in the Import window. This will be addressed in a future release.

- 6 ♦ Specify whether you want to use **Auto Properform Strings**. Proper-forming refers to doing certain kinds of automatic data formatting.

- 7 ♦ Specify whether to generate DataFields.

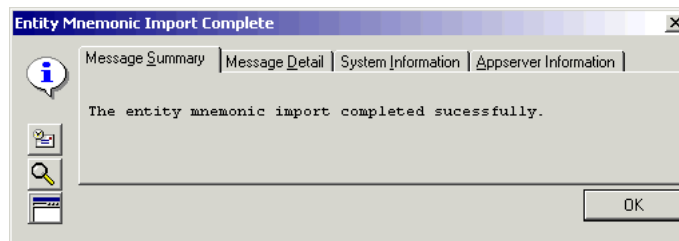
NOTE: Generating DataFields the first time you import a table means you will automatically have lists of fields to choose from when configuring what fields to display in browsers and viewers. When *regenerating* entity information, checking this will replace all your existing data field information.

- 8 ♦ Specify whether you want to enable, disable, or ignore auditing.

The framework can maintain an audit trail for records in any table in your database. There is a flag in the Repository to indicate whether auditing is enabled for each table. The radio set here in the Entity Import window determines the initial value of that flag. If you select **Yes**, then auditing will initially be enabled for every table you have selected to import. If you select **No**, then auditing will initially be disabled. If you select **Ignore**, then there will be no specific setting. In any case, you can set or reset the Auditing flag for individual tables in the Entity Maintenance utility, described in the next section.

CAUTION: Do not enable auditing if you do not specifically require it. It adds a certain amount of overhead to every update operation for a table. Auditing also requires you to write special trigger logic. Therefore, you should set this auditing option to **No** or **ignore**.

- 9 ♦ In the DataFields frame, specify the **Product Module** and **Object Type** where DataFields should be generated.
- 10 ♦ Choose **Import**. When the process is complete, the following confirmation alert box appears:



- 11 ♦ Select a **product module** in the Product Module field.

You must designate a product module for all of the entities you select. For this reason, it makes sense to do the Entity Import module by module, grouping tables into the product modules they support (or primarily support as there might be overlap). If there is no meaningful relation between tables and specific product modules, then you could create a special Product Module just for database entities, or whatever makes sense for your situation.

4.3.2 Entity record fields

It is worth understanding a bit more about exactly what gets assigned when you perform the Entity Import. The following sections describe some of the fields in the Entity record for each table that is assigned values that might not be completely obvious.

Entity short description

The Entity Short Description is formed from the table name without the prefix (based on prefix length if any) and with the field separator (if any) applied. This description is used as a substitute for the table name in messages, to make the table name look more like part of a natural language phrase. How effective this description is depends on the extent to which the table name adheres to the Progress Dynamics convention of making table and field names a series of meaningful words with a standard separator.

You can look at the schema to see exact Repository field names. For the most part, you do not have to concern yourself with the actual names, unless you are writing additional utilities to use or set those fields.

Entity description field

Progress Dynamics supports the notion of having a field in a table that is designated as the Description field for that table. This field provides a user with a meaningful description of the record, as opposed to a key value that can be an arbitrary numeric value of no significance to the user. In general, the framework tries to assist you in joining a primary table to one or more other tables. Then you can include this Description field in the display list for the primary table, in place of (or possibly in addition to) the key field on which the code might depend as a join field and as a unique identifier. The proper automatic identification of this field depends on your schema conforming to the Progress Dynamics naming conventions. If this is not the case, you can set this field for each table in Entity Maintenance.

The naming convention looks for a key field of data type Character, whose name begins with the table name without its prefix, plus one of the following strings:

- `_description`
- `_desc`
- `_name`
- `_short_name`

Failing this, it looks for a field name without the table name, but ending in one of these four strings.

Entity narration

Entity Narration is a description field for the table. It is initialized from the table description in the schema, if any.

Entity object field

The Import tries to determine whether there is a field that can serve as the Object ID field for the table. If there is a field that adheres to the standard Progress Dynamics naming convention for Object IDs (table name minus the prefix plus `_obj`), then this is set as the Object Field. The field must be a numeric field with a unique index, adhering to the format of the `o_obj` domain in the Progress Dynamics *ERwin* template, that is, with nine decimal places.

If this field exists, then an additional Table Has Object Field field is set to **Yes**; otherwise it is set to **No**. You will have the opportunity to define these fields table by table in the Entity Maintenance, if the default settings do not work for your database. However, the field must be a numeric field that has unique values for a given table.

Entity key field

For databases that conform to the naming conventions used in the Progress Dynamics Repository, this field is assigned the name of the first Character field in a unique single-field index, matching the table name minus prefix plus one of the following suffixes:

- `_code`
- `_reference`
- `_type`
- `_tla`

- `_number`
- `_short_desc`

Failing this, it takes a Character field name not matching the table name but matching one of these suffixes; failing that it takes a Decimal or Integer field matching one of the suffixes.

For other databases that do not have such a coded, specially named character field, the import utility uses the **Entity Key Field** to store the names of the fields that make up a unique index for the table. These can be fields of any data type. This field or field list is used by parts of the framework that associate a special record such as a Comment record or Auditing record with a table name and key value. Providing a value for this **Key Field** makes it possible for any database, even one with no actual Object ID field, to take advantage of these features.

Display fields

There is a separate **Entity Display Field** table where a record is stored for each field in each table imported. The fields in this table are populated with data from the database schema, including the field name, label, format, and display order. A record is created for every field in each imported table, except Object ID fields which are identified using the `_obj` naming convention, with the expectation that the Object ID field is not displayed in a standard User Interface.

4.3.3 Entity maintenance

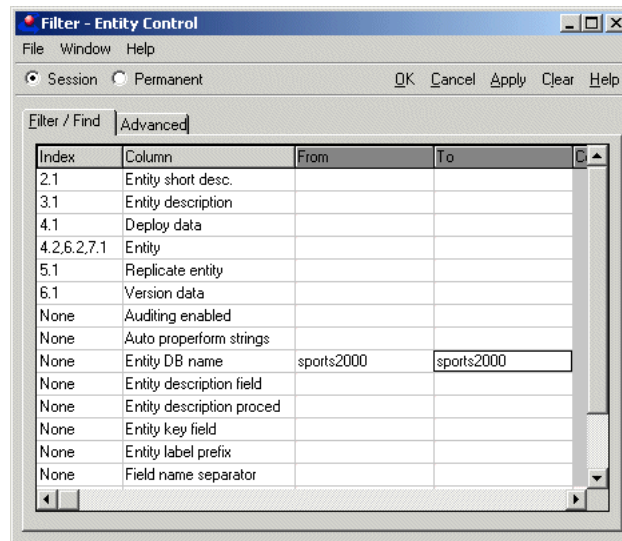
Once you have imported default data into the entity tables from the database schema, you can customize that data for each table in your database as needed. You might want to do this to change the list of fields to be placed into the dynamic Browser and Viewer that you can generate automatically, for example, so that not all fields are displayed for a large table, or so that fields containing large amounts of data needing special display formats are not put into these default visual objects. In addition, you can modify any of the defaults assigned for the **Key Field**, **Object Field**, and other entity values where the Import utility cannot make the correct decisions for your database.

To customize the default data for one or more tables in your database, follow these steps:

- 1 ♦ From the Progress Dynamics Administration window, select **System→Entity Control**.

The Entity Control window displays a list of all the tables known to the Repository including the tables in the Repository database itself. If you want to filter Repository tables out, follow these steps:

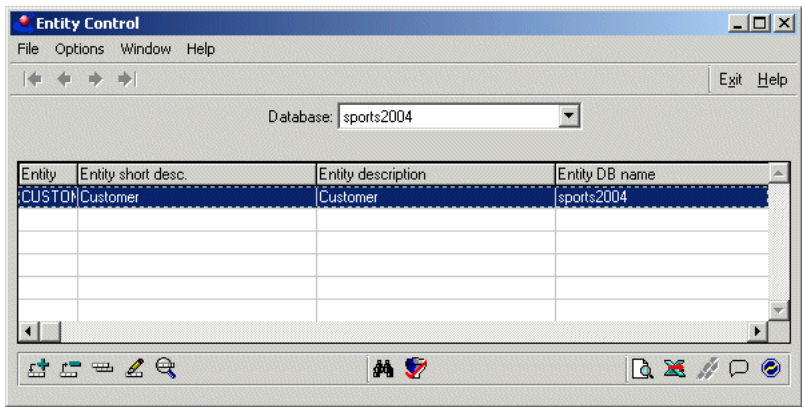
- a) Choose **Filter**. The Filter window appears:



- b) Enter the **name of your application database** under Entity Dbname.
- c) Select **Permanent** so that this filter is saved permanently for you in the Repository.
- d) Choose **OK**.

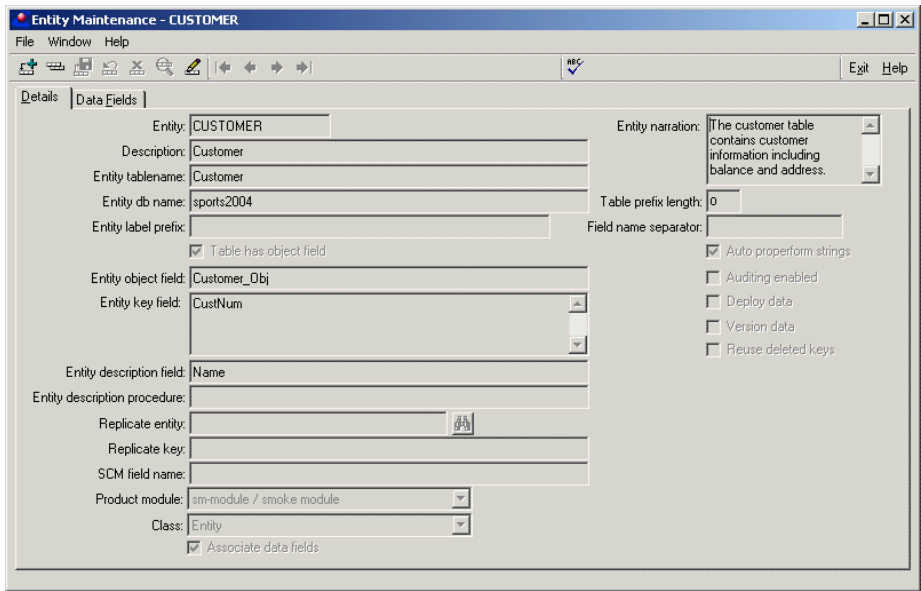
NOTE: In this case, a warning message displays because your filter is on a non-indexed field. This is probably all right because the total number of records in the Entity table (one for each database table) will not be huge. This is an example of using the Filter that is a standard part of every browse window in the framework tools. It is also an example of how user preferences such as filters, window positions and sizes, etc., are saved in the Repository for each user. See [Chapter 8, “Using the Progress Dynamics Container Builder,”](#) for examples of some of the standard Progress Dynamics toolbars that have buttons supporting functions such as this.

Now the Entity Control shows only the tables in your database:



(Note that the Filter button in the bottom toolbar now has a check mark in it, to remind you that you applied a filter.)

- 2 ♦ Select an entry in the browser and either double-click on it or select **Update**. The Entity Maintenance window appears. Use this window to change settings for individual tables, and assign the display list for each table where the default is not appropriate:



Many of the fields in the Entity maintenance window are the same as those in the Entity Import utility. Here you can check to see what values the Import utility has assigned, modify them if they are not appropriate, or fill them in if the Import utility did nothing at all with the field. The following sections describe the use of some of the fields not already discussed.

4.3.4 Entity mnemonic

The *Entity Mnemonic* is simply the database dump name for the table. This name must be unique across all databases used together in an application. The framework uses this as a key to identify the table in a number of general purpose operations. For example, the framework combines the Object ID Field (or if there is none, the Key Field) and the Entity Mnemonic to associate Comments and Audit records with individual records from any application database table. It also uses this string as the first part of all automatically generated names for objects.

For example, for the Customer table shown above, the Object Generator creates a SmartDataObject™ named `customerfull0`, a dynamic Browser called `customerfullb`, and a dynamic Viewer called `customerviewv`. If you are designing a new database or are otherwise in a position to assign dump names for your tables, you might want to create a naming convention for them with this in mind. The standard naming convention for Progress Dynamics uses a five-letter dump name combined with standard abbreviations for object types, resulting in ten-character object names, a manageable size. The framework generates the value for this field directly from the dump name, and you cannot modify the value.

Entity Mnemonic Short Description

The Entity Mnemonic Short Description field contains a short reference used in Lookups for the table and in other places where it is useful to display the table name as part of a longer string, such as a message. The framework derives the default value from the table name, minus the prefix if any, and with the designated separator character (or “upper” for separating based on mixed case) replaced by a space. For example, a table name of `customer_header` with a separator of underscore, or `CustomerHeader` with a separator of “upper”, both become “Customer Header”.

Entity Tablename

The **Entity Tablename** field contains the full table name, including any prefix.

Entity Dbname

The **Entity Dbname** field contains the logical database name for the table.

Entity Mnemonic Label prefix

The **Entity Mnemonic Label Prefix** field contains an optional prefix the framework uses to selectively replace the first word of screen labels for fields in the associated table. Progress Dynamics primarily uses this prefix for tables in the framework Repository to provide more meaningful labels. For example, a table `gsm_region` in a specific application can be used for suburbs and have a label prefix of **suburb** rather than **region**. The label prefix actually replaces the first word in the field label defined on the database. Where the table has more than one meaning, for example a `gsm_person` can be a member, a contact, a practitioner, and you can define the label elsewhere.

Entity Object Field

If there is a standard Progress Dynamics Object ID field, then the framework sets the value of the Entity Object Field to that field name, and sets the Table Has Object Field flag. Progress Dynamics uses this information, among other things, to allow the Object Generator to follow joins when creating SDOs to include descriptive fields from related tables.

Entity Key Field

If there is a Character field holding a meaningful unique code value for the table, the Entity Key Field is assigned that field name. Otherwise, you should assign the name or comma-separated list of names of fields in a unique index that can identify records in the table.

Entity Description Field field and Entity Description Procedure

The Description Field is discussed in the previous section on Entity Import. The Entity Description Procedure field contains the name of a procedure to run in order to work out the description of the entity. This field is only required in the event that the entity description spans multiple fields (potentially from related entities). If this field is specified, it overrides the entity description field. You must include a relative path to the procedure. The procedure you write must take the object ID as an input parameter, and output a single character string containing the object description.

There is a call in the General Manager, `getEntityDescription`, that you can use to return the description for any record in any table, given the Object ID and table name mnemonic, if the Description Field or Description Procedure is defined.

Replicate Entity field and Replicate Key

The Replicate Entity and Replicate Key fields correspond to the user-defined properties ReplicateFLA and ReplicateKey in the ERwin Progress Dynamics template, and are initialized accordingly for databases forward engineered from ERwin. ReplicateFLA should indicate the unique table code (FLA or *five-letter-acronym*) of the primary table being versioned. The Replicate Key is the field name in the table used to join to the primary table being versioned, for example, the Object ID field. They are not actively used yet in the entity table, but are there to capture the ERwin settings. If the corresponding ERwin user-defined properties are set, then appropriate replication trigger code is generated for the table.

SCM Field Name

For source code management, only set the SCM Field Name on the primary table being versioned. It is the unique field for the data that is also used as the object name in the SCM Tool. In Progress Dynamics, this is only set on the ryc_smartobject table and would have a value of object_filename. Setting it in entity maintenance does not do anything. Like the Replicate fields, the setting in the entity table can only reflect what has already been set in ERwin, and without the ERwin setting, the trigger code is not generated to support the use of the field. So generally the use of this field, as well as the UDP (user-defined property), is not appropriate or meaningful for application databases.

Entity Narration

Progress Dynamics gets the Entity Narration value from the table description in the application database schema.

Auditing Enabled

Progress Dynamics provides for a generic auditing capability to log all changes to any record to a table in the Repository. The Auditing Enabled toggle box setting determines whether auditing is enabled for the selected table. You should only enable auditing where required so as not to adversely affect the performance of the application.

Deploy Data and Version Data

If you select the Deploy Data toggle box, you should deploy the data in the selected table with the application. What data is deployed and how it relates to other data is specified when you set up deployment data sets. In the Progress Dynamics Repository itself, a considerable amount of data is required as part of a deployment package, because much of the data represents the application components themselves, including dynamic object definitions. In ordinary application databases, normally you need to deploy relatively little data with the application. Some examples are: code values needed by the application, messages not stored in the Repository, and other information that needs to be a basic part of the application setup.

If you also do not check the Version Data field, all the data in the selected table is deployed, regardless of what changed since the last deployment dataset was generated.

Select the Version Data toggle box in addition to the Deploy Data toggle box if the data in the table should be versioned, to allow only modified data to be deployed. However, the writing of the versioning records to the Record Version table in the Repository depends on a replication trigger generated by the Progress Dynamics macro code when the ERwin model is forward-engineered. Therefore, unless you set the equivalent user-defined property VersionData in ERwin, the trigger is not needed and, if you set this flag in the entity maintenance, it will not, in and of itself, do anything. For details on using ERwin, see the deployment white paper at <http://psdn.progress.com/library/dynamicswp.htm>.

4.3.5 Maintaining Entity Display fields

The Entity Maintenance window has a browser that lists all the fields from the selected table. If you selected the toggle box to Update Display Field List in the Entity Import utility, then all fields in the table except for Object ID Fields have their Include flag set to **YES**. This setting means that the framework, by default, includes them in the field list for automatically generated Browsers and Viewers, as described in [Chapter 5, “Using the Object Generator.”](#) If you did not select the Update Display Field List toggle box, then all fields initially have their Include flag set to **NO**. In either case, you can adjust the field list for the selected table by double-clicking on the Include cell for any field to reverse its setting.

Also, notice the **Include in default list view** radio set. This option lets you distinguish between including the field in viewers and list views like browsers. If you leave this set to Use Default, then the Include in default view setting determines whether this field will appear in browsers and other list views by default. Otherwise, choose Yes or No to determine if it should be included in browsers and other list views.

You can also adjust the Display Field Order, the Field Label, the Field Column Label, or the Field Format by selecting the corresponding cell and editing it. Choose **Save** to save these changes along with any other data you entered for the table.

The default Display Field Order is based on the order defined in the database schema. The Label, Column Label, and Format were all initialized from the schema information.

Once you have adjusted the Entity information for the tables you have imported, you are ready to generate SDOs and other objects to get you started developing the parts of your application that will use them. [Chapter 5, “Using the Object Generator”](#), explains how to do this.

NOTE: When you select a data field in Entity Maintenance, you can access more properties of the field by displaying the Dynamic Property sheet.

Using the Object Generator

This chapter describes the Object Generator tool, which creates a dynamic SmartDataObject™ for each table or combination of tables you specify in your database. These objects handle database queries to retrieve data from the database, send it to the client, and make updates to the data back on the server, applying any validation logic you have defined. In addition, the Object Generator can create a default dynamic Browser and/or a dynamic Viewer for each SDO, so that you can quickly begin to assemble useful table maintenance windows with almost no development work.

This chapter includes the following sections:

- [Introduction – field, field container, and window objects](#)
- [Field container object basics](#)
- [Using the Object Generator](#)

NOTE: The Object Generator requires AppBuilder to be open while it is open.

5.1 Introduction – field, field container, and window objects

A Progress Dynamics™ Framework application is constructed from many individual components, or objects. Some objects are defined at the individual field level. Other application objects are defined at the level of containers of these individual fields. These objects are generally referred to as SmartObjects because the objects consist of not only a definition of fields and their properties, but also a body of standard supporting code that is automatically made a part of each object of a given type. This supporting code provides each object type with a substantial amount of useful behavior, as well as a mechanism for communicating with other objects, using the Progress 4GL's support for named events. The following objects act as containers for individual fields:

- **Dynamic SmartDataObject (SDO)** — Encapsulates a database query for one or more tables and defines a temp-table of the fields or a subset of the fields in the tables.
- **Dynamic SmartDataBrowser (Browser)** — Creates a Progress frame containing a browse control to allow a user to display, scroll through, and select the records from an SDOs query.
- **Dynamic SmartDataViewer (Viewer)** — A Progress frame containing fill-ins and other representations of the fields in an individual record from a query, used to display record details and allow record updates.

Still other application objects are containers for these field-grouping objects. Generally these are Progress SmartWindows™ (Windows), and possibly tab folders made up of multiple Pages within those windows. A Window can contain one or more SDOs, Browsers, and Viewers that interact to retrieve data from the server, send it to a client session, display it, allow it to be updated, and return those updates to the server for validation and writing to the database. Other supporting SmartObjects used in Windows include the Progress SmartToolbar™, which visually represents a toolbar of action buttons and/or a menu with a hierarchy of items; the Progress SmartFolder™, which represents the pages of a multi-page window and supports the selection of a page at a time to work with; and other specialized objects. There are also nonvisual container objects, such as the static or dynamic SmartBusinessObject (SBO), which groups multiple SDOs to define complex transactions and business logic to support them.

This chapter describes how to:

- Use the Object Generator tool to build a whole suite of objects at the Field Container level for your application database.
- Edit these objects in the AppBuilder and create additional objects with more specific properties.

5.1.1 Generating field objects

The “Entity Import” section of [Chapter 4, “Preparing to Build Application Objects,”](#) discusses the fact that there are objects in the Progress Dynamics Repository to describe the fields in your application database tables, so that you can use them to build other objects containing those fields. The Repository objects that represent database fields are called *Data Fields*. Each Data Field describes a field from a database table. Associated Attribute Value records describe the field’s attributes or properties, such as its data type, format, and so forth. If you select the option to Generate Data Fields in the Entity Import utility, as recommended in that chapter, then the framework creates all the necessary Repository records to represent your application database fields at that time. As described later in this chapter, you can also generate these Data Fields using the Object Generator.

5.1.2 Generating field container objects

The focus of this chapter is on the Object Generator tool. This utility creates a starting set of application components at the level of Field Container objects (SDOs, Browsers, and Viewers). You can then use these objects to build many different types of window-level containers for data display and maintenance.

5.1.3 Generating window objects

[Chapter 8, “Using the Progress Dynamics Container Builder,”](#) shows you how to use the Container Builder tool to create templates for entire windows and pages in a tab folder. It also describes how to use those templates to assemble application windows containing Field Container objects, Toolbars, Folders, and other objects.

5.2 Field container object basics

This chapter discusses how Field Container objects—dynamic SDOs, Browsers, and Viewers—are generated. This section describes some of the essential characteristics of these objects so that you understand better what Progress Dynamics generates for you.

5.2.1 SmartDataObject basics

The Progress Dynamics framework is, to a large degree, based on the use of SmartObjects, and of the Application Development Model 2 (ADM2), to define components, their properties, and their behavior. This guide does not attempt to replace other product documentation on SmartObjects and the ADM2, but it attempts to provide enough background so that if you are not already familiar with the ADM2, you can still get a serious start developing Progress Dynamics applications.

The dynamic SmartDataObject (SDO) is in many ways the heart of a Progress Dynamics application. It is a Progress 4GL procedure that defines a database query for a table, or for a join between multiple tables, and a list of fields from those tables to present to the user. It defines a temp-table (named *RowObject*) with that field list. It is designed to operate transparently in a distributed environment, as are all the other components of Progress Dynamics. To do this, each SDO can run in an AppServer session on a server machine where the Progress Dynamics Repository database and/or the application databases are located. The SDO loads data as requested into its temp-table, and sends that temp-table over to a client session that in principle runs without any direct database connection of its own. Designing for client sessions without a database connection prepares you for deploying your application with the Progress WebClient. This means that user sites, which can be anywhere in the world, do not need a preinstalled Progress run-time session connected to a central server or servers where the databases and the supporting application business logic are installed.

Each client session runs what is called a *proxy* version of the SDO. This proxy version has the same temp-table definition. It can receive data from the server, make it available to other objects (such as Browsers and Viewers) in the client session, collect updates, and pass them back to the server. Progress Dynamics handles the connection between client and server objects internally. If you have defined the logical name of the application Service for your SDOs, and properly configured it in the framework (as described in later chapters), the framework will properly initialize your SDOs and other objects. The objects will communicate with each other so that data is made available on the client efficiently and automatically.

NOTE: Progress Dynamics supports both static and dynamic SDOs, SBOs, Browsers, and Viewers. However, most of the material in this manual assumes you are working with dynamic objects since they are the natural domain of the framework. For clarity, when speaking of only dynamic objects, the dynamics class name may be used as a shorthand. For example, the class name for dynamic SDOs is DynSDO.

Figure 5–1 shows the basic interaction between the server DynSDO and the client proxy DynSDO.

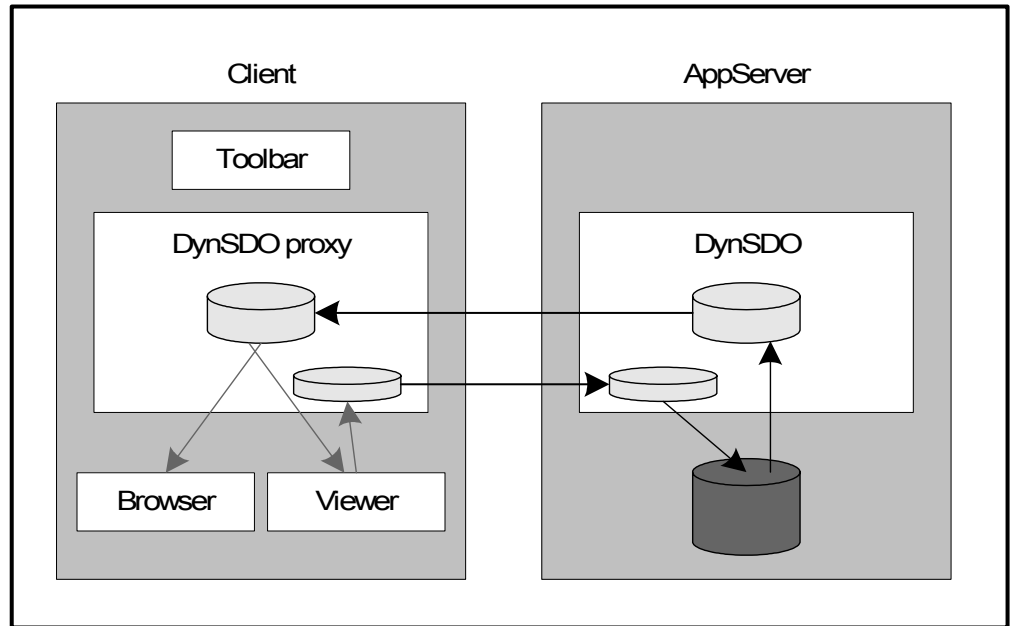


Figure 5–1: Interaction between the server and client proxy DynSDO

Data is loaded from the database into the RowObject temp-table on the server and sent to the equivalent temp-table in the client. Data from the temp-table is displayed in Browsers and Viewers. You can update records in a Viewer (or sometimes in a Browser). Any updates go into a similar temp-table called RowObjUpd containing only updated and added records; these are sent back to the server. Validation logic to assure that all changes are correct is normally executed on the server so that the application database will be available for lookups and updates of related records. Validated changes are written back to the database in the server session. Again, all required server-side objects are started automatically as needed. The client session acts in effect as if all the data were available locally. In fact the same application could run entirely on the client, with a local database connection or a direct client/server database connection, simply by configuring it that way, and everything would work exactly the same.

SDO data logic procedure

The SDO gets data back and forth from client to server properly without any coding at all on the developer's part. But updates to that data almost always require some degree of validation, and perhaps the execution of other business logic to update related tables or to do whatever else is required. This business logic is the essence of your application.

The SDO is designed to make it as easy as possible for you to define at least the first level of business logic, that which applies to updates to a specific table, within the SDO, so that table validation will be executed reliably whenever changes occur to the data in that table.

Later chapters discuss the details of defining business logic in your application. So that you can use the Object Generator and understand what it creates for you, this section covers some of the basics.

The Object Generator can, in a completely automated way, create a dynamic SDO for a set of fields in a table that handles all the basic operations for you. A major accomplishment for SmartDataObjects in Progress Dynamics is that they have been made fully dynamic objects, like most other framework components. This means that the individual source procedures have simply disappeared, and have been replaced by data in the Progress Dynamics Repository that describes everything the application needs to know about the object: its query, its field list, its Application Service logical name, and so forth. A single “dynamic SDO” procedure is able to read the data for any SDO and create the query, temp-table, and other constructs at run time.

The Object Generator writes all business logic relating to the SDO to a separate source procedure. This *Data Logic Procedure* becomes a super procedure of the SDO at run time, so all the internal procedures it contains effectively become part of the SDO when it executes. The Object Generator creates these Data Logic Procedures as part of the overall SDO generation process and even fills in some of the validation logic that can be derived from looking at the database schema.

Figure 5–2 illustrates the whole “family” of objects that make up an SDO.

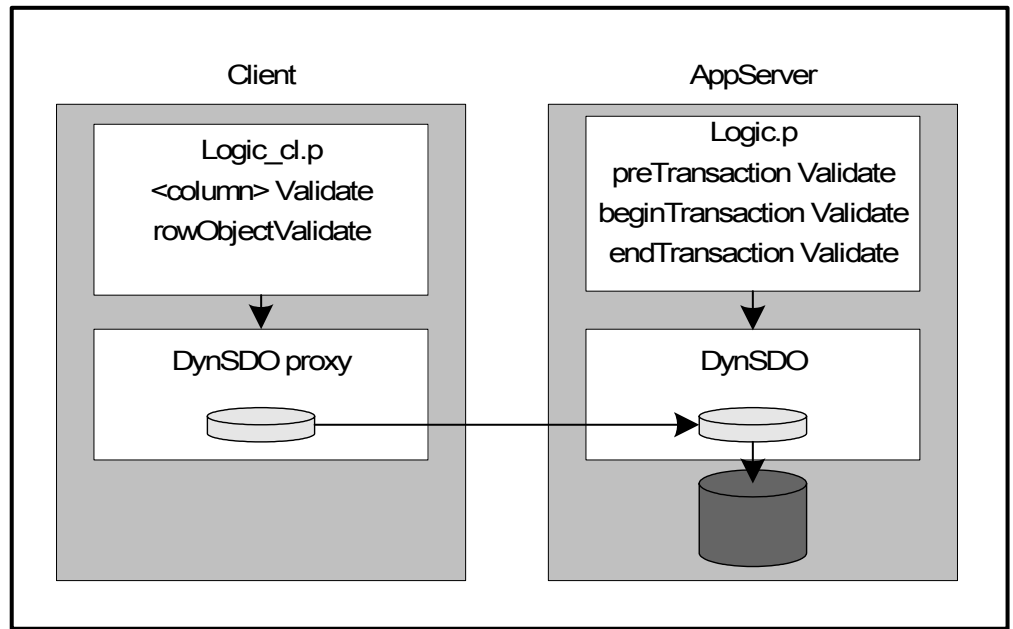


Figure 5–2: Business logic procedure

There are four separate compilable procedures that make up an SDO:

1. The full SDO procedure, which for dynamic SDOs, is a completely Repository-based object.
2. The client-side “proxy” version of the SDO procedure, which has server-side logic compiled out.

The SDO and Logic Procedure proxy files are simple “wrapper” procedures that set a preprocessor value signaling that they execute on the client, and then include the full SDO procedure or Logic Procedure, so that the executable code exists in only one place. Again, for dynamic SDOs, the file does not exist on disk. It is a completely Repository-based object.

3. The server-side Logic Procedure for the SDO, where validation logic requiring database access is written.
4. A client-side proxy for the Logic Procedure, where the developer writes validation logic that can be executed without reference to the database.

There is one more 4GL source file not shown, the include file containing the definition of the RowObject temp-table shared by all the other procedures. The Object Generator creates all of these for you.

The illustration in [Figure 5–2](#) shows some examples of specially named internal procedures in the Data Logic Procedure, such as `rowObjectValidate` and `preTransactionValidate`, where particular types of validation logic go, depending on when it should execute, whether it can execute on the client or whether it must execute only on the server, and what kind of operation (Create, Write, or Delete) it applies to. The details of how to use these hooks are discussed in [Chapter 10, “Building Basic Business Logic in a Progress Dynamics Application.”](#) This chapter discusses some examples of logic created for you by the Object Generator.

5.2.2 SmartDataBrowser basics

The SmartDataBrowser, or more simply Browser, is a Progress frame with a browse control in it. It includes a great deal of supporting behavior and a number of properties to allow browse control to interact with other objects in the application to:

- Display and scroll through records in a related SDO.
- Enable and disable toolbar buttons as certain actions occur.
- Reset record position to a selected record so that other objects such as Viewers can display and update fields from the same record.

The standard Version 9 SmartObject set includes both a static Browser procedure and a dynamic Browser. In releases of the ADM and SmartObjects before the introduction of Progress Dynamics, you must always define a dynamic Browser as an instance in a particular Window, because code written into the Window procedure holds the property settings for the Browser, including its column list, size, etc.

You can use both static and dynamic Browsers with Progress Dynamics, but the framework tools are designed to generate dynamic Browsers for you. Because Progress Dynamics can store instance properties in the Repository, you can define Browsers as entirely data-driven objects without associating them with a particular container. This allows you to use your Browsers in many different windows, without needing any source file procedures for them at all. Also, if you need custom code to extend the behavior of the Browser, you can write it into a custom super procedure, so there is still typically no need for static Browser procedures.

One typical way to use a Browser in an application is to define the Launch Container property for the Browser where you identify the associated maintenance window you want Progress Dynamics to run when a user wants to update a selected record. Alternatively, you could design a single window with a Browser together with a Viewer, on the same page or on different pages of a tab folder. The Viewer will always display the currently selected row, allowing the user to update the row there. A dynamic Browser can also be made updateable itself.

As described in the “Progress Dynamics Links” section in [Chapter 8, “Using the Progress Dynamics Container Builder,”](#) Progress Dynamics provides numerous extensions to standard SmartDataBrowser behavior. In addition to the Launch Container property used to associate the Browser with a maintenance window, various toolbar functions can also act on the currently selected record, including viewing or editing comments for the record, and viewing the audit trail for a record. Toolbar functions can also act on the entire data set displayed by the Browser, including data export functions to send the data to Excel or to a Print Preview tool, and filtering the data set or repositioning to a particular record.

The Progress Dynamics Object Generator described in this chapter creates a dynamic Browser for you, for each table you select.

5.2.3 SmartDataViewer basics

A *SmartDataViewer*, or Viewer, is a Progress frame containing fill-ins and other representations of fields in a single SDO RowObject record. You can also extend it to have buttons, rectangles, and other visual objects. Generally a Viewer displays the currently selected record in an SDO query (which can also be the selected record in a Browser linked to that SDO). Its fields are enabled when data entry is permitted. When a user types changes into an existing record, the **Save**, **Reset**, and **Cancel** buttons of the associated toolbar become enabled. On **Save**, all changes are returned by the SDO to the server for validation. If the user clicks on the **Add** button in a toolbar, then initial values are displayed in the Viewer, and the user can use it to enter other data for the new record. If the user clicks on the **Copy** button, then the Viewer’s initial values match those of the previously selected record. On **Delete**, the currently selected record is deleted.

The Object Generator creates a dynamic Viewer for each table you select. This Viewer has the same field list as the default dynamic Browser that can be generated. This list can include either all the fields in the SDO (minus **Object ID** fields), or those fields specified in the Entity Maintenance tool.

NOTE: See [Chapter 4, “Preparing to Build Application Objects,”](#) for more information.

This Viewer has one or two columns of fields, in the tab order specified either by the SDO field list or the **Display Field** list. The Object Generator calculates the layout of the fields in the Viewer. You can use the SmartDataField Maintenance tool, described elsewhere, to set the Viewer's other properties, including the exact position of fields, the use of dynamic Lookup and Combo objects on fields, and the use of additional visual objects on the Viewer.

A dynamic Viewer can also have a custom super procedure, when you need to write 4GL code to specialize the Viewer's behavior. The code in the custom super procedure can respond to both programmatic events in the object (such as initialization) and User Interface events.

5.3 Using the Object Generator

CAUTION: If you use the Object Generator to regenerate SDOs and you have coded custom logic into the data logic procedures, you must ensure that those procedures are backed up before using the Object Generator. The procedures will be overwritten by this process.

The Object Generator tool creates SDOs, Browsers, and Viewers for all the tables you select. From the AppBuilder window, select **Build→Object Generator**. The **Object Generator** tool appears, as shown in [Figure 5–3](#).

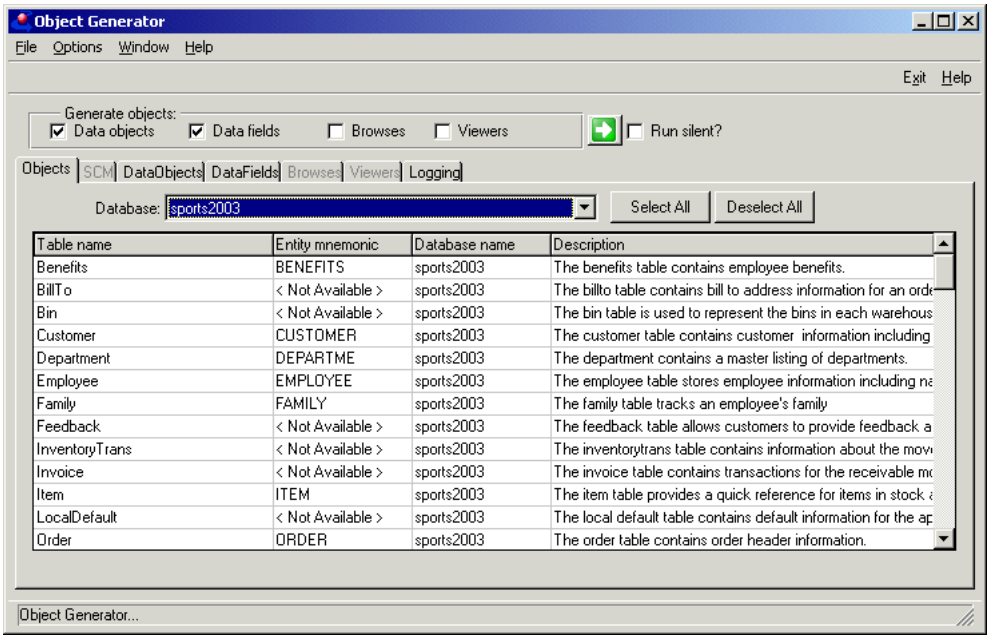


Figure 5–3: Object Generator with multiple tables selected

A series of check boxes at the top of the window allows you to choose what types of objects to generate. In the same row is the green-arrow button which you press to initiate the object generation. Alongside it is the **Run Silent** check box which gives you the option of suppressing message display while generating objects.

The tab folder has several pages on it, each with a specific purpose.

Pages

- **Objects** — This page displays a list of tables belonging to the combo-selected database. This is for selecting the tables for which various objects will be generated. Multiple tables can be selected by using the CTRL key while clicking with the command button on the mouse. If the Data objects check box is not selected, the combo changes to display **Product Module** and the **DataObjects** belonging to the selected product module are displayed, so existing SDOs can be used to generate Browsers and Viewers. The green-arrow button is also disabled until an object type is selected for generation.
- **SCM** — This page is only enabled when working through an SCM session and is for defining related options such as product module. Currently, the only SCM tool supported is Roundtable.
- **Data Objects** — This page is for defining parameters related to generating SDOs. It is only enabled when the Data Objects check box is selected. Datalogic Procedure parameters are also defined here.
- **Data Fields** — This page is for defining parameters related to generating Datafields. It is only enabled when the Datafields check box is selected.
- **Browsers** — This page is for defining parameters related to generating Browsers. It is only enabled when the Browsers check box is selected.
- **Viewers** — This page is for defining parameters related to generating Viewers. It is only enabled when the Viewers check box is selected.
- **Logging** — This page displays the progress of object generation via a two-paned display; the left pane shows a treeview of the object generation steps while the right pane shows the details of any selected item on the treeview. It automatically gets the focus when the object generation is initiated and is dynamically populated, so that object creation progress can be tracked. It is cleared once you press the green-arrow button again.

5.3.1 Object preferences

Choose **Options→Preferences** to bring up the Object Generator Preferences dialog box.

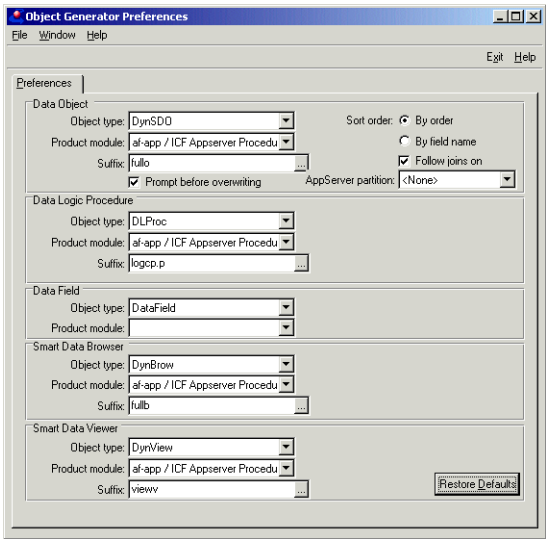


Table 5–1 describes the preferences.

Table 5–1: Object Generator preferences (1 of 2)

Option	Description
Data Object	Specify the default entries Dynamics should use on the Data Objects page of the Object Generator.
Data Logic Procedure	Specify the default entries Dynamics should use on the Data Logic Procedure section of the Data Objects page of the Object Generator.
Data Field	Specify the default entries Dynamics should use on the Data Fields page of the Object Generator.
SmartDataBrowser	Specify the default entries Dynamics should use on the Browsers page of the Object Generator.
SmartDataViewer	Specify the default entries Dynamics should use on the Viewers page of the Object Generator.

Table 5–1: Object Generator preferences

(2 of 2)

Option	Description
Sort Order	Specify the default sort order to use for records from your data sources. Sort by tab order or alphabetically by field name.
Follow Joins	By default, Dynamics does not enable the follow joins feature. Here you can choose to change the default for all AppServer partitions or for just a particular partition. See “Follow Joins,” for more information on the feature.

5.3.2 Objects page

The **Entity Mnemonic** column displays the dump name - an 8-letter acronym assigned by Progress at the time the table is created. If the table has not been imported using the **System→Entity Import** tool from the Progress Dynamics Administration console, the field will display **<Not Available>**, indicating that the entity is not valid for generating objects based on it.

5.3.3 SCM page

The **Source Code Management** page has settings specific to using the framework together with the Roundtable SCM system. Generally you should not modify these settings; they are discussed more in conjunction with other SCM information in other documentation.

5.3.4 Data Objects page

Settings for SDOs are defined on the Data Objects page, shown in [Figure 5–4](#).

Root folder: C:/d/work ☒ Create missing folders relative to the root directory ?

DataObject

Object type: DynSDO Field sequence: ☒ By order ☐ By field name

Product module: EMP (Employees) ☒ Suppress all validation

Relative path: emp ☒ Follow joins

Name suffix: fullo Follow depth: 0

Appserver partition: <None>

DataLogic Procedure

Object type: DLProc

Product module: EMP (Employees)

Relative path: emp

Name suffix: logcp.p

Template: ty/obj/tytemlogic.p

Validation based on:

☐ Mandatory fields

☐ Index membership

Figure 5–4: The Data Objects page of the Object Generator

Above the **DataObject** and **DataLogic Procedure** frames is a field to select the root folder, as described in [Table 5–2](#).

Table 5–2: Data Objects page general fields

Option	Description
Root Folder	This prefix forms the first part of the physical location on disk where files will be stored. The Relative Path makes up the second part.
Create Folder if missing	This check box is used to create physical folder on disk if the one specified does not exist.

The settings are described in [Table 5–3](#).

Table 5–3: Settings for data objects

Option	Description
Object Type	DynSDO is the only type listed.
Product Module	The product module where the data objects will be generated is selected from this combo box.
Relative Path	This suffix forms the second part of the physical location on disk where .i files will be stored. The Root Folder makes up the first part.
Name Suffix	The ending that is added to the Entity Mnemonic (dumpname) for each table to form a complete name for the SDO procedure. You can change this to something other than full0 if you like.
Field Sequence	See “Field Sequence.”
Suppress All Validation	See “Suppress All Validation.”
Follow Joins	See “Follow Joins.”

Create new SDOs or use existing SDOs

As discussed earlier, to use existing SDOs you would have to uncheck the **Data Objects** check box. The reason for this choice is that you might want to make some changes to your SDOs after generating them, but before generating Browsers and Viewers based on them. These are some of the ways in which you can edit your SDOs:

- **Remove some fields from some SDOs** — In this case, you select all fields in the table for the generated SDOs. The **Display Fields** list that you might have edited in Entity Maintenance tool affects only the fields added to dynamic Browsers and Viewers, not SDOs. In some cases, there might be fields that are never used at all on the client, and that are updated only by server-side business logic that is executed when a record is created or updated. Since the SDO's primary function is to send data to the client and to receive and validate updates that come back, you do not need to include these fields in the field list. By removing them, you can improve performance by reducing the amount of data sent to the client. Sometimes, in a case where the number of fields in a table is quite large (which can cause problems in SDO generation), you will find that many of these fields are used only in back-end processing.

Otherwise, you should leave all fields in the “full” SDO that is generated here. You can create alternative SDOs with a subset of fields later to use, for example, in Browsers that just show a few fields for record selection purposes. However, you should always have one “full” SDO through which updates can pass.

- **Make some fields non-updateable in an SDO** — If you do this, it means that the field should never be modified, but only displayed, on the client. If you mark fields in an SDO as non-updateable, then a Viewer generated from the SDO will have those fields disabled by default.
- **Join in other tables, so that useful descriptive fields from them can be displayed** — For every foreign key relationship in a table (for example, `Order.CustNum`), the key value used as the join field might not be meaningful to a user who is, say, updating or adding Orders to a Customer. For this reason, you might want to join an Order SDO to the Customer table to display the Customer Name when entering Orders. The Object Generator does these joins automatically for tables whose keys and field names conform to the Progress Dynamics naming standards. For existing databases that do not observe this convention, you need to do some of this yourself. Note that normally you should make fields from additional tables non-updateable, so that your SDO is only displaying that descriptive data, and never trying to update both ends of a join through a single object.

So as a first pass through your tables, you should generate Data Objects so that you can modify SDOs in the AppBuilder as needed.

Then as a second pass through the objects, you can create Browsers and Viewers based on the modified SDOs. When you do this, the Table browser changes to display SDOs instead, as shown in [Figure 5-5](#).

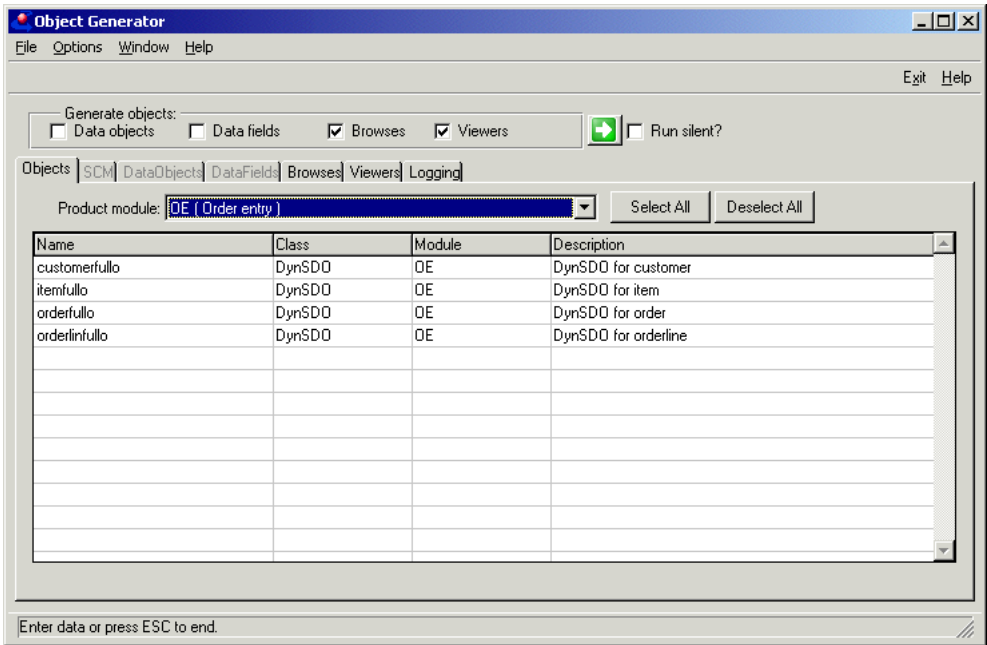


Figure 5-5: Object Generator showing existing SDOs

Now you can select one or more SDOs, check the **Browsers** and/or **Viewers** toggle boxes, and have those objects created.

Suppress All Validation

The **Suppress All Validation** toggle box is checked by default, and you should almost always leave it that way. If you uncheck it, then any field validation expressions defined in the schema will be compiled into the SDOs temp-table definition, and in turn into visual objects such as static Viewers built from the SDO. If this happens, then validation logic involving CAN-FIND expressions or other code requiring a database lookup will fail at run time, because there is no local database connection. In fact, a static procedure requiring a database that is not connected will not even load without error in the client session. In addition, schema field validation code will not be available to any dynamic object, so it will not be executed reliably in any case.

Other kinds of schema field validation are best left out as well. The default behavior that is compiled into a frame field with validation on it is to block an attempted LEAVE event on the field until its value is correct. A user could trigger a LEAVE event by pressing a button, in addition to tabbing out of the field. This behavior would be inappropriate. In general, it is not good GUI design to force validation of a value in a field before entering something in another field. You can write schema field validation into the client-side proxy for the SDO so that it is executed when the user tries to have all updates accepted, but before the record goes back to the server. If certain fields need specific validation, you can write code for this in the custom super procedure for the object, if it is dynamic, or into the object itself if it is procedure-based.

Field Sequence

The **Field Sequence** radio set lets you choose whether the order of the fields in the SDO will be according to the Order defined in the schema for the table (the default) or in alphabetical order. You can modify the order in the SDO after it is created. The Display Fields order determines the order of fields in default visual objects.

Follow Joins

The Object Generator can create joins to other related tables automatically, **if** your database schema conforms to the Progress Dynamics conventions described in [Chapter 2, “Database Design Principles in Progress Dynamics”](#) and [Chapter 4, “Preparing to Build Application Objects.”](#) This means you must meet the following conditions:

- Your join fields must be conformant Object ID fields with names derived from the table name.
- You must have an **Entity Description** field with an appropriate name; or you could have identified the **Entity Description** field in Entity Maintenance.

If these conditions are met, and you set the **Follow Joins** toggle box checked on (default is off), then the Object Generator adds joins to other related tables to the SDO query, and adds the **Entity Description** field for each other table to the field list. You can then add these fields to a Browser or Viewer for the SDO.

If your database does not allow the framework to determine the proper joins and related fields, you can add these other tables and fields yourself after the SDOs are generated, as described earlier.

DataLogic Procedure

The settings for the contents of this frame are described in [Table 5–4](#).

Table 5–4: Settings for DataLogic procedure

Option	Description
Object Type	The object type is selected from this combo, typically DLProc.
Product Module	The product module where the DataLogic Procedure will be generated in is selected from this combo box.
Relative Path	This 'suffix' forms the second part of the physical location on disk where .p files will be stored. The <i>Root Folder</i> makes up the first part.
Name Suffix	The ending that is added to the Entity Mnemonic for each table to form a complete name for the Logic procedure. You can change this to something other than logcp if you like, but the .p extension has to remain the same.
Template	The procedure file that is used as the basis for generating the Logic procedure associated with the SDO. Again, you would not normally change this name; if you want to customize it, the same caveats apply as for the SDO template name.

SDO validation logic generation

The Object Generator creates a Logic procedure for each SDO and puts some of the initial validation logic in place, based on what it can derive from the database schema. This can be a useful starting point for your SDO logic, which you can edit and extend as necessary. [Chapter 11, “Building Advanced Business Logic in a Progress Dynamics Application,”](#) discusses writing code of this kind in more detail.

As an example, the Logic procedure for the Customer SDO contains these three internal procedures that come from the Object Generator:

- RowObjectValidate
- createPreTransValidate
- writePreTransValidate

The Validation based on setting controls what kind of logic gets generated:

- **Mandatory fields** — If checked, mandatory fields are included in the RowObjectValidate validation procedure.
- **Index membership** — If checked, unique fields are included in the RowObjectValidate validation procedure.

RowObjectValidate

RowObjectValidate is the procedure executed on the client side of the application when a record is saved, but before it is sent back to the server. Validation that can be done without database lookups can be placed in this procedure. The isFieldBlank function checks whether a field is either blank or has the unknown value:

```

/*-----
Purpose:  Procedure used to validate RowObject record client-side
Parameters: <none>
-----*/
DEFINE VARIABLE cMessageList AS CHARACTER NO-UNDO.
DEFINE VARIABLE cValueList AS CHARACTER NO-UNDO.

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''}.

IF isFieldBlank(b_Customer.Country) THEN
  ASSIGN cMessageList = cMessageList +
    (IF NUM-ENTRIES(cMessageList,CHR(3)) > 0 THEN CHR(3) ELSE '':U) +
    {aferrortxt.i 'AF' '1' 'Customer' 'Country' ''Country''}.
/* Additional similar validation omitted. */
ERROR-STATUS:ERROR = NO.
RETURN cMessageList.

END PROCEDURE.

```

NOTE: When this procedure is executed, the saved record is available in a buffer with the SDOs principal table name preceded by b_. The procedure builds up a message list (using a standard variable available to all such procedures) of any error messages generated. The validation check itself is handled by the include file, aferrortxt.i, which takes a variety of possible include file parameters, as described in [Chapter 11, “Building Advanced Business Logic in a Progress Dynamics Application.”](#) The framework accumulates any error messages so that it can return all errors to the user in one call.

The second validation procedure Progress Dynamics generates handles new record creates and is intended to execute on the server side of the SDO. This procedure is `createPreTransValidate`, which means that this is record CREATE validation that runs before the server-side update transaction begins. This example verifies that the `CustNum` value entered does not already exist in the database. Even though Progress would catch this error at the database level and report it (because there is a unique index on the `CustNum` field), it is very useful to catch these types of errors in application code before getting down to the default Progress error message generation. This way, you have control over the text of the error message (which will include an intelligible form of the table name, based on the Entity Short Description, etc.), and how it is handled:

```

/*-----
Purpose:  Procedure used to validate records server-side before the
         transaction scope upon create
Parameters: <none>
-----*/
DEFINE VARIABLE cMessageList AS CHARACTER NO-UNDO.
DEFINE VARIABLE cValueList AS CHARACTER NO-UNDO.
IF CAN-FIND(FIRST Customer
            WHERE Customer.CustNum = b_Customer.CustNum) THEN
DO:
  ASSIGN
    cValueList = STRING(b_Customer.CustNum)
    cMessageList = cMessageList +
      (IF NUM-ENTRIES(cMessageList,CHR(3)) > 0 THEN CHR(3) ELSE '':U) +
      {aferrortxt.i 'AF' '8' 'Customer' ' ' "'CustNum, '" cValueList }.
END.
ERROR-STATUS:ERROR = NO.
RETURN cMessageList.

END PROCEDURE.

```

The third example is the procedure `writePreTransValidate`, which executes on the server side for all database write operations of new or existing records. This variation on the previous logic for `CREATE` checks whether this is a write of a new record by using a standard function `isCreate()`, and if not, verifies that the `CustNum` value in the record does not match the `CustNum` of some **other** existing database record:

```

/*-----
Purpose:  Procedure used to validate records server-side before the
          transaction scope upon write
Parameters: <none>
-----*/
DEFINE VARIABLE cMessageList AS CHARACTER NO-UNDO.
DEFINE VARIABLE cValueList AS CHARACTER NO-UNDO.
IF NOT isCreate() AND CAN-FIND(FIRST Customer
    WHERE Customer.CustNum = b_Customer.CustNum
    AND ROWID(Customer) <> TO-ROWID(ENTRY(1,b_Customer.RowIDent)))
THEN DO:
    ASSIGN cValueList = STRING(b_Customer.CustNum)
           cMessageList = cMessageList +
           (IF NUM-ENTRIES(cMessageList,CHR(3)) > 0 THEN CHR(3) ELSE ':U' +
            {aferrortxt.i 'AF' '8' 'Customer' ' ' "'CustNum, '" cValueList }).
END.

ERROR-STATUS:ERROR = NO.
RETURN cMessageList.

END PROCEDURE.

```

5.3.5 The DataFields page

This page gives you the option of generating Data Fields. Data Fields are the Repository objects that represent individual application database fields. As discussed in [Chapter 2, “Database Design Principles in Progress Dynamics,”](#) it is usually best to go ahead and generate Data Fields when you import tables into the Repository. If you have done this, you should not need to check the **Generate Data Fields** toggle box. If you have not already generated Data Fields for your tables, then you should do it here. The DataFields page is shown in [Figure 5–6](#).

Product Module: sm-module (smoke module)

Object Type: DataField

☐ Use Data Object fields

Figure 5–6: DataFields page

Although DataFields are not SmartObjects in the traditional sense of being procedure-based objects with a procedure handle and super procedures to provide behavior and so forth, they are represented in the Repository as if they were SmartObjects. Each DataField has both an Object record and a SmartObject record. Attribute Values are associated with each DataField Smart Object record as for any SmartObject. When you generate DataFields for the fields in your database, these attributes are assigned for each field:

- **ObjectName** (CHR) — The ObjectName of a DataField is its TableName and FieldName, separated by a dot.
- **DatabaseName** (CHR) — The logical database name of the field's table.
- **TableName** (CHR) — The table containing the field.
- **FieldName** (CHR) — The unqualified field name.
- **Label** (CHR) — The field format, derived from the database schema.
- **ColumnLabel** (CHR) — The field's Column-Label from the schema.
- **Format** (CHR) — The field's Format, also from the schema.
- **Help** (CHR) — The field's Help string, if any, from the schema.
- **Mandatory** (LOG) — Whether the field is marked as mandatory or not.
- **FieldOrder** (INT) — This is by default the value of the _Order field for the database field. It could be modified in an SDO instance to reflect a change in the ordering of fields.
- **VisualizationType** (CHR) — This is the View-As property of the field.
- **InitialValue** (CHR) — The Initial attribute of the field in the schema.
- **DataType** (CHR) — The field's data type.

Any SmartObject has attribute values for its “master” object, which are the defaults defined for the object wherever you use it. You can then extend or override these values when you use the object in a particular context. For a SmartObject, this would be, for example, when you use an SDO, Browser, or Viewer in a particular Window. The object takes on additional attributes such as its position, which do not apply to the master object, and you can override other attributes (for example, which fields are enabled) for this instance of the object.

Though not a true SmartObject, the DataField also has master and instance attributes. The master attributes are those derived from the database schema. These master attributes form the basis of any use of the field in a Field Container object. In the case of a static SDO, the field list is defined within the SDO procedure, and the attributes of each field are compiled into that procedure from the include file that defines the SDO's temp-table. However, to support dynamic SDOs, the field records and their attributes as defined in the Repository make it possible to generate the SDO out of the Repository as a dynamic object. For this purpose, instance attributes are generated for each occurrence of a field in an SDO. This includes additional attributes, such as Enabled, to flag whether the field is updateable or not in this SDO. Having these DataFields and their master and instance attributes in the Repository makes it straightforward for you to convert static SDOs to dynamic ones.

In addition, instance attributes are generated for each occurrence of a field in a Viewer. These attributes record its position, size, and other characteristics. Every Progress widget attribute for a field can be assigned for a field in a Viewer. The dynamic Viewer, as a data-driven object, depends on these field definitions and attributes. The static Viewer does not, but as with the SDO, having the fields and attributes in place will simplify converting static Viewers to dynamic ones at a later date, to reduce r-code size and simplify application maintenance.

5.3.6 Browsers page

As discussed earlier, you can generate dynamic visual objects for your SDOs either at the same time you generate the SDOs themselves, or later. Each Browser has a column list based either on the field list of the SDO or on the **Display Fields** list in the Entity Maintenance if this has been defined. The framework assigns the dynamic Browsers to the Module you specified, but of course they have no source code filename or pathname. The Object Name of each generated Browser is the dump name plus a standard suffix, which is fullb (for "full Browser") by default. To change this suffix, edit the field as shown in [Figure 5–7](#).

The screenshot shows a dialog box titled "Browsers page fields" with the following fields and options:

- Product module:** EMP (Employees) (dropdown menu)
- Object type:** DynBrow (dropdown menu)
- Name suffix:** fullb (text input field)
- Max number of fields:** 48 (text input field)
- ☐ Delete contained DataField instances
- ☐ Use data object fields
- ☐ Use data object field order

Figure 5–7: Browsers page fields

Table 5–5 describes the fields in this display.

Table 5–5: Browsers page fields

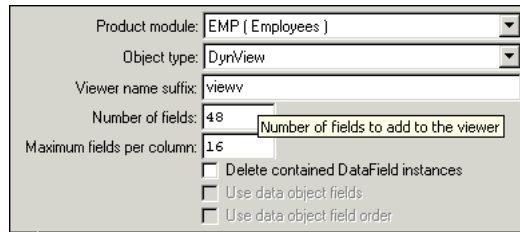
Option	Description
Object Type	The object type is selected from this combo.
Product Module	The product module where the Data Fields will be generated.
Name Suffix	Text added to the end of the Entity Mnemonic for each table to form a complete name for the Browser. Typically, the suffix would be fullb.
Number of Fields	The maximum number of fields allowed for the Browser.
Delete contained DataField Instances	If checked, this deletes existing DataField Instances for the Browser.
Use Data Object Fields	If checked, the Object Generator will base the Browser fields on those specified in the SDO. If left unchecked, it will default to all the fields in the table. This is only enabled if generated from existing SDOs. This option is only enabled if generated from existing SDOs.
Use Data Object Field order	Only enabled if generated from existing SDOs. If checked, the Object Generator bases the field tab order on the order specified in the SDO. If left unchecked, it will base the field order on the schema. This is only enabled if generated from existing SDOs.

5.3.7 Viewers page

If you check the **Viewers** toggle box, the framework creates a dynamic SmartDataViewer for each selected table, with the same field list as the dynamic Browser. Each dynamic Viewer is laid out in one or two columns, with field positions based on the field order in the SDO or in the list of fields specified in the Entity Maintenance tool, whichever applies. Visualization types are based on the VIEW-AS property of each field in the schema. You can also change a few defaults for the generated Viewers, including:

- The maximum number of fields to display in the Viewer
- The suffix to be appended to the table dump name to form the Viewer object name
- The maximum number of fields to display in each column, which can determine whether the Viewer is displayed in one column or two

Figure 5–8 shows this page.



Product module: EMP (Employees)

Object type: DynView

Viewer name suffix: viewv

Number of fields: 48 Number of fields to add to the viewer: 16

Maximum fields per column: 16

☐ Delete contained DataField instances

☐ Use data object fields

☐ Use data object field order

Figure 5–8: Viewers page fields

Table 5–6 describes the fields in this display.

Table 5–6: Viewers page fields

Option	Description
Object type	The object type is selected from this combo box.
Product module	The product module where the Data Fields will be generated in is selected from this combo box.
Name suffix	The ending that is added to the Entity Mnemonic for each table to form a complete name for the Viewer. Typically, the suffix would be viewv.
Number of fields	The maximum number of fields allowed on the viewer.
Max number of fields per column	The maximum number of fields allowed in one column before the next column is rendered.
Delete contained DataField instances	If checked, this will delete existing DataField Instances for the Viewer. The original viewer fields replaced by SDFs will be restored to the viewer.
Use data object fields	If checked, the Object Generator will base the Browser fields on those specified in the SDO. If left unchecked, it will default to all the fields in the table. This option is only enabled if generated from existing SDOs.
Use data object field order	Only enabled if generated with from existing SDOs. If checked, the Object Generator will base the field tab order on the order specified in the SDO. If left unchecked, it will base the field order on the schema.

5.3.8 Logging page

The double-pane window looks like the one in [Figure 5–9](#).

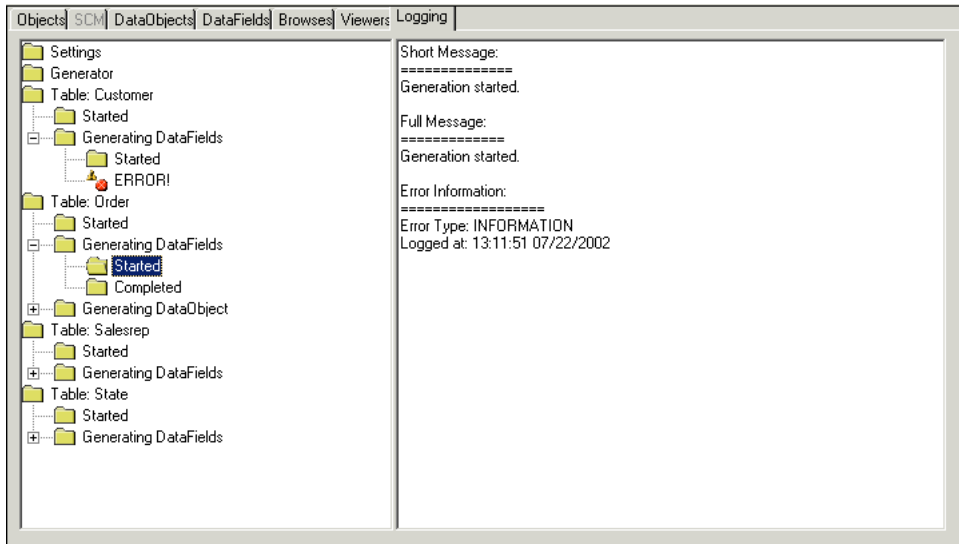
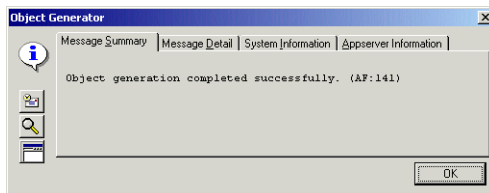


Figure 5–9: Logging page

The pane on the left displays a treeview of the object generation steps while the right pane shows the details of any selected item on the treeview. It automatically gets the focus when the object generation is initiated and is dynamically populated, so progress can be tracked as the objects are created. It is cleared once you press the green-arrow button again.

5.3.9 Generating objects

When you have completed filling in the pages, choose the green arrow button to begin the object generation. Enable the Run Silent check box to minimize the amount of feedback the Object Generator provides. The process could take some time. When the process is complete, the following message appears:



Using the AppBuilder in Progress Dynamics

The Object Generator creates many objects at a time, with common default characteristics. This is a useful way to get started in building your application, but you might need to make changes to some of these objects and create additional ones for different purposes. You can do all of this through the AppBuilder, as described in the following sections:

- [Creating and editing individual objects](#)
- [Building viewers](#)
- [The Dynamic property sheet](#)
- [Migrating static objects to Progress Dynamics](#)

6.1 Creating and editing individual objects

Two reasons for being able to create individual objects are:

- An object may need to be extensively customized, unlike one generated by the Object Generator.
- It is the only way to create a static object.

Some of these are dealt with under the following sections:

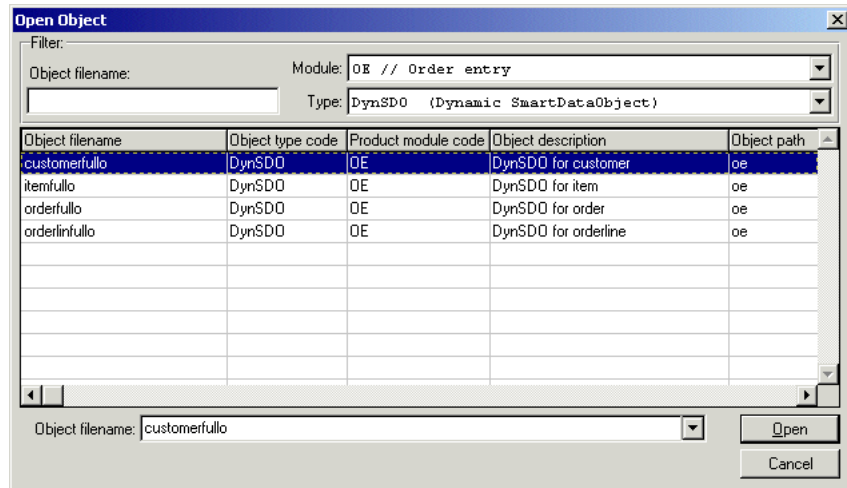
- [Creating and editing SDOs](#)
- [Creating and editing browsers](#)
- [Creating and editing viewers](#)

6.1.1 Creating and editing SDOs

To edit an SDO created with the Object Generator, follow these steps:

- 1 ♦ In the AppBuilder, you can open the SDO either as a file, because it is a procedural object, or as an object, because the SDO was created in the framework and it is registered in the Progress Dynamics Repository:
 - a) To open it as a procedure, select **File→Open File** or choose **Open File** from the AppBuilder toolbar. The standard operating system **File Open** dialog box appears. Select the SDO with its .w filename extension.
 - b) To open the SDO as an object, select **File→Open Object** or choose **Open Object** in the toolbar. The **Open Object** dialog box appears. It shows the object name without the filename extension, because the object name is only the simple filename.

Locate the object based not on the operating system directory it is in, but by its Product Module or Object Type:

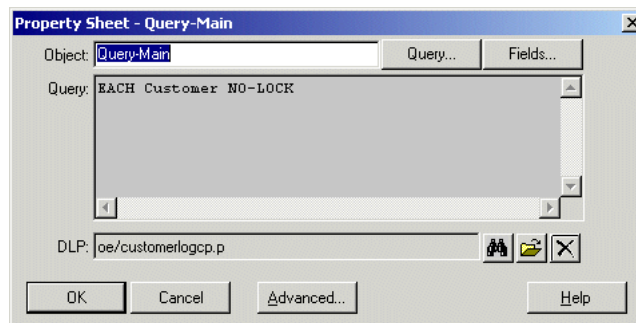


Either way, a design window for the SDO appears:



The SDO, as a nonvisual object, has no real window. This design window lets you get at its properties.

- 2 ♦ To bring up the SDO's properties, as with any AppBuilder design window, you can double-click on the window or choose the **Object Properties** toolbar button:



From the property sheet, you can edit the SDO's query, for example, to add another table to the primary table, or choose **Fields** to modify the field list for the SDO, to remove fields, to add fields from a new table, or to change the updateable setting for one or more fields.

3 ♦ To edit the SDO's query do one of the following:

- From the property sheet, choose **Query**.
- Choose the **Procedure Settings** button in the toolbar or select **Tools→Procedure Settings**.

In the **Procedure Settings** dialog box, you can change the default AppServer partition name for the SDO, as well as other settings.

4 ♦ After opening an object, editing it in any way in its property sheet and other dialog boxes, and returning to the design window, remember to choose **Save** from the design window. Changes are only held in memory until you save. This allows you to go in and out of a property sheet more than once before finishing your changes.

The standard Version 9 ADM product documentation describes all of these options in more detail. They are presented here just so that you know where to look to make the kinds of changes you are likely to need to make to SDOs you have generated.

You might also want to create additional SDOs, either to join tables in different ways or to create objects with different field lists. You should create additional SDOs with care, however. If you want to create SDOs designed specifically for browsing small numbers of columns efficiently so that users can look up a record to edit in another page or window, that is fine. Avoid a situation where different parts of your application are updating records through different SDOs on the same table, because then you can be at risk of not having your business logic execute consistently.

Generally, let your full SDO be the only one through which updates are done; create others as needed for browsing or specialized joins. And keep in mind that having fully dynamic SDOs, the temp-table definitions for the SDOs have become dynamic as well, so that the dynamic SDO can support a variable fieldlist as a property of the object. This removes the need to have multiple SDOs just to modify the fieldlist. The fieldlist is a property you can set for the dynamic SDO.

NOTE: The standard AppBuilder preference (Options →Preferences) Qualify Database Fields with a Database Name must be turned off to create and edit SDOs in Progress Dynamics. Errors occur when trying to save an SDO when this option is enabled.

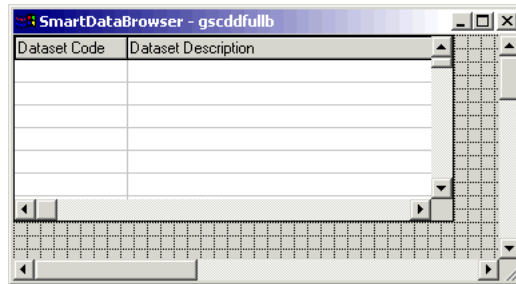
6.1.2 Creating and editing browsers

The Object Generator creates the first dynamic Browser for each SDO. To edit this Browser, or to create additional Browsers, use the Browser Property Sheet. You can bring this up for an existing Browser using the **Open Object** button or menu item in the AppBuilder. Because the dynamic Browser is not a procedural object, you cannot access it via the **Open File** dialog box.

To create a new dynamic Browser, follow these steps:

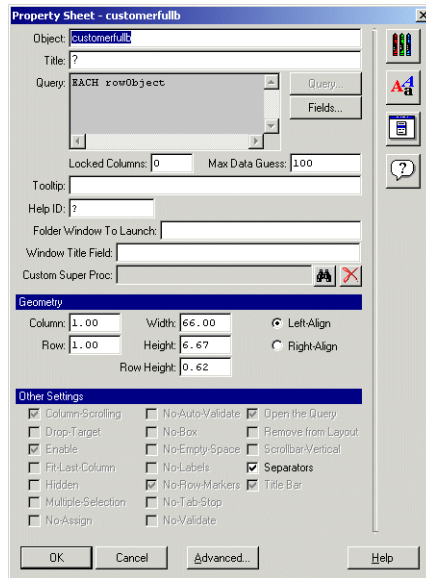
- 1 ♦ Choose the **New** button in the toolbar or select **File→New** from the menu.
- 2 ♦ Select **Dynamic SmartDataBrowser (DynBrow)** from the list of object types.

When you create a new browser or open an existing one, it displays with a nonvisual design window of its own:



The AppBuilder provides a design-time visualization of the dynamic browser, so you can now edit it using its property sheet.

- 3 ♦ Double-click on the design window outside the icon to bring up the property sheet:



If you create a new browser, the Product Module is initialized to the currently selected Module in the AppBuilder.

- 4 ♦ Change the Product Module, if necessary.
- 5 ♦ Enter a unique object name for the browser and a meaningful description.
- 6 ♦ Type in the name of the SDO from which this browser should be derived. (The fill-in field is a Progress Dynamics Lookup, which automatically completes the name for you if you enter a unique substring of an existing SDO name.) Choose the **Lookup** button to select an SDO from the Repository.

When you choose an SDO, the **Available Fields** list is populated with all the names of the fields in the SDO.

If the Browser needs custom 4GL code to extend its behavior, you can write it in a separate procedure and attach it to the browser instance at run time as a custom super procedure.

- 7 ♦ If your Browser needs code of this kind, enter the name of the Custom Super Procedure. The procedure does not need to exist yet, but you will get an error if you put your dynamic Browser into a Window and try to run it before the custom super procedure is created. Be sure to include a relative (not absolute) pathname as part of the procedure name, so that Progress Dynamics can find the procedure at run time relative to your application Propath.

- 8 ♦ You can also configure your Browser to display a separate data maintenance window when:
- The user double-clicks on a row.
 - If the Browser contains a toolbar, the user chooses an update button (as described in [Chapter 8, “Using the Progress Dynamics Container Builder”](#)).

The Launch Container field lets you specify a default name for the dynamic container to launch when a row is selected.

NOTE: At some point you may design an interface with two browsers in a parent-child relationship. When the child browser is open in update mode, Dynamics prevents user changes from occurring in the parent browser. Certain features like the pop-up menu won't be available on the parent browser.

6.1.3 Creating and editing viewers

The Progress Dynamics framework provides a full-featured dynamic viewer. At the same time, the Viewer is typically the object type you customize most frequently in an application, both to give it a specific layout (which might include rectangles, buttons, and other additional visual objects), and to customize its behavior (to provide specific LEAVE triggers or other UI events). The dynamic Viewer as an object supports all of these things: precise layout of fields, visualization types for fields, UI events and trigger code to respond to them, rectangles, buttons, and more. You can adjust the attributes that control these features in the DataField Maintenance tool.

Dynamic Viewers can be created and edited in the AppBuilder.

When you build a dynamic Viewer, you have full control over placement of visual objects, defining events and trigger code to handle them, adding procedures to the file, etc. Create Static Viewers in Progress Dynamics using the same technique you use to create them in standard Version 9 Progress ProVision®: Select **File→New→Static SmartDataViewer** and follow the Wizard's instructions.

6.1.4 Changing an object's product module

If you want to change the product module for an existing object, you can do so in the Repository Object Maintenance (ROM) tool.

CAUTION: Before you change the Product Module, you should understand the following implications of making such a change:

- If you use a Source Code Management (SCM) tool, it might not allow you to change the Product Module of the selected object. (You should check your SCM tool guidelines first.)
- When you change the Product Module of an object, Progress Dynamics moves the object in the repository, but it does not move its associated Application Data Object (ADO) file on the disk. You must manually move the ADO file from the old Product Module directory to the new Product Module directory in the central repository. If you do not, you could experience problems when you deploy your application in a distributed development environment. If you deploy datasets without first moving the ADO file, Progress Dynamics could inadvertently load invalid data into a target repository.
- If you change the Product Module for a static object you must manually move both the static object file, as well as its associated ADO file, from the old Product Module directory to the new Product Module directory. (The repository only stores the association between the static file and the Product Module.)
- If there are any references to the object using its old relative path, you must change those references so that they use the new object path. (You should not have to do this for a logical object.)

Follow these steps to change an object's Product Module:

- 1 ♦ From the AppBuilder main window, select **Build→Repository Maintenance**.
- 2 ♦ Specify **Object type**, **Object name** and the **Product Module** of the dynamic object you want to change, then choose **Apply**.
- 3 ♦ In the TreeView, under the **Objects node**, select the object name. The **Repository Object Control** section displays to the right of the TreeView.
- 4 ♦ In the Details tab, change the Product and/or Product Module selections, then choose the **Save** button.

A message displays, warning you that you are about to change the Product or the Product Module for the selected object.

- 5 ♦ To make the change, choose Yes in the Confirm Product Module Change alert box.
- 6 ♦ If necessary, make the additional changes described in the above Caution statement.

6.2 Building viewers

The Progress Dynamics static viewer is essentially the same object as a standard Version 9 SmartDataViewer. It is based on a slightly different template, the file ry\obj\rysttvieww.w, which has a few Progress Dynamics-specific definitions at the top, and a slightly different set of wizard pages to guide you through building the object. Otherwise, it is the same as the standard SmartDataViewer.

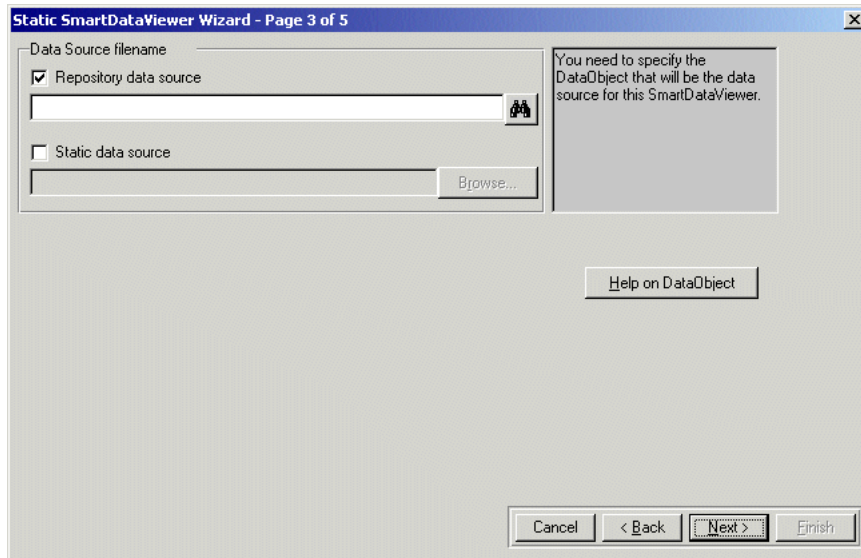
To build a static viewer, follow these steps:

- 1 ♦ From the AppBuilder main window, select **New→Static SmartDataViewer**.
- 2 ♦ Proceed past the Welcome page of the wizard. The second wizard page is not in the standard Viewer wizard. It prompts you for descriptive information about the Viewer:

The screenshot shows the 'Static SmartDataViewer Wizard - Page 2 of 5' dialog box. It is divided into several sections. On the left, there are input fields for 'Object' (containing 'rysttvieww.w'), 'Description' (empty), 'Product module' (containing 'OE // Order entry'), 'Purpose' (empty), 'Parameters' (containing '<none>'), and 'Version notes' (containing 'Created from Template rysttvieww.w'). On the right, there is a 'Source Code Management Info' section with fields for 'Version' (010000), 'Task' (0), 'Date' (09/16/2003), 'Author' (empty), and 'Ref.' (empty). Below these fields are five buttons: 'Blank Definition Section', 'ReRead Definition Section', 'Import from SCM', 'Import Full Version History', and 'Just Copy to SCM'. At the bottom of the dialog are four buttons: 'Cancel', '< Back', 'Next >', and 'Finish'.

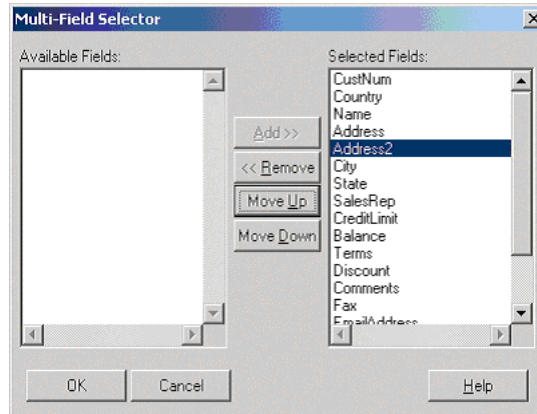
- 3 ♦ Fill in a meaningful **Description** and **Purpose**. This text will be written to the top of the **Definitions Section** of the Viewer when you save the file. To clear the text in the **Definitions Section**, choose **Blank Definitions** section. Then type your **Description** and **Purpose** over again. Choose **Re-Read** to restore the previously saved **Definitions** information.
- 4 ♦ In the **Object** field, replace the template name with the name of your viewer.

- 5 ♦ If you are using Progress Dynamics in conjunction with the Roundtable SCM system, fill in the **Source Code Management Info** section. Otherwise, leave these fields unchanged.
- 6 ♦ Choose **Next** to proceed to Page 3:

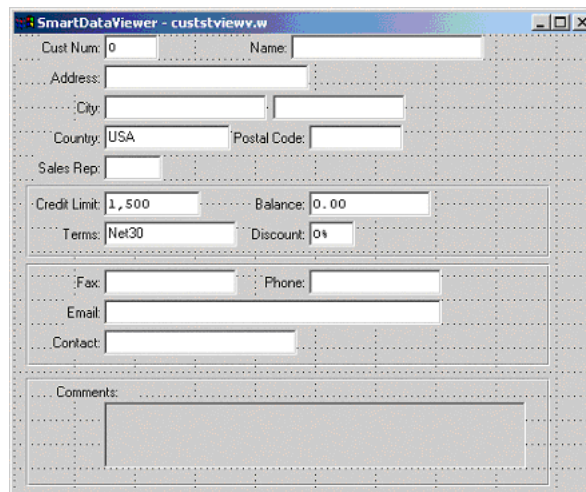


- 7 ♦ Choose whether the data source should be from the Repository or Static, and specify whether the Viewer itself will be Static or Dynamic.
- 8 ♦ For **Data source filename**, enter or browse for the name of the static or dynamics SDO or SBO whose fields will go into the Viewer.
- 9 ♦ Choose **Next**.

- 10 ♦ Choose the **Add Fields** button on the next page to select fields from your SDO or SBO to place onto the Viewer:



- 11 ♦ Choose **OK**.
- 12 ♦ Choose **Next** in the Fields page, and you are done.
- 13 ♦ Press **Finish** in the final Wizard page, and your Viewer appears with a default field layout:



- 14 ♦ Arrange the visual layout of the Viewer any way you like, then save it.

You can learn more about building SmartDataViewers and using the AppBuilder for both visual layout and editing code blocks in the standard Progress ProVision documentation.

NOTE: The PROGRESS 4GL renders radio-set widgets as CHARACTER data type.

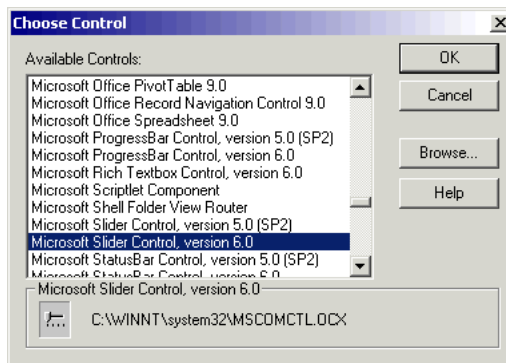
ActiveX controls

ActiveX or OCX controls are not supported on Dynamic Viewers. They can, however, be placed on a Static Viewer as follows:

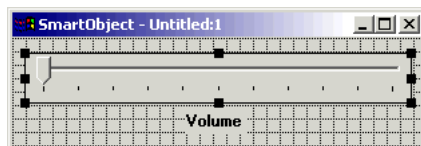
- 1 ♦ Open any existing Static SmartObject or create a new one to place the control on.
- 2 ♦ From the AppBuilder Palette, click on the OCX icon to place an OCX control on the Static SmartObject:



- 3 ♦ The following dialog appears, prompting you to select a control:



- 4 ♦ Select a control and click OK.
- 5 ♦ Place the new ActiveX control on the Static SmartObject as follows:



- 6 ♦ Save changes.

6.3 The Dynamic property sheet

This tool is specifically aimed at simplifying the task of applying attributes or properties to a dynamic object. It has been added to the AppBuilder (specifically for Progress Dynamics), the Toolbar and Menu Designer, and the Container Builder.

It provides an intuitive way of applying Repository based attributes and events to individual or multiple objects simultaneously. It also provides for reverting to the original attribute value where an attribute has been overridden, and where changes have not yet been saved.

An advanced filtering mechanism has also been added to this utility to simplify searches.

It supports customization through the use of result codes and allows you to assign default attribute and event values, and to override these values for alternate customizations.

6.3.1 User interface

The Dynamic Property Sheet functions similarly to an ActiveX property sheet and somewhat to the AppBuilder Properties Window. It consists of a non-modal window with a list of all allowable properties that can be modified for a specified object.

It depends on the tool being used (AppBuilder, Container Builder or Toolbar and Menu Designer) to register the object or objects which will have attributes assigned. The object may be an object which is contained in another object, such as a dynamic fill-in field that is contained in a Dynamic Viewer. Or it can be a master object that is not necessarily contained within another object, such as a dynamic window, or a dynamic SmartToolbar. Only attributes that are applicable to an object are displayed, based on where the object fits into the class hierarchy. Additionally, run-time properties are also displayed.

There will always be only **one instance** of the property sheet per session. This window is structured to so it can be called from multiple areas and refreshed accordingly, as shown in [Figure 6-1](#).

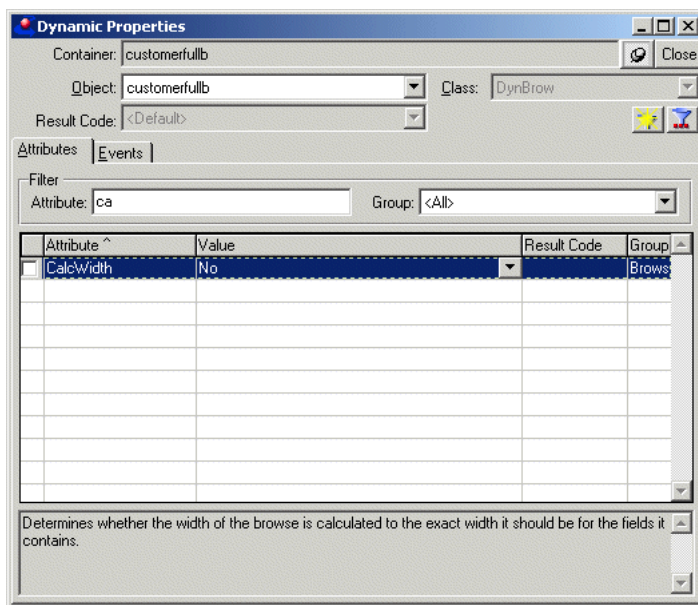


Figure 6-1: Dynamic property sheet

6.3.2 Description fields

The first group of fields at the top of the window describe the object:

- **Container** — Displays the container name.
- **Object** — Displays a drop-down list of objects in the container.
- **Class** — Displays the object type. It is only enabled for objects whose class can change and displays a list of allowable alternatives.
- **Result code** — Displays a list of all possible result codes that are currently defined in the system. Two other entries are available: <DEFAULT> is the default attribute, which is used for displaying the default attributes having no result code; <ALL> displays all result codes. Upon changing the value, the browser is refreshed with the selected result code.

6.3.3 Tab folders

The lower part of the window is occupied by two tab folders: **Attributes** and **Events**, the two applicable components.

Attributes

The **Attributes** tab shows a data grid of all the attributes for the selected object. It acts as an updateable browser, allowing changes to be made to the selected row, where applicable. The columns are as follows:

- The first column shows a check box for the selected row, which is selected as soon as a change is made. By clearing the default restores the value to its default as a change to the property values; this can happen whether another change has been committed or not.
- The **Value** field is either a fill-in or a combo box, depending on the row selected.
- The **Result Code** field indicates the result code that was selected in the **Result** combo box. If the user specifies a specific result code, only those result codes are displayed. If a result code is selected, and the value has not yet been overridden, the value displayed in the value field will be the same as the value for the **Default** attribute.
- The **Group** field is used for grouping similar attributes together. All Attributes have a group assigned to it.
- The bottom pane of the **Attributes** tab folder displays a detailed description of the selected attribute.

Events

The **Events** tab displays all allowable events for the current object. Some of the columns are as follows:

- The first column shows a check box for the selected row, which is selected as soon as a change is made. By clearing the default restores the value to its default as a change to the property values; this can happen whether another change has been committed or not.
- The **Event** field displays all event labels. All available events are based on the object class and are retrieved from the Repository (read-only). The list of fields are initially sorted based on the event label (read-only).
- The **Action** field specifies the name of the procedure that is run when the event is fired. The location of the procedure is based on the target field.

- The **Result Code** field indicates the result code that was selected in the **Result** combo box. If the user specified a specific result code, only those result codes are displayed (read-only).
- The **Type** field may either contain the value of either **RUN** or **PUB** (Publish).
- The **Target** field specifies from where to publish or run the action. [Table 6–1](#) shows the possible options.

Table 6–1: Target field options

Option	Description
SELF	This specifies to publish the action in the target procedure.
CONTAINER	This specifies to publish the action from the container.
ANYWHERE	This specifies to publish the action.
Specific manager	This is a list of specific managers that exists in Progress Dynamics. If specified, the action is run in that manager.

- The **Parameter** field is a character parameter string that is passed to the event procedure.
- The **Disabled** field disables the event if set to YES.





Sorting

Sorting involves clicking on any one of the column headers as the field to be sorted by, in ascending order. For descending order, click another time on the same column header.

Filtering

Table 6–2 describes the filtering options available.

Table 6–2: Filtering options

Icon name	Icon	Description
Hide Filter Fields		On the Attributes tab, the Filter Attribute and Group fields are visible. Clicking hides them. On the Events tab, it hides the Attribute field.
Show Filter Fields		On the Attributes tab, the Filter Attribute and Group fields are not visible. Clicking displays them. On the Events tab, it displays the Attributes field.
Clear Filter		On the Attributes tab, there is a value in the Filter Attribute or Group field or both. Clicking clears these values, thus displaying all available dynamic properties. On the Attributes tab, there is a value in the Filter Attribute field. Clicking clears this value, thus displaying all available dynamic properties.
Clear Filter		On both tabs, there is no Filter value set. All dynamic properties are listed. Clicking produces no effect.

Multiple selections

If more than one object is selected, only those similar properties are displayed. The value will be blank, unless the values are the same for each object. The user can then change common attributes to ALL selected objects at one time.

6.4 Migrating static objects to Progress Dynamics

The purpose of a migration tool is to enable you to convert static objects to dynamic objects. This section briefly describes the two current methods that facilitate this functionality.

6.4.1 Individual object migration

This functionality is discussed earlier in [Chapter 3, “Progress Dynamics Integration with the AppBuilder.”](#)

6.4.2 Batch object migration

This utility generates Dynamic versions of specified Static objects, including SmartBusinessObjects. It works on the same principle as the **File> Save as Dynamic Object** option, except it is done for a selected and filtered group of objects. To launch it, choose the ADM to Dynamic icon from the ProTools palette. The window shown in [Figure 6–2](#) opens.

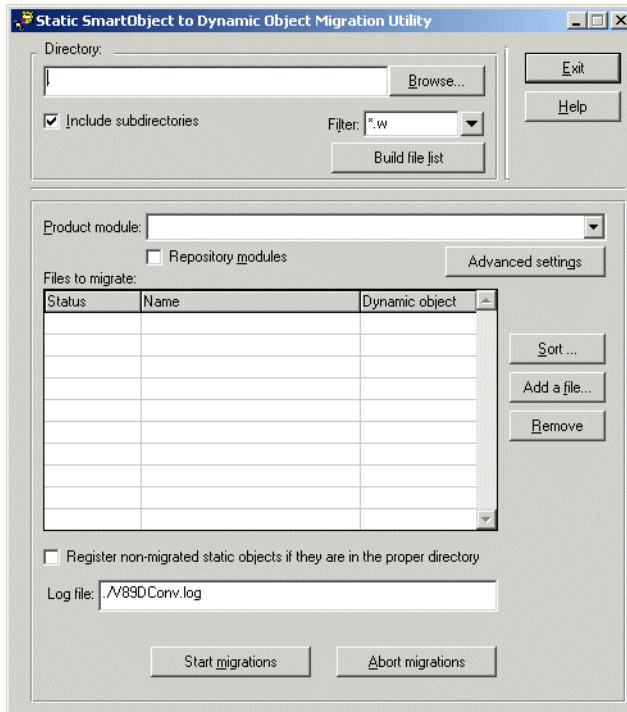


Figure 6–2: Static SmartObject to Dynamic Object Migration Utility window

The filtering mechanism works on two levels: the physical directories and subdirectories on disk and the product modules in the Repository.

Some of the controls on the UI are explained below:

- **Directory** — Field defaults to the root directory, but you can change it.
- **Filter** — Shows the file type, which defaults to .w.
- **Build/Rebuild file list** — Clears and populates the list of files to migrate.
- **Product module** — Is the product module to which the new objects will be assigned.

- **Repository modules** — Lets you specify to include Progress Dynamics framework modules.
- **Register** — Non-migrated static objects if they are in the proper directory: If the utility determines that it cannot convert a static object to its dynamic form, it will register the object as a static object (but only if the static object already exists in the directory for the specified Product Module.) You should specify to register non-migrated objects so that the Progress Dynamics framework will be aware of their existence. You must register all objects that you want to deploy.
- **Log file** — Lets you specify a different name and relative path for the log file, which you can review and print.
- **Start migrations** — Initiates the processing of files shown in the list. Each file is read into the AppBuilder silently (without any visualization.) If it is a SmartDataViewer (Version 8 or Version 9), a SmartDataBrowser (Version 8 or Version 9), or a SmartBusinessObject (Version 9), it is written to the Repository. Any other file type is rejected or not converted, although all files will be registered in the Repository. A log file is kept that indicates any rejections, any errors encountered, any unusually situations and all of the objects successfully converted. If no Product Module is specified, then nothing is converted and an alert box displays explaining the problem.
- **Advanced settings** — Displays the Advanced Migration Settings Window as shown in Figure 6–3.

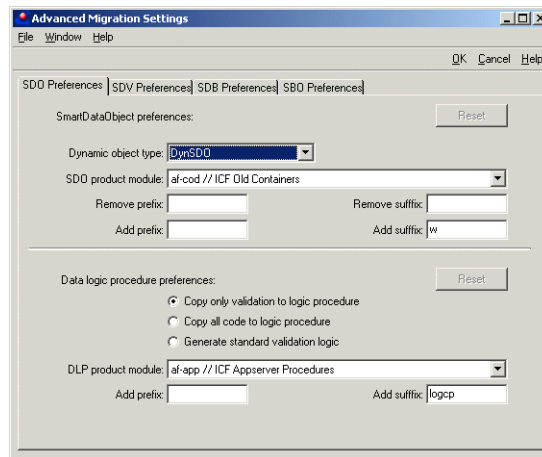


Figure 6–3: Advanced Migration Settings window

This Advanced Migration Settings window lets change the naming conventions and actions of the Static SmartObject to Dynamic Object Migration Utility tool. For example, you can:

- Specify different Product Modules for different object classes.
- Generate Datalogic Procedures containing all custom code sections of the original static SDO.
- Generate Custom Super Procedures containing all custom code sections of the original static SDV or SDB.
- Add or remove suffixes and prefixes to or from the object name.

Building Progress Dynamics Lookups and Combos

A *SmartDataViewer* (Viewer for short) is a Progress Dynamics object visualized as a Progress frame. It contains field-level widgets, like fill-ins and editors, representing fields in static or dynamic SmartDataObjects™ (SDO) or SmartBusinessObjects (SBO). Progress Dynamics supports both static (procedural) and dynamic (data-driven) Viewers.

A Viewer is normally linked to an SDO or SBO, from which it gets its data. It displays the currently selected record in the data object's query, which is typically positioned to by using a Browser to scroll through a set of records and select a record, or by using a Navigation band of buttons in a toolbar to advance to a desired record. You can make the Viewer a TableIO link target for an Update band of toolbar buttons, so that the user can modify, add, copy, and delete records through the Viewer.

Because of the architecture of the Progress Dynamics framework, a Viewer is significantly different from a traditional Progress frame where records are updated. Because of the design of the framework to support distributed applications, with no direct database connection on the client, the field widgets in a Viewer are not directly associated with database values. Instead, they normally display fields in a temp-table record from an SDO, retrieved on the server and sent to the client for display. Field validation expressions, which you would traditionally write into the database schema and compile from there into frames that reference those fields, are normally left out of Viewers so that any database connection dependencies do not prevent the Viewer from running in a Progress session without a database connection. Changes to field values are therefore normally collected when an entire record is saved, sent back to the client-side SDO, and from there to the server for validation and writing to the database.

A Viewer can also contain other visual objects. This includes other field-level widgets mapped to variables or other non-database values, browses with their own queries, decorative objects such as rectangles and images, buttons with their own actions, and so on.

This chapter discusses how to build Viewers in the AppBuilder, and in particular, how to add dynamic Lookups and Combos to them. In the course of adding Combos and Lookups to dynamic Viewers, you also get a substantial introduction to the Repository Maintenance tool, which lets you add, delete, and modify individual records in the Repository to make changes to dynamic objects.

This chapter includes the following sections:

- [Adding dynamic combos and lookups to viewers](#)
- [Combo and lookup performance and caching considerations](#)
- [Old or New Version 2.1B API](#)
- [Subclassing dynamic combos and lookups](#)

7.1 Adding dynamic combos and lookups to viewers

The most common addition to a Viewer is likely to be a choice list for a foreign key field. A *foreign key field* is a field in one table whose value must come from a unique key in another table. Examples of foreign keys in the Sports2000 database are the CustNum field in the Order table, which is a foreign key for the unique CustNum key field in the Customer table, and the SalesRep field in the Customer table, which is a foreign key for the unique SalesRep key field in the SalesRep table.

Progress Dynamics provides two different built-in visualizations for choice lists. Both are based on the SmartDataField SmartObject, which is a type of object designed to provide a specialized representation of a single field in a Viewer.

You can learn more about SmartDataFields as a class, and about other specific objects of this type, in the standard Progress ProVision product documentation. If you are familiar with the standard ADM2 and SmartObjects, you should understand that the dynamic Combo and Lookup objects are very similar to the SmartSelect object, which can be visualized in several ways, including as either a lookup browser or a combo drop-down list. However, the dynamic Combos and Lookups in Progress Dynamics do not need a dedicated SDO for each object, which saves the overhead of that extra object. Also, when there is more than one Combo or Lookup on a Viewer, they are coordinated so that data to populate all the objects in a Viewer is retrieved from the server in a single call. They also have additional features not found in the SmartSelect.

An entire Progress procedure supports the visualization of the field so that you can associate any needed behavior with the field. The two standard Progress Dynamics objects of this type are:

- The dynamic Combo, which displays the list of valid choices as a drop-down list
- The dynamic Lookup, which can display an arbitrarily large set of possible values in a separate Lookup window with its own browser and filtering capabilities

These are *dynamic* objects because, in each case, an instance of the SmartDataField procedure supporting the object is added to the Viewer's field list. All the particulars of the field in that Viewer are entered into a property sheet and stored in the Repository database.

Thus, you need only one actual source procedure to represent all Combo objects in a Progress Dynamics application, and you need only one for all Lookup objects. These SmartDataField source procedures for these objects are dyncombo.w and dynlookup.w, in the src\adm2 directory.

There is also a set of static Combo objects that you can use with Progress Dynamics. However, in almost all cases, the dynamic Combo provides the same capabilities and more. In addition it avoids the need to actually build and save individual procedure objects representing every different field that could have a combo box list for it. You can learn more about static Combos in Progress Dynamics reference documentation. This chapter focuses strictly on the dynamic objects.

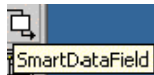
NOTE: When using a static viewer, do not alter combo boxes and selection lists to use a delimiter other than the comma.

7.1.1 Defining and using dynamic combos

The dynamic Combo choice is appropriate if the total number of possible valid values to choose from is limited to around 50 or fewer, depending on your user interface preferences. More values than this will not fit easily into the drop-down list of the combo box visualization. For a larger set of values, use the dynamic Lookup.


Once you have built a static Viewer, you can easily add one or more dynamic Combo objects to it. For example, follow these steps to replace the **SalesRep** field in a Viewer with a Combo with a list of all SalesReps:

- 1 ♦ Open the Viewer in the AppBuilder.
- 2 ♦ Right-click on the SmartDataField icon on the AppBuilder palette:



A pop-up menu displays that lists the predefined objects of this type.

- 3 ♦ From this list, select the Dynamic Combo.

The mouse cursor changes to look like the icon  for a fill-in field.

- 4 ♦ Position the cursor over the field you want to replace, and left-click. The field's visual representation is replaced by a field that looks like a drop-down list, with a ventilator icon on the left:

The screenshot shows a window titled "SmartDataViewer - custstview.w". It contains a form with the following fields:

- Cust Num: 0
- Name: [text box]
- Address: [text box]
- City: [text box]
- Country: USA (highlighted with a drop-down arrow)
- Postal Code: [text box]
- Credit Limit: 1,500
- Balance: 0.00
- Terms: Net30
- Discount: 0%
- Fax: [text box]
- Phone: [text box]
- Email: [text box]
- Contact: [text box]
- Comments: [text area]

Immediately following this change, the Choose Existing SmartDataField dialog appears:

The screenshot shows a dialog box titled "Choose Existing SmartDataField (DynCombo)". It has a menu bar with "File", "Window", and "Help". Below the menu bar is a "Details" tab. The "SmartDataField Name" is set to "ComboCategory". There is a "Create New SDF" button. The main area contains two lists:

Object filename	Object description
ComboCategory	
dcSDOProdMod	Product Module combo
dcDLPPProdMod	Product Module Data Log
dcDIFProdMod	Product Module Data File
dcSDBProdMod	Product Module SDB
dcSDVProdMod	Product Module SDV
CbLoginCompany	Login Company Combo -
cbSCMTool	SCM Tool Combo - Secu
cbUserCategory	
ComboNationality	Nationality Combo SDF
ComboLanguage	Language Combo SDF
ScmXrefOEMCombo	OEM DynCombo for gsm
dyncombo.w	Dynamic Combo
comboEntityMnemonic	Combo - Entity Mnemonic
afcommentcatcombo	A Category Combo used i
SalesRepCombo	Salesrep Combo

Follow these guidelines as you write the query:

- The query must start with the FOR EACH keywords.
- The query must **not** end in a period or colon.
- The query must **not** include the END statement.
- In the simplest (and probably most typical) case, the appropriate query definition is:

FOR EACH <table-name> NO-LOCK

- The query should always include the NO-LOCK keyword because it will be reading values for selection only.
- The query can join to as many tables as you want.
- You can place the tables in any order.
- The tables can contain any WHERE clause you require.
- The query can also contain a BY clause if necessary.
- The query should include any tables whose field values you want to display in the combo box.
- Table names should not contain a database prefix.

When you have finished typing in the query, choose **Refresh**.

The following other items in the property sheet are filled in from the database schema information for the field:

- **Query tables** — This read-only field is derived from the query you typed. It is a list of the database tables used in the query.
- **Browser fields** — This browser shows all the fields in the query tables.

- **Field name** — This read-only field shows the name of the field whose value will be assigned when a user selects an entry from the combo box list. This is the field you originally placed on the Viewer and replaced with the dynamic Combo.

CAUTION: When an SDF is on a datafield in a viewer, the name of the SDF and the datafield must match or the viewer will not function correctly. While you cannot change the fieldname here, you could do so by accessing the `FieldName`.

- **Key field** — This is the field whose value you should assign to the External Field on completion of selecting a value in the Combo. You can use any of the fields from any of the buffers in the specified query, but the field you select should have the same data type as the external field.
- **Field label** — This is the label for the dynamic Combo object in the Viewer. It is initialized to the label of the Key Field, but if needed you can change it to some other sensible label for the External Field.
- **Datatype** — The data type of the selected Key Field. This is a read-only value.
- **Format** — The format of the selected Key Field. This is a read-only value.
- **Field width** — Specify how wide this instance of the SDF should appear in the viewer.
- **Build sequence** — Defines the order in which objects are initialized at runtime in a frame or window,
- **Enable Field** — Specifies whether this instance should be enabled for input.
- **Display Field** — Specifies whether this instance should be displayed when the Viewer is displayed. When a fill-in widget is dropped onto a viewer, the Display toggle is checked by default. Remove the Display toggle to use the field as a parent field. When checked, Display causes the widget to lose its current value before it can be used as a parent field in a combo or lookup.
- **Sort** — Specifies whether the values in the combo should be alphanumerically sorted in the list.

Browser fields

It is likely that the foreign key value you assign will not be very meaningful to the user. In fact, if it is an arbitrary numeric key value such as a Progress Dynamics Object ID, you normally will not want to display it at all. For this reason, the dynamic Combo allows you to display any meaningful value you like for the user to choose. The **Key Field** is the value that is assigned regardless of what you choose to display to the user.

So from the browser containing all the available fields in the query tables, you can select one or more whose values you want to display as a single concatenated string in each combo box entry. Choose fields that will have a meaningful descriptive value for the user. To select a field, click in the **Display Sequence** cell for that field, and change the 0 to a sequence indicating the order in which you want the field values displayed (so 1 for the first, 2 for the second, etc.).

Once you have entered a display sequence for one or more fields and tabbed out of the field browser, the list of fields you have chosen is displayed as the **Description Fields**, and a default **Description Substitute** is shown.

Details tab options

This tab provides information and configuration options for the Key field selected in the browser.

Description substitute

If you select more than one field to display in the Combo entry, the **Description Substitute** field is enabled and displays a default substitution string to be applied to the values. The default is &1 / &2. This string means that the first **Description Field** will be shown, followed by a slash, followed by the second field. You can edit this string if you want to use a different string to separate the values. You can also change the order of the fields by changing the numeric sequence of the substitution arguments in the string (&1 and so on).

Field label

This is the label of the displayed field to show on the Viewer.

ToolTip

Enter a ToolTip that will help the user understand what value is being requested here.

Inner lines

In this field, specify the number of inner lines you want associated with the Combo; that is, the number of choices visible when the Combo is in the drop-down position. The default is 5. If the number of inner lines is less than the total number of items in the Combo's list, the user will have to scroll down the list to see all the items.

External field data type and format

These display-only fields on the right side of the property sheet show the data type and the display format of the **External Field**. It's possible that the Key field may not be the same as the external field. Although you can select any field for the Key field, you must ensure that you choose one with a matching data type of the external field.

Field width

This fill-in on the right side of the property sheet shows the width in character units of the **Displayed Field** in the Viewer. Normally, you can easily modify this value visually, by using the grab handles on the widget itself in the Viewer design window. If you need more precise or standardized sizing, you can set this fill-in.

Build sequence

If you have multiple dynamic Combos that have a parent-child relationship and you need to have a value in one Combo before you build data for the other, then you must set the Build Sequence field for each Combo. The Build Sequence indicates the order in which data for the Combos must be retrieved. The following example shows how to specify the **Parent Fields** and **Parent Filter Query** for the dependent Combo in this kind of parent-child relationship. The Build Sequence for the child Combo is set to 2; and the Build Sequence for the parent Product Combo is set to 1, to ensure that the parent data is populated prior to the child data.

Other tab options

In some cases, you may want a combo that gives you an **All** or **None** choice. The ComboType radio set on the Other provides this option. Data only, the default, specifies a normal combo without a All or None entry. **None** means that the user is effectively defining a record where the foreign key dependency is simply not established. Whether this is a valid option depends on the referential integrity rules of your own application. If it is valid, for example, to create a Customer record without assigning it to a SalesRep, then **None** is a valid choice. If your business rules forbid this, then it would not be. Since the combo box widget does not support a blank as a valid value for the visual item list, and since it can be visually awkward to select a blank value from the list, the dynamic Combo lets you put a value in the list that is visually represented by the string **<None>**. You can then map it in the **Default Value** field (described below) to any appropriate value for the database field.

Likewise, where the choice list represents a filtering mechanism of some kind, **All** might be a logical choice to indicate that no filtering is to be done. For example, many of the screens in the Progress Dynamics tools themselves (which are largely built in the framework) provide Combos used to filter a selection, for example to filter objects by their Product Module or to associate a user login with a particular Login Company. If you want to see objects in all Modules or if you want to log in with the privileges of all companies, then you can provide **<All>** as an option in the choice list and map it to a meaningful value for the underlying field.

Select **<None> and Data** if you want the value **<None>** added to the choice list. Select **<All> and Data** if you want the value **<All>** added to the choice list. Otherwise select **Data Only**.

Default value

If you select **<None> and Data** or **<All> and Data** for the **Extra Options**, then the **Default Value** field is enabled. In this field, enter the value that you want mapped to that option. You can use any value your application interprets as the default value in the **Key Field**, representing All or None.

Parent fields

In some cases you might have multiple Combos in a single Viewer that represent a parent-child dependency. Products and Product Modules are an example of this kind of relationship. If there were a Region field in the Customer table, and SalesReps were organized by Region, then the user might select a Region for a Customer in one Combo and then select from a separate Combo a Sales Rep from that Region. The list of valid Modules in the first example must be created after the Product is chosen, since Modules are organized by Product. Likewise the list of valid choices for Sales Rep must be created after the Region is selected. The dynamic Combo gives you the ability to modify the query for the child Combo in this kind of relationship, so that you get the right list of values.

If you are defining a child Combo in a parent-child relationship, you can enter the name of the key fields in the parent that should be used to filter the query for the child Combo. The parent does not need to be another Combo. It could be a Lookup, or it could simply be an ordinary field or fields in the Viewer. There is no need for the parent values to be values for database fields. They are just field names in the Viewer. If you use more than one parent field, separate them with commas.

NOTE: If a Dynamic Combo's parent field is an object other than another Dynamic Combo, you should run the RefreshChildDependancies procedure in the Dynamic Combo to let it know that it should refresh its contents because the value of the parent field might have changed. This procedure engages the parent filter query on the Combo. Any other Dynamic Combo does this automatically when it is a parent field to another Dynamic Combo on the viewer. Run the RefreshChildDependancies procedure in the handle of the Combo whose query you want to change. Pass in the name of one of its parent fields or the one whose value has changed. If your Combo's parent filter query is dependent on more than one field, you only need to pass in one of its parent fields and it will fetch the values of the rest automatically.

Parent filter query

If the Combo is the child object in a parent-child relationship, then you must enter the query phrase that will be added to the child's query to filter the records based on the parent values. If you define values for **Parent Fields**, enter the query phrase in the **Parent Filter Query** field, using Progress substitution arguments (&1, &2, etc.) to represent the **Parent Fields**.

The query string must be a syntactically correct Where-clause phrase, but not a complete query statement. So it might not contain the words WHERE, EACH, FIRST, LAST, NO-LOCK, SHARE-LOCK, or EXCLUSIVE-LOCK, or a comma (,) or colon (:). Since you cannot check the syntax at design-time, it is important that you test this query properly. If the query fails at run time due to any syntax error, the Combo will simply be blank. In such a case, you must check the AppServer log file for an error message.

Functionality has been added to the SmartDataField Maintenance tool for Dynamic Lookups and Dynamic Combos to allow the developer to specify the **parent filter query** with a pipe (|) delimiter to indicate where that string should be applied.

```
FOR EACH customer NO-LOCK, FIRST order OF customer NO-LOCK
```

In the above example, the parent filter query would be applied after the joined table. To force this to be placed before the first join the Parent Filter query should look as follows:

```
Customer.CustNum = INTEGER('&1') |
```

Only the pipe (|) added at the end tells the program to place the filter string as part of the first table. If we would change the Parent filter field as follows:

```
|Customer.CustNum = INTEGER('&1')
```

With the pipe (|) at the front, the program will add the filter string after the second table join. If you do not specify a pipe (|) it will always add it after the last joined table. This is to ensure backward compatibility. When using the pipe(|) delimiter in the filter string, you no longer have to specify **AND** as a joining operator between to filter strings, unless that belong to the same table.

The code looks like the following:

```
Customer.Custnum = INTEGER('&1') | Order.OrderDate = DATE('&2')
```

This is resolved at run-time in the following way:

```
FOR EACH Customer WHERE Customer.CustNum = INTEGER('1'), FIRST Order OF  
Customer WHERE Order.OrderDate = DATE('01/01/2001') NO-LOCK
```


7.1.2 Saving the combo

After you have completed your dynamic Combo, choose **Save** to register its properties and return to the Viewer design window. When you save the new dynamic combo, you are creating a master and an instance whose properties match.

7.1.3 Runtime example

Figure 7–1 shows the example Customer Update window at run time with the SalesRep Combo.

Figure 7–1: Example of run time Customer Update window

7.1.4 Defining and using dynamic lookups

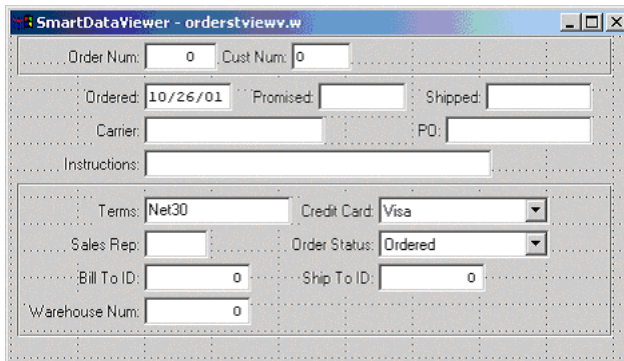
The dynamic Lookup object is parallel in many ways to the dynamic Combo, providing an alternative visualization of the list of valid choices for a field. The lookup is the best choice when you need to display many possible choices. The Lookup brings up a separate window where a browser with a list of values displayed. Given the filtering and record batching capabilities of Progress Dynamics, a user can select a value efficiently from a list of any size.

Because the Lookup is similar to the Combo in many respects, it is described here as a variant, so as not to repeat all of the property sheet field descriptions and other details. So be sure to read the [“Defining and using dynamic combos”](#) and [“Saving the combo”](#) sections before reading this section. Unlike the Combo, which does have static counterparts, there is no standard static Lookup object in Progress Dynamics. All of this description applies just to the dynamic object (simply referred to as a Lookup).

Follow these steps to add a Lookup to a Viewer:

- 1 ♦ Right-click on the **SmartDataField** icon on the palette and select **Dynamic Lookup** from.
- 2 ♦ Position the cursor over the field to replace by a Lookup, and select with a left mouse click.

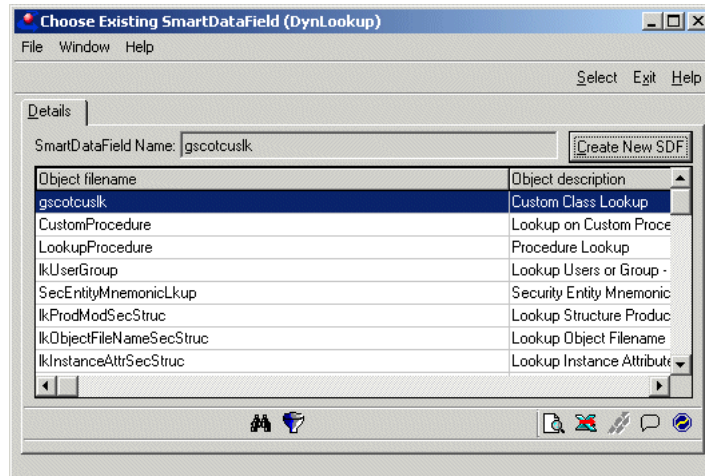
This example builds an Order Viewer for the Sports2000 database. The user must assign each order to a valid existing Customer. The Lookup is on the **CustNum** field so that the user can select a Customer by browsing through a list that includes the **Name** and other useful fields. The Viewer displays the **CustNum** field. There is some space to the right of **CustNum** field where the **Lookup** button and **Customer Name** field can be shown:



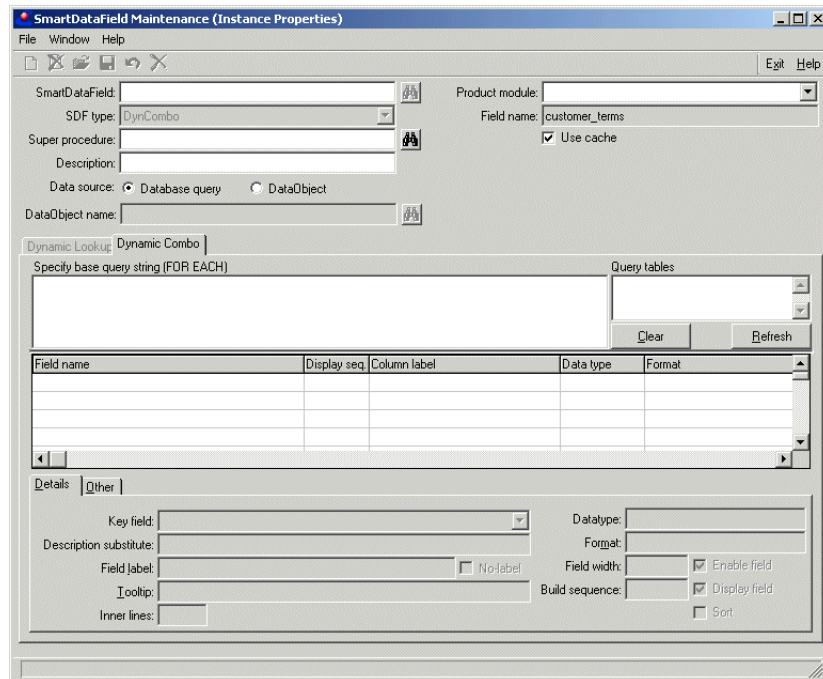
The screenshot shows a window titled "SmartDataViewer - orderstvieww.w". It contains a form with the following fields and controls:

- Order Num: Cust Num:
- Ordered: Promised: Shipped:
- Carrier: PO:
- Instructions:
- Terms: Credit Card:
- Sales Rep: Order Status:
- Bill To ID: Ship To ID:
- Warehouse Num:

When you drop the Lookup onto the **CustNum** field, the Choose Existing SmartDataField dialog appears:



- 3 ♦ You can select a lookup from the list, or choose **Create New SDF**. The SmartDataField Maintenance utility appears:



- 4 ♦ Fill in the top fields, enter the Base Query String, and choose **Refresh**. The query can reference up to 10 tables. This step initializes the lower portion of the Dynamic SmartDataField Maintenance window with defaults for many of the fields, based on the database tables in the query. The filled-in Dynamic SmartDataField Maintenance window for a Customer Number Lookup looks like this:

The following subsections describe fields and values in the Dynamic SmartDataField Maintenance window that are different from those for the Combo.

Rows to batch

An attribute of Lookups not relevant to the Combo is the number of database rows to send across from server to client in one batch. If there are many possible values for the foreign key field the Lookup represents, it would not be practical to populate a client-side temp-table with all of them before displaying the list to the user; this might take ages. So records are loaded into the temp-table and sent to the client in batches. The default value of 200 is somewhat arbitrary, but a reasonable starting figure. If you expect the user typically to filter the Lookup value list before choosing a value, then you might set this number to be significantly smaller than 200, since filtering the list repopulates the Lookup browser. Decreasing the **Rows To Batch** value improves run time performance somewhat.

On the other hand, if you know that the total number of possible values (total number of Customers in the Customer table, for example) is a number greater than but not too much greater than 200, then you might set **Rows To Batch** high enough to retrieve them all at once. Otherwise, the framework retrieves one batch of rows, and other batches as needed, if the user scrolls around or repositions in the list.

Browser fields

As with the Combo, the Dynamic SmartDataField Maintenance window shows a list of all fields in the tables in the Base Query. The list of fields in this browser, and how you use them, is somewhat different for the Lookup.

As with the Combo, you can select one or more fields to display to the user. In the case of the Lookup, these fields will be columns in a browser. To select a field to be added to the browser, give it a Browser Sequence number. The example in [Step 4](#) above has been sorted (by clicking on the **Browser Seq.** column label) to show that five fields are selected to display in the browser. Because the browser is displayed in a separate window, and because the user can scroll it horizontally, you can add as many fields to the browser as you like., Keep in mind that this is a mechanism for selecting a foreign key value, so you should choose fields that would help the user select the right record quickly.

NOTE: The browser is based on a temp-table built dynamically from the field names selected. If your Base Query joins two or more tables, you cannot include more than one field in the Lookup browser with the same field name but a different table name.

Linked fields

In addition to showing fields from the related table in the Lookup browse window, you can elect to display one or more of these fields in the Viewer itself. This makes the Viewer's display more effective. It also causes the related descriptive fields to be seen as soon as the Viewer is entered, not just when the Lookup is used to set or change the foreign key field value.

To select a field as a linked field, click in the **Link Field** cell for that field name and change **NO** to **YES**. This example uses the **Customer.Name** field to display in the Viewer with the value taken from the Lookup browser.

Linked widgets

Normally, for each **Linked Field** you select, you will want to map that field to a local variable in the Viewer where the field's value will be displayed. Enter the name of the local variable in the **Linked Widget** browser cell for that field. Then when you return to the Viewer design window from the property sheet, you must define that fill-in in the Viewer by following these steps.

- 1 ♦ Select the **Fill-In** icon from the AppBuilder palette.
- 2 ♦ Drop it onto the Viewer where you want it to be displayed.
- 3 ♦ Set the **Object** field in the AppBuilder toolbar to be the variable name you entered in the property sheet.
- 4 ♦ Enter an appropriate **Label** for the field.

In the example, a character variable is defined, called `cCustomerName`, and placed next to the `CustNum` Lookup.

NOTE: If you do not define a Linked Widget for a **Linked Field**, then you have the responsibility of intercepting the assignment of the **Linked Field** value and handling it yourself. To help you do this, Progress Dynamics defines several standard hooks to which named events your Viewer can subscribe. In this case, you would likely put the code to handle the value in a custom internal procedure that you would write called `LookupComplete`. You would only need to do this if your Lookup required special formatting or some other kind of special handling of the **Linked Field** value.

Override label and override format

You can change the column label and display format for the fields selected to be shown in the Lookup browser.

NOTE: If you translated the label using the Progress Dynamics Translation utility, you cannot override the translated label here.

Displayed field

As for Combos, the **Key Field** is chosen to be the field whose value is assigned to the external field on completion of the lookup operation. You can choose another field value to show in the Viewer in place of the **Key Field**. This field is called the **Displayed Field**. The **Displayed Field** can be the same as the **Key Field**, but it can also be different, if the **Key Field** is not a meaningful value to display to the user. In the example, the Order Viewer displays the **Customer Name** as a Linked Widget **in addition** to the **Customer Number** as **Displayed Field**. If you wanted to eliminate the display of the **Customer Number** altogether, you could designate the **Name** as the **Displayed Field** and not bother defining a Linked Widget and variable name for it at all.

The Lookup supports entry of the **Displayed Field** for resolution of the Lookup without calling up the browse window. It also does auto-completion of a unique partial entry for the field. You should use a sensible indexed field as the **Key Field** wherever possible.

Parent field and parent filter query

These entries allow you to define a parent-child relationship in a Viewer. Some other **Lookup**, **Combo**, or other field is expected to provide one or more parent values used to filter a dependent child lookup, to display only values valid for that parent value. This works exactly the same as for Combos.

Browser title

This is the title to be displayed in the **Lookup** browse window. It should describe the data that is being looked up, for example “Customer Lookup.”

Maintenance SDO and maintenance object

In some cases, you might want to allow the user to enter a missing record in the parent table of this foreign key relationship, or to edit an existing record, directly from the **Lookup**. For example, if the user was entering an **Order for a Customer** and discovered that the **Customer** record had not yet been created, you might want to allow the user to create it here, without the user having to bring up a separate **Customer Maintenance** window from some other menu.

If this is the case, you must enter the name of the SDO used to maintain that parent table in the **Maintenance SDO** field. You must also enter the name of the maintenance window to launch to create or edit the record in the **Maintenance Object** field.

If you fill in these fields, then a toolbar with an **Update** button band is added to the **Lookup** browse window, so that the user can select **Add**, **Copy**, or **Edit** to change the parent table on the spot.

When you exit the property sheet, you must remember to define the variables for any Linked Widgets you defined for the Lookup. In the example shown in [Figure 7–2](#), a variable is defined for the **Customer Name** field, and placed next to the **CustNum Lookup**. The label for the **Name** field is not included, so that the fill-in is displayed right up against the **Lookup**. This is strictly a visual choice.

The screenshot shows a window titled "SmartDataViewer - orderstview.w". It contains several input fields and dropdown menus arranged in a grid-like fashion. The fields include:

- Order Num: 0
- Cust Num: (with a small icon next to it)
- Ordered: 10/26/01
- Promised: (empty)
- Shipped: (empty)
- Carrier: (empty)
- PO: (empty)
- Instructions: (empty)
- Terms: Net30
- Credit Card: Visa (dropdown)
- Sales Rep: (empty)
- Order Status: Ordered (dropdown)
- Bill To ID: 0
- Ship To ID: 0
- Warehouse Num: 0

Figure 7–2: SmartDataViewer example

Changing labels in viewers

You can change labels for fields in a SmartDataViewer in several ways:

- Use the Data Field Maintenance tool to make a global change to the label. Changes made in this tool will not be immediately reflected in open viewers. Close and reopen the viewer to see your changes.
- Override the data field master with the Override Label property. Changes made here apply even if the data field master is changed at a subsequent time.
- Right-click the field and choose Edit Datafield Master. Changes made in this way will take effect only if there are no overrides on the field.

7.1.5 Using the SmartDataField Maintenance tool

The preceding discussion and examples have shown you how to replace a field in a Viewer with a dynamic Combo or Lookup object. You might also want to define Combos and Lookups out of the context of a Viewer, because you want to place independent of any particular Viewer.

The SmartDataField Maintenance utility allows you to edit the master or instance properties. To edit the master properties, you can use the Open Object dialog to select and open the SDF. The SmartDataField Maintenance utility appears with “Master Properties” showing in the title bar. You can also select an SDF instance in a viewer, right-click, and select **Edit Master**.

To edit the properties of an instance, select an SDF instance in a viewer, right-click, and select **Instance Properties**. The SmartDataField Maintenance utility appears with “Instance Properties” showing in the title bar.

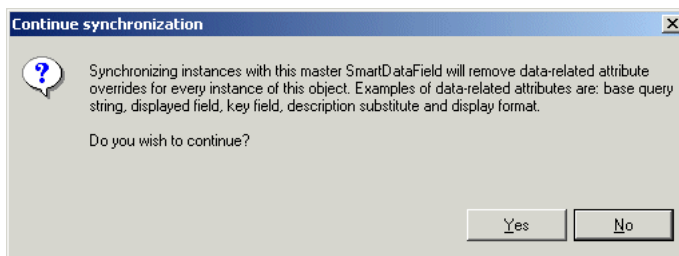
This tool presents you with all the same elements in the **Combo** and **Lookup Properties** dialog boxes, and you can create and edit dynamic Combos and Lookups here in the same way as described earlier.

NOTE: When you want to edit SDF properties from a container, right-click and choose Instance Properties or Edit Master. Do not open the property sheet and click Edit. Use care when using Edit Master. If you change a property for the master, then that value will be applied to all instances currently open in AppBuilder even if those instances have overrides. Close all instances that you do not want affected before using this feature.

Where Used and Synch instances buttons

The Master Properties view in the Smart Data Field Utility contains two features to help you manage master and instance properties. The Where Used button provides a message box with a list of the objects that contain an instance of the SDF. It is only available for SDFs that have been saved as SmartObjects.

The Synch instances button is a tool to allow you to force data-related instance properties to match the master. Click the button and the following message appears:



Click **Yes** to synch the properties.

Map fields tab

This tool allows you to create mappings between data source fields and viewer fields. It is a feature used to improve performance. See the performance section of the [Progress Dynamics Administration Guide](#) for more information.

Other tab

On the Other tab, the lookup has some different options:

- **Maintenance SDO** — If the user can create or edit a record in the parent table from the Dynamic Lookup, enter the name of the parent maintenance SDO. A toolbar with an update button band is added to the Dynamic Lookup window. Choose the lookup button to display a list of available SDOs.
- **Maintenance Object** — The name of the maintenance window object to launch when the user creates or edits a record in the parent maintenance SDO. Choose the lookup button to display a list of available objects.

NOTE: Only containers that contain a viewer and toolbar without an SDO are valid containers for this functionality. When creating this kind of object, use the 'rywinFolder' object as a template.

Notice the set of check boxes on the right. When a dynamic lookup results in more than one matching entry, a dialog box automatically appears to allow the user to select the desired matching value. This automatic behavior can now be suppressed or modified with the following attributes:

- **PopupOnAmbiguous** — If YES, the dialog box pops up on LEAVE of the field when the typed value matches more than one record. The default value is YES. If set to NO, the value is left as typed in or it is blanked out (depending on how the BlankOnNotAvail property is set).
- **PopupOnUniqueAmbiguous** — If set to YES, the dialog box pops up on LEAVE of the field when the typed value uniquely matches a record, but the search returned other non-unique matches. The default value is NO.

For example, suppose a user types Smith and the search logic is a BEGINS query. The application returns the following:

- Smith
- Smithers
- Smithson

With the default behavior of NO, one, and only one, record matched the typed value exactly. Dynamics assumes that it has found a unique match and does not pop up a browse to display the other potential matches. If you would like Dynamics to pop up a dialog in this case, override this property and set it to YES.

- **PopupOnNotAvail** — If set to YES, the dialog box pops up on LEAVE of the field when the typed value matches no records. The default is NO. If set to NO, the value is left as typed in or is blanked out (depending on what how BlankOnNotAvail property is set).
- **BlankOnNotAvail** — Controls whether or not an invalid value should be blanked from a lookup on LEAVE of the field.

Map fields tab

This tool allows you to create mappings between data source fields and viewer fields. It is a feature used to improve performance. See the performance section of the [Progress Dynamics Administration Guide](#) for more information.

7.1.6 Running a window with a lookup

If you run the window containing the example Order Viewer, the Customer Number key appears, with the Customer Name field alongside it, as shown in [Figure 7–3](#).

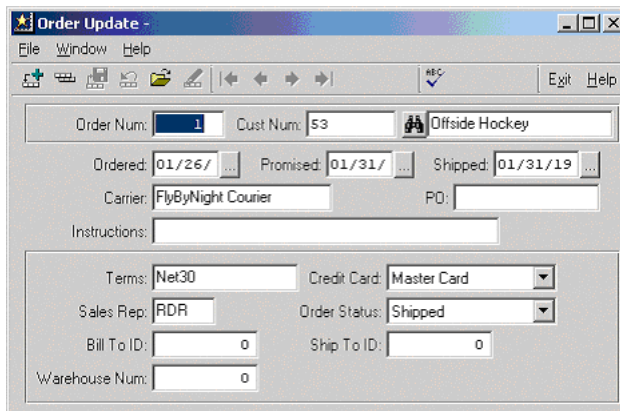
The screenshot shows a software window titled "Order Update -". It has a menu bar with "File", "Window", and "Help". Below the menu is a toolbar with icons for adding, deleting, and navigating records, along with a "REC" button and "Exit" and "Help" buttons. The main area contains several input fields: "Order Num:" with value "1", "Cust Num:" with value "53", and a dropdown menu showing "Offside Hockey". Below these are date fields: "Ordered:" (01/26/), "Promised:" (01/31/), and "Shipped:" (01/31/19), each followed by a calendar icon. There are also text fields for "Carrier:" (FlyByNight Courier) and "PO:". An "Instructions:" text area is below that. A section at the bottom contains "Terms:" (Net30), "Credit Card:" (Master Card dropdown), "Sales Rep:" (RDR), "Order Status:" (Shipped dropdown), "Bill To ID:" (0), "Ship To ID:" (0), and "Warehouse Num:" (0).

Figure 7–3: Order Update window

The Lookup provides several different ways to identify a new key value.

First, if the user enters a correct key value or partial key value in the **CustNum** field, the Lookup verifies the value, completes it if necessary, and displays its associated **Linked Fields**. If more than one record matches the partial value entered, then the Lookup automatically launches the browse window and filters the values down to the value entered. If no value matches what is entered, then the key value and linked fields are empty to show that the value entered is invalid.

Alternatively the user can choose the binoculars button or press the **F4** keyboard shortcut to launch the Lookup browse window. If there was already a value or partial value in the key field, the browser comes up filtered for that value. This means that if the user is editing an existing record, and chooses the Lookup button or presses **F4** without first clearing the key field or typing a partial new value into it, the browse window comes up filtered to display only the record with the current value. To change this, the user must select the **Filter** tab and blank out the values in the key field, or enter another appropriate filter. This behavior might be modified in the future so that an existing value that has not been changed will not automatically filter the browser values.

When the Lookup browse window appears, as shown in [Figure 7-4](#), it lists the records matching the filter or partial value, if any. Otherwise, it shows the unfiltered value list. The browser shows all the fields you picked as **Display Fields** when you defined the Lookup.

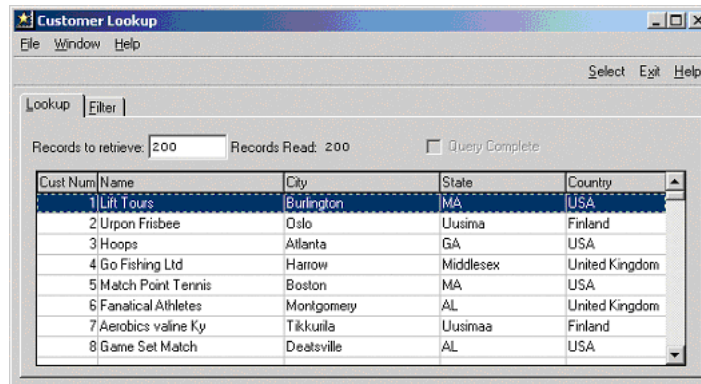


Figure 7-4: Customer Lookup

The setting for **Rows To Batch** from the Lookup definition is shown in this window as **Records to retrieve**. If the entire data set has been retrieved (with or without a filter applied), then the size of the data set is displayed in the **Records Read** field and the **Query Complete** toggle box is selected. If there are more records in the database matching the current query than have already been retrieved, the **Query Complete** toggle box is cleared. In this case, when the user scrolls down to the bottom of the current list or chooses the **Filter** tab and enters a reposition **From** value, the framework retrieves another batch of records from the server. If users want to change the size of the batch, they can do so in the **Records to retrieve** fill-in field. A user might, for example, set this to a high number to force retrieval of all remaining records at once so that they could be sorted on the client. Using a high number might cause a significant delay in the filling of the next batch.

The browse window is resizable, to show more rows at a time or to show more of the columns in the browser without scrolling. The user can also resize individual columns to display data more efficiently. The user can sort on any column by clicking on the column heading. This sort is done on records already retrieved from the database, so if the **Query Complete** toggle box is not selected, the sort will not be complete. This is why a user might want to force retrieval of all possible records before sorting, if this is really necessary to locate the right record.

To select a record, the user can position to the row and press **ENTER**, double-click the row, choose the **Select** toolbar button, or key the shortcut **ALT-S**.

The **Filter** tab shows a list of all the fields in the Lookup browser, in the order they appear in the browser, and allows the user to pick **From** and **To** values for any field, as shown in [Figure 7-5](#).

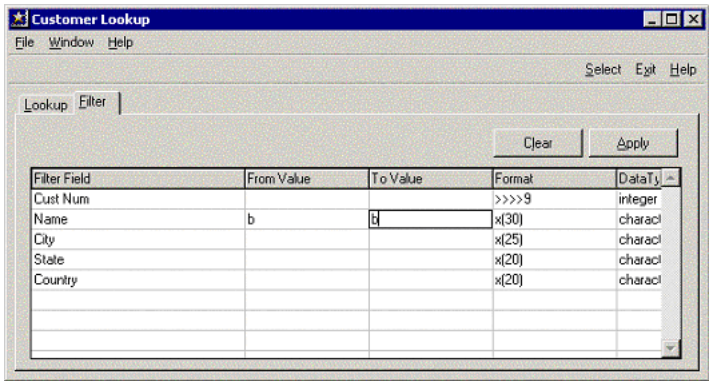


Figure 7-5: Customer Lookup window - Filter tab

The **Clear** button clears all filter values, and **Apply** applies the filter settings, reopens the query accordingly, and returns to the **Lookup** tab, where the matching values are displayed.

If the user enters just a **From** value, then the query is effectively repositioned to that value, showing all records with a value greater than or equal to the value entered. If the user enters both **From** and **To** values, then that range is filtered. To see all records with a particular value in a filter field, the user must enter the same value in both the **From** and **To** fields. A high values tag is automatically appended to any value entered as a **To Value**, so that, for example, the filter shown above, From “b” To “b”, will return a set of all customers whose names **begin with ‘B’**.

The framework uses validation checks that assure the data the user enters is of the correct data type and that the ranges the user specifies are valid.

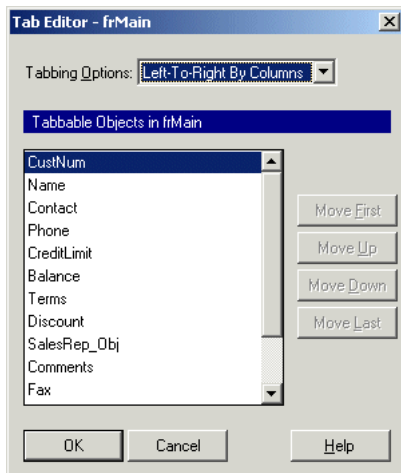
This filter capability is very similar to that offered by the **Filter** button on many Progress Dynamics tools windows. You can make the **Filter** button a standard part of any application window by using a toolbar with the **Filter** button. However, the two filter techniques differ in the following ways:

- Index information is not displayed next to the filter field names. It is your responsibility to make sure that the fields available as filter fields are sensible fields on which to sort. Otherwise, you probably should not include them in the browser to begin with. The browser fields should be fields that will help the user identify the proper record to select.
- There is no provision to save the filter information for the session, or in the Repository database as a permanent setting, as there is for standard framework filters. This is because a filter in a tool such as Entity Control, or in an application function such as Order Entry, might be relevant to save as a persistent setting. A user might only work with certain customers or regions or databases, or might want to work with a particular value for one of those types of entities for the duration of a single session. It makes sense to allow the setting to be kept. But in the case of a Lookup browser, the filter values will likely be different for every data entry operation. At least the user will not need to use the Lookup to identify an entity such as a customer that is used in a number of records entered in a row, after the entity has been located and entered once.

7.2 Viewer tabbing options

The AppBuilder supports all tabbing options available to static viewers for dynamic viewers. The tab editor can be used to change the tabbing option for the dynamic viewer and that tabbing option is stored in the AppBuilderTabbing attribute for the viewer master.

You can access the Tab Editor dialog box for any frame or dialog object by choosing Tab Order from the main AppBuilder Edit menu or by clicking the Tab Order icon in the object's property sheet, as shown below:



The default tabbing option is “Default,” which is the same for static viewers. The default value of the AppBuilderTabbing attribute on the DynView class is “Default.” The AppBuilderTabbing attribute cannot be modified with the Dynamic property sheet for a viewer, use the Tab Editor to modify it.

Tab order for fields of a dynamic viewer are adjusted in the following circumstances:

- When the tab editor is started.
- When the tabbing option is changed with the tab editor.
- When db fields are dropped onto the viewer.
- When the viewer is saved.

What this means is that if fields are dropped onto a viewer and the tab order is changed with the Tab Editor and then fields are moved, their tab order is not adjusted until the viewer is saved. This is how static viewers behave. The difference with dynamic viewers is that the tab order is visible in the Dynamic property sheet as the Order attribute and can be modified. Therefore, the tab order visible with the Dynamic property sheet may not be the tab order saved when the viewer is saved.

The Dynamic property sheet should not be used to adjust the tab order. If it is, the AppBuilderTabbing option is set to “Custom” and the tab order can only be adjusted manually. It can be set back to one of the other options with the Tab Editor, which could then change any manual changes made with the Dynamic property sheet.

7.3 Combo and lookup performance and caching considerations

This section describes how dynamic Combos and Lookups operate, so you can understand better how to use them in your applications.

The Progress Dynamics Viewer support code coordinates data retrieval for all dynamic Combos and Lookups in a given Viewer. On initialization, all the database records necessary for populating the description fields of a Combo or the linked fields of a Lookup are read from the database and returned to the client-side Viewer in a single AppServer call. No extra SDOs or other objects are needed to handle the data. When the value in a parent object changes, all the data for dependent child objects in the same Viewer will be retrieved in a single call. This represents a significant improvement over the standard Version 9 SmartSelect object, which associates each client-side Lookup or Combo visualization with an SDO that needs to run on both client and server.

You can, however, use an SDO as a data source for a dynamic combo. One advantage of this is the ability to use the client-side application data caching abilities of the SDO with the combo. See the chapter in the *Progress Dynamics Programming Handbook* for information about the client-side application data cache and dynamic combos and lookups.

As discussed in [Chapter 4, “Preparing to Build Application Objects,”](#) it can be useful to add joins to SDOs to pull in descriptive fields from related tables, as read-only fields. Even if you display those SDO fields in Viewers with Lookups and Combos that can pull in the related descriptive fields on their own, it is still valuable to have the extra fields from other tables be a part of the SDO field list. First of all, you might want to display those fields in a Browser, or elsewhere where the Lookup and Combo do not apply.

NOTES:

- To control whether dynamic combos and lookups whose datasource is a database table cache or don't cache, use the session property `field_cache_options`.
- Map fields (Map fields tab) are another feature of lookups that can be used for better performance. See the performance section of the *Progress Dynamics Administration Guide* for more information.

Old or New Version 2.1B API

Beginning in Progress Dynamics Version 2.1B, the ADM2 functions that make up the API for dynamic combos and lookups have been retooled for performance and caching improvements. The signatures have not changed, so no code changes are likely needed. You should use the new APIs by default, but s provides session properties to control this decision. See the application data caching chapter in the *Progress Dynamics Programming Handbook* for information about these session properties.

7.4 Subclassing dynamic combos and lookups

Beginning in Progress Dynamics Version 2.1B, the underlying class structure of combos and lookups changed. The change supports the performance enhancements and new client data caching scheme. (See the chapter in the *[Progress Dynamics Programming Handbook](#)* about the client data cache and caching of dynamic combos and lookups.) The old class structure had Field as the parent class and DynCombo and DynLookup as children. The new class structure inserts a new parent class called LookupField between Field and DynCombo and DynLookup.

The normal way to create custom subclasses here would have been to subclass DynCombo or DynLookup. If you did this, the new class structure should not affect your code. In the future, you should continue to subclass DynCombo and DynLookup. LookupField should not be subclassed.

Using the Progress Dynamics Container Builder

This chapter describes the tools you can use to build layout templates for windows and pages of tab folders. It also describes how to build dynamic windows from those layouts and how to assemble your individual application objects into complete windows.

This chapter contains the following sections:

- [Introduction](#)
- [Relative positioning in the containers](#)
- [Launching the Container Builder](#)
- [Using the Container Builder](#)
- [Advanced container features](#)
- [Standard toolbar objects](#)
- [Preferences](#)

8.1 Introduction

One of the basic principles of the Progress Dynamics framework is to let designers and developers create reusable behavior for different kinds of objects. The existence of standard objects such as SDOs, Browsers, Viewers, and Toolbars is a prime example. Each of these object types has a large amount of built-in functionality that is inherited by all objects of that type without any additional programming or testing on your part.

Building dynamic windows in Progress Dynamics also follows this behavior. The expectation is that you will use most dynamic visual layouts in a number of different situations, whether what is being created is a simple browse window (which could apply to many different tables), a table maintenance window (again for many tables), or a parent-child relationship between browsers in a window or between pages of a tab folder.

The layout can represent a single-page window (one with no tab folder), or a page for a tab folder window with multiple pages. In the latter case, you can later assemble a particular window from multiple individual page layouts, one for each page in the completed window.

In addition to specifying what objects appear in a window and in what relative positions, you can specify exactly what links are required to make the window or page function properly. They are then reproduced automatically for the actual objects in each window built from the template. In this way, no one but the template designer needs to understand the details of how the links operate and what links are required. Once the proper links are established, they will be created consistently for every window or page of that type. Thus the template mechanism also helps assure consistency and correctness of windows in an application, without repeating details such as links and positions each time you build a new window of the type.

The Container Builder is designed to build Progress Dynamics containers while managing the layout of objects and the links between them. A container is saved in the Progress Dynamics Object Repository as a set of related records, describing the container itself, the objects it contains, their links, and so forth. A flag in the Repository, displayed as a check-box on the Container Builder, identifies the container as a template, so that you can select it later as the basis for building a window or tab folder page. There are also Repository records with the same template flag set for the various types of contained objects that you can use as template objects in a container. The flag to identify this is the `template_smartobject` field in the `ryc_smartobject` table. There is no specific tool for defining new template objects, but setting this flag for a representative record will designate it as the template for that type. The standard Progress Dynamics Repository contains predefined template objects for the kinds of objects you are likely to need.

8.2 Relative positioning in the containers

The Progress Dynamics framework supports the creation of dynamic container windows, where many or all objects in the container can be realized at run time from data in the Object Repository. In principle, these windows are always resizable, and the objects inside them resize appropriately. The framework also supports the notion of a tab folder with multiple pages within a single window. In such a container, the tab folder visualization itself is considered to be on Page 0, meaning that it is always displayed. Other objects can also be on Page 0, for example a Toolbar to be used by objects on the different pages of the folder, or perhaps a Browser or Viewer above the folder to display summary information you always want to have visible. You can then define other pages starting at Page 1, with the page numbers corresponding to the order of the tabs on the folder. You can place one or more objects on each page and the user can view them when selecting that tab. In the Container Builder, you can define a layout for a window with no pages, for Page 0 of a tab folder window, or for a Page Layout to be displayed on a page of a tab folder. You can then assemble the pages flexibly to create a particular application window.

Progress Dynamics defines several different layout types for dynamic, resizable layouts. Previously, some of these layouts had names such as “Top/Multi/Bottom,” meaning (for example) that the container type supports an object in a top section of the window, one or more objects (on top of one another) in a central section, and an object in a bottom section. However, there is now a single, very flexible dynamic layout called “Relative,” which is used by the Container Builder. This layout type is generally the only one you need for new dynamic containers.

The Relative layout type supports a Main section that starts at the top of the window or the page. Within this section, you can place up to nine rows of up to nine objects. In addition, you can also use a separate Bottom section with a single row of up to nine objects. The Bottom section, if it exists, is always bottom-justified in the window or page. That is, if you resize the window such that there is extra space at the lower end of the Main section of the window, then the objects in the Bottom section move down to the bottom of the window, leaving empty space above them.

Each object in a window has a layout position represented by a three-character code. The first character is “M” or “B” depending on whether this object is in the Main or Bottom section. The second character is a digit representing the row, and the third character is a digit representing the left-to-right position or sequence within that row.

NOTE: Row in this case does not mean the actual coordinate of the object, as in a Row/Column designation, but simply the relative position of the object from top to bottom.

The third character can also be a “C,” meaning that the object is centered within the row (or within the remaining available space in the row) or “R,” meaning that the object is right justified within the row. SmartDataObjects and other nonvisual objects are automatically assigned to row 0, since they do not actually appear in any way at run time.

For example, a window or page with a Toolbar at the top, and an SDO, followed by a Browser, under that a Viewer, and another Toolbar at the Bottom, would assign to those objects the layout positions “M11” for the toolbar, “M01” for the SDO, “M21” for the Browser, “M31” for the Viewer, and “B11” for the Bottom Toolbar.

If you want the Viewer to be to the right of the Browser rather than underneath it, change its position to “M22.” The Container Builder assists you in assigning these positions to objects in the layout template. The objects in an actual window you build from that template are placed in the same relative positions.

Each object, depending on its object type, can be resizable in the horizontal dimension, in the vertical dimension, or both. For example, a Browser is normally resizable both horizontally and vertically (though it, like all visual objects, has the properties `resizeHorizontal` and `resizeVertical` where this can be set differently for a specific object). A Viewer is normally not resizable at all. A Toolbar is normally resizable horizontally but not vertically. All objects also have an initial size, which has a default value for each object type. This means that when a window is initially instantiated for a user, the Layout Manager determines the minimum size for the window based on the initial size of each object inside it. The window initially displays with the minimum size. If you resize the window by dragging one of its corners or edges, the Layout Manager recalculates the size of each of its contained objects. A Toolbar stretches horizontally but not vertically. A Browser stretches in both dimensions unless its properties are set otherwise. A Viewer remains the same size.

If there are resizable and fixed-height objects in the same row, the resizable objects stretch vertically to match the height of the tallest fixed-height object in the row. If the resizable object is initially already taller than the fixed-size object, it remains fixed at its original height. If there are multiple horizontally resizable objects in a row, they share equally the available space after fixed-width objects are accounted for. For tab folder pages, the Layout Manager uses the available space within the tab folder to allocate each of the pages. The size of the tab folder itself, its Toolbars if there are any, and any other objects that have been placed on Page 0 are taken into account before allocating other pages within the tab folder. Because Progress Dynamics creates the objects on each page of a folder only when that page is first selected, it might automatically resize a tab folder the first time a user selects its pages.

All of this is handled automatically and allows you to make the most effective use of the available screen real estate. It is a standard Progress Dynamics feature that the latest size and position of each window is saved as a user preference, so that the next time you open that window, it will have the same size and position as before.

8.3 Launching the Container Builder

The Container Builder can be launched from the **Build** menu on the AppBuilder or the Development console, or by opening a container itself from the **File→Open** or **File→New** menu options.

8.3.1 Launching the Container Builder from the Build menu

When you launch the Container Builder from the Build menu on the Appbuilder or from the Progress Dynamics Development menu, it launches in to the default **open** mode. The **Add** button enables to create a new container. The **Container** frame opens an existing container. If the **Add** button is selected, the Container Builder changes to **add** mode. The **Save** button and the **Cancel** button are then enabled to add a new container or cancel the add action, respectively. [Figure 8–1](#) shows the top of the Container Builder in **open** mode.

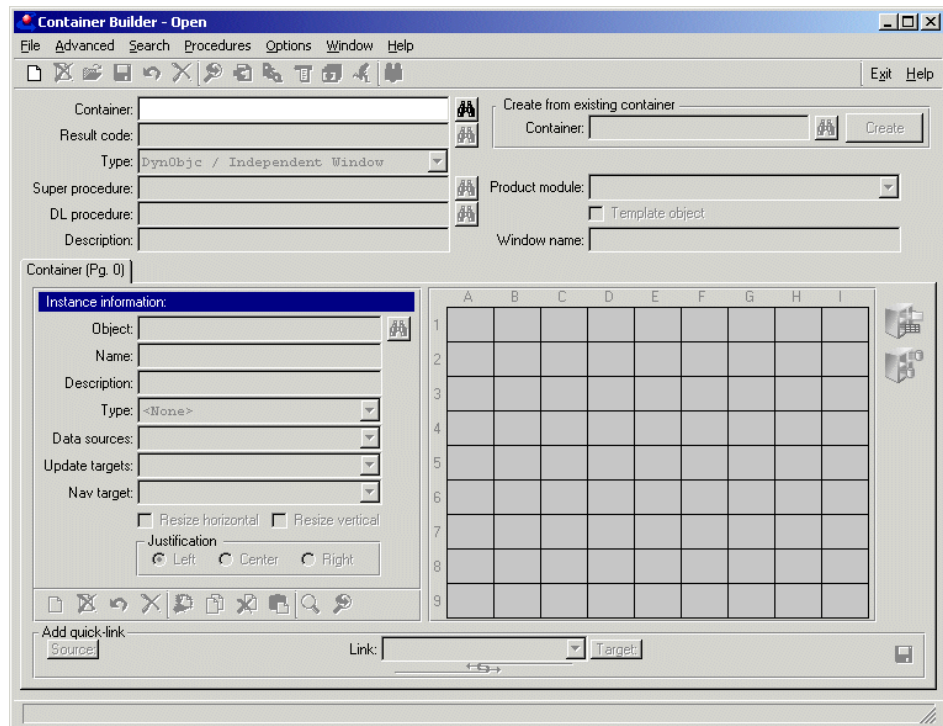


Figure 8–1: Container Builder - open

Figure 8–2 shows the Container Builder in **new** mode.

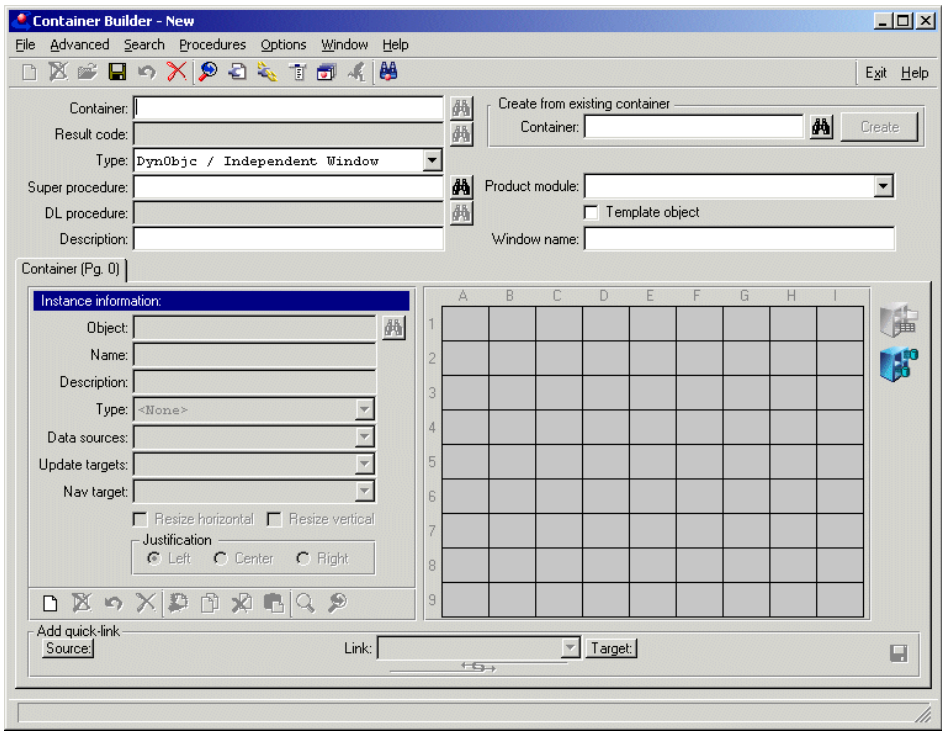


Figure 8–2: Container Builder - new mode

Figure 8–3 shows the Container Builder displaying an existing container.

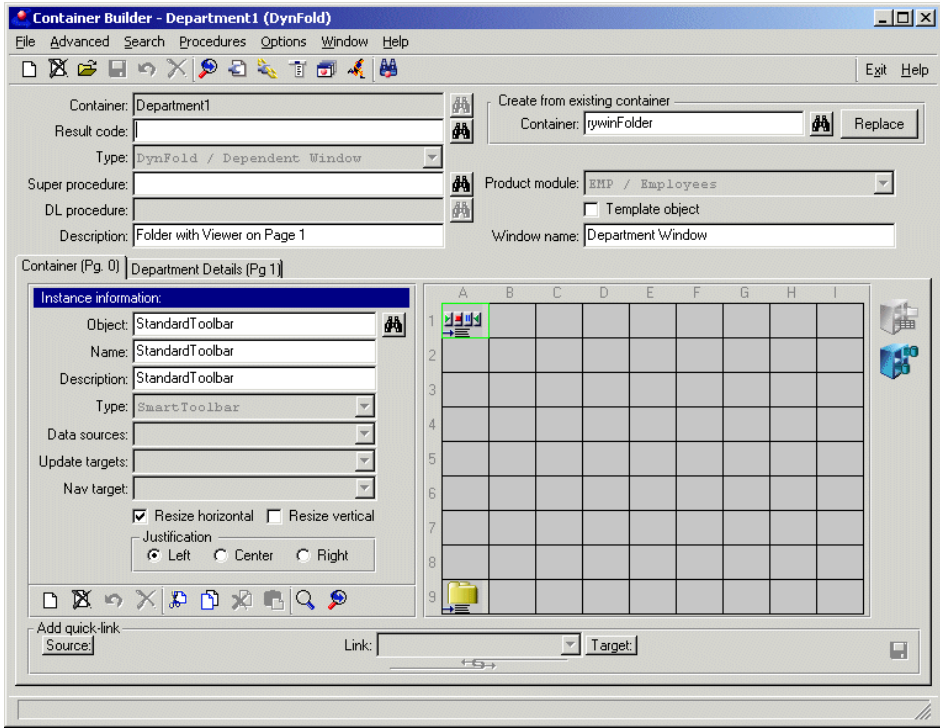


Figure 8–3: Container Builder - existing container

8.3.2 Creating containers using the AppBuilder

When you select the Appbuilder option **File→New**, the window shown in [Figure 8–4](#) is invoked.

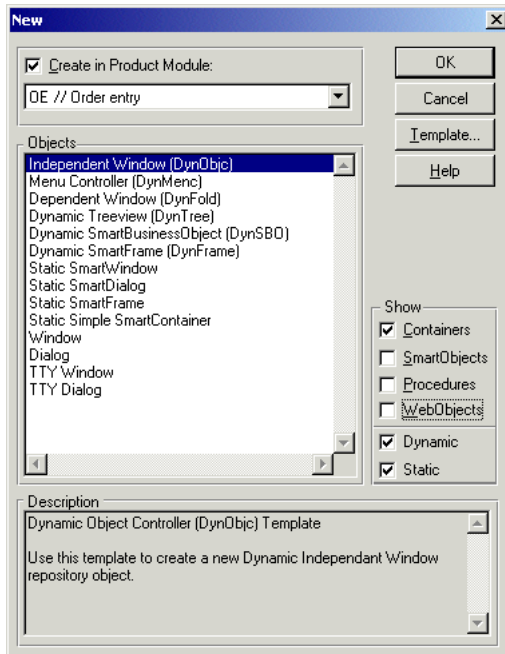


Figure 8–4: New object window

If the **Independent Window**, **Dynamic Menu Controller** or **Dependent Window** is selected, the Container Builder is launched in **add** mode with the standard object design widget running in the background, as shown in Figure 8–5.



Figure 8–5: Object design widget

If the Container builder is closed or hidden, double-clicking on any of the **open** area on the widget launches the Container Builder.

The toolbar is different in that **New**, **Open** and **Delete** functionality has been removed. This is because the Container Builder has been launched from the Appbuilder to create **one** new container, as shown in Figure 8–6. The object **Type** combo box has also been disabled because it was selected through the AppBuilder's **New** dialog box. The **Product Module** combo box also defaults to the one selected.

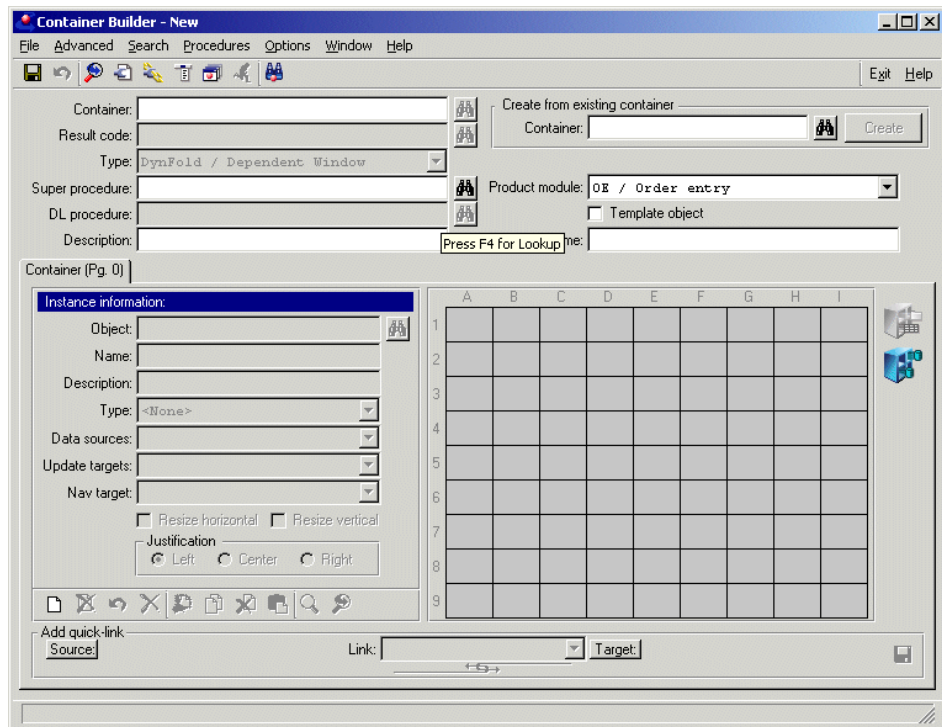


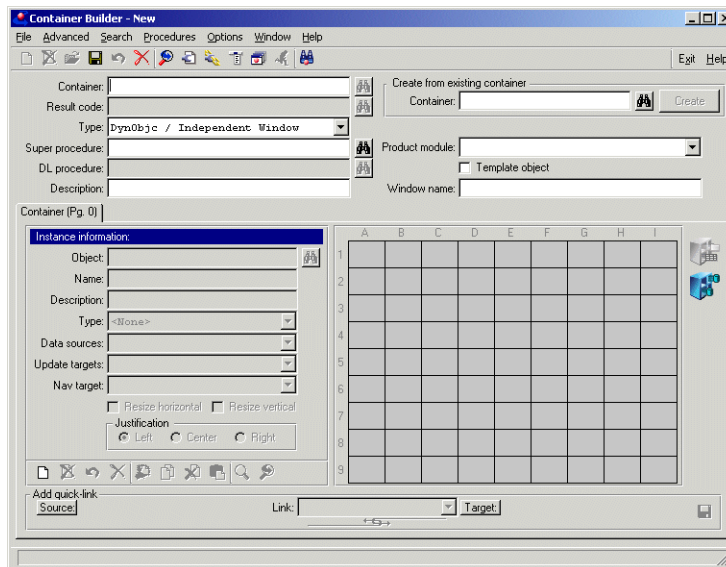
Figure 8–6: Container Builder launched from the AppBuilder

8.4 Using the Container Builder

The following sections discuss how to work with containers in the Container Builder.

8.4.1 Creating a new container

- 1 ♦ From the AppBuilder main window, select **Build→Container Builder**. The Container Builder window appears.
- 2 ♦ The window is launched in **Add** or **Find** mode; the **Add** button and **Container** list box are both enabled. Choose **Add** to create a new container:



- 3 ♦ Enter a **Container name** (the logical name of the container) just as you would assign a name to any dynamic container.
- 4 ♦ Select the **Type** of container (DynFold, DynObjc, or DynMenu).
- 5 ♦ Enter a description and assign the layout to an existing Product Module. Like every other object in the Progress Dynamics Repository, a container must be assigned to a Product Module. You might want to define a general or system Product Module for objects, such as container templates, that can be used in different application modules.

6 ♦ Optional fields:

- a) Select a **Super procedure**- name of a procedure that will be instantiated as a super-procedure of the SmartObject.
- b) If the container being created is to be based on an existing container (template), select the **Container** from the list box.
- c) If this container is to be flagged as a template, select the Template **Object** check box.

7 ♦ Click **Save**.

8.4.2 Assigning objects to the container

Again, the container can be for a window (Page 0), or for a page of a tab folder, or it can be appropriate for both. Typically, most of the objects you place in the layout will be template objects, to be replaced by specific objects of that type when you build windows and pages from the layouts. The standard exceptions to this are Toolbar objects and Folder objects. You will likely select a specific named Toolbar type (or possibly more than one) as a fixed component of many types of containers.

Object instance details

The section is made up of four distinct tool-sets. The first is the Object Instances details where typical IO functions are supported and the current object's are edited, as shown in [Figure 8–7](#).

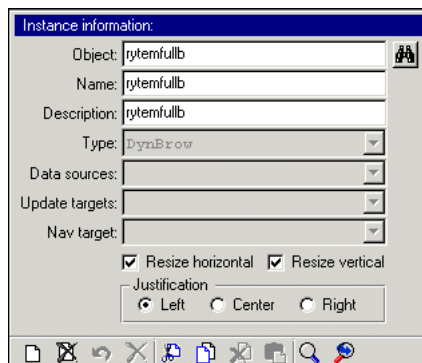


Figure 8–7: Object instances details

The IO buttons on the Object Instance editor are, in order:

- Add, Delete, Undo, Cancel Add (Reset)
- Cut, Copy, Cancel Cut/Copy, Paste
- Object Properties, Dynamic Properties

Layout grid

The layout grid is central to managing information of the object instances. It displays all the objects *per page* in their current positions, and the highlighted object's details are displayed alongside in the details section.

Figure 8–8 shows the layout grid.

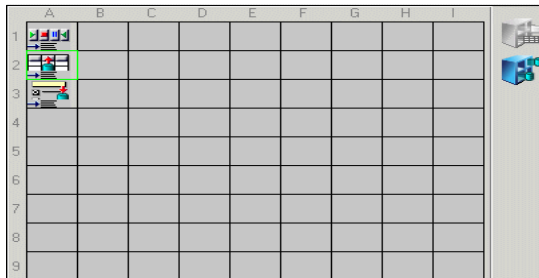


Figure 8–8: Layout grid

Pages of the layout grid

The first tab is the container itself, the 'Main Page' or 'Page0'. Selecting a tab displays the objects on that page, both visual and non visual, depending on the view (icon) selected. The exact layout of these objects is represented in the grid.

Moving an object to a different position/page

This is achieved by simply dragging and dropping objects on their desired position, provided it is available - indicated by a cell being white.

A few considerations to bear in mind when placing an object:

- If an object is center-justified, only that object is allowed on the specific row.
- Two adjacent objects cannot have the same justification, for example both right.
- If an object is dropped on another, the system determines if the two can be swapped, and if not, you are warned.

Advanced layout grid features

These are made available by right-clicking on the Grid to invoke a pop-up menu, as shown in [Figure 8–9](#).

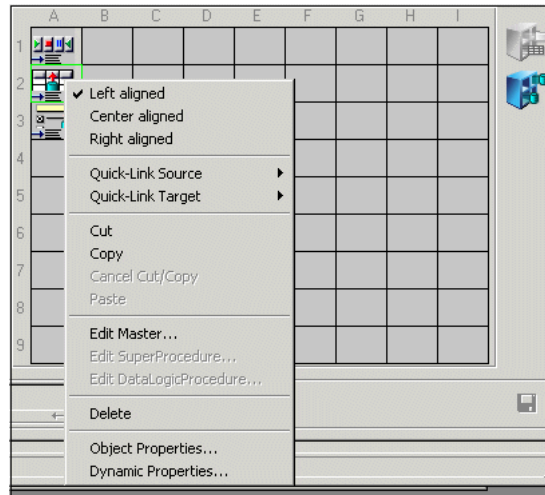


Figure 8–9: Advanced layout grid features

The Advanced Features are as follows:

- **Left aligned** — Aligns the object so that it is left-justified. Indicated on the pop-up menu by a check-mark. The Layout Grid evaluates valid alignment adjustments and disables invalid options (visual only).
- **Center aligned** — Aligns the object so that it would be centered. Indicated on the pop-up menu by a check-mark. The Layout Grid evaluates valid alignment adjustments and disables invalid options (visual only).
- **Right aligned** — Aligns the object so that it would be right-justified. Indicated on the pop-up menu by a check-mark. The Layout Grid evaluates valid alignment adjustments and disables invalid options (visual only).
- **Quick-Link Source** — Flags the object as a source for a SmartLink and is reflected in the Quick Link tool when the object is selected.
- **Quick-Link Target** — Flags the object as a target for a Smartlink and is reflected in the Quick Link tool when the object is selected.

- **Cut** — Flags the object as such until the object is Pasted elsewhere. This makes moving objects between pages easy. This is currently limited to one object, so if the cut action is used successively, only the last object has a **cut** status. The Grid displays available cells for the object to be pasted.
- **Copy** — Flags the object as such until the object is pasted elsewhere. This makes copying objects easy. This is currently limited to one object, so if the copy action is used successively, only the last object has a **copy** status. The Grid displays available cells for the object to be pasted.
- **Cancel Cut/Copy** — When you have either cut or copied an object, Grid positions become available (white). Select this option to cancel the **Cut** or **Copy** action, and the cells return to their normal status.
- **Paste** — Remains disabled unless the **Cut** or **Copy** action has been initiated on an object.
- **Edit Master** — Opens the Master version of the object in the AppBuilder for editing.
- **Edit SuperProcedure** — Opens the Procedure Editor with the super procedure of the selected object for editing.
- **Edit DL Procedure** — Opens the data logic procedure of the selected object in the Procedure Editor window for editing.
- **Delete** — This action deletes the selected objects, along with any links associated with it.
- **Object Properties** — Launches the AppBuilder Property Sheet for the selected object.
- **Dynamic Properties** — Launches the Dynamic Properties Sheet for the selected object.

Object types

The two icons shown in [Figure 8–10](#) indicate whether the objects for a page being viewed are visual or non-visual in nature courtesy of the presence of a red check mark on either. Typically, non visual objects are data source objects.



Figure 8–10: Object type selectors

Add Quick-Link tool

The **Add Quick Link** tool, as shown in [Figure 8–11](#), enables developers to quickly link objects together, even objects that exist on different pages. This works effectively thanks to the visual nature of the layout in conjunction with the drag-and-drop functionality.

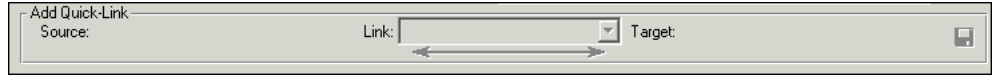


Figure 8–11: Add Quick-Link tool

Using the Add Quick-Link tool

To add a quick link, follow these steps:

- 1 ♦ Select the object to be used as the source. Drag and drop it out of the grid onto the source area. Alternatively, use the pop-up menu and select the appropriate option on the layout grid.
- 2 ♦ Select the object to be used as the target. Drag and drop it out of the grid onto the target area. Alternatively, use the pop-up menu and select the appropriate option on the layout grid.
- 3 ♦ Select a valid link type.
- 4 ♦ Click **Save**.

Considerations when using the Add Quick-Link tool:

- If a link already exists, the source and target areas will be grey and the Save button disabled until the link type is changed, as shown in [Figure 8–12](#).

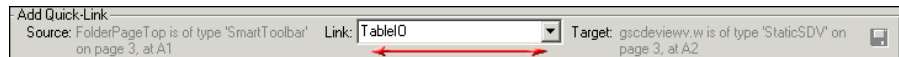


Figure 8–12: Add Quick-Link tool disabled

- The tool provides the functionality to swap source and target objects by clicking on the I-directional arrow at the bottom of the Add Quick-Link tool, as shown in [Figure 8–13](#).

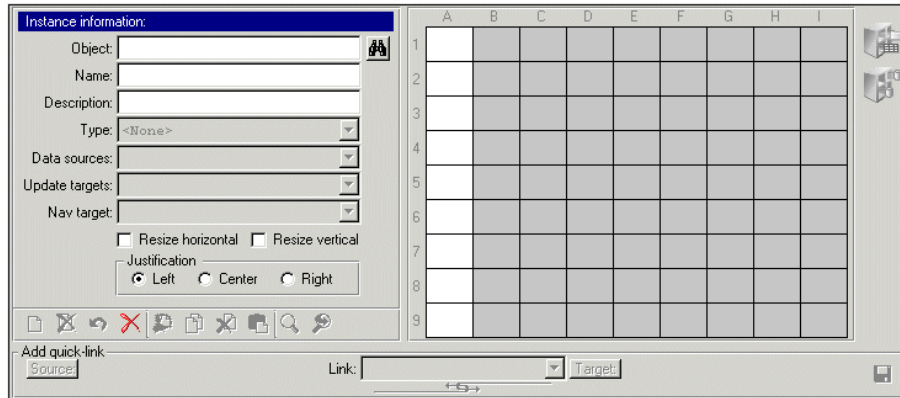


Figure 8–13: Add Quick-Link tool enabled

Adding an object

To select an object to add to the container, follow these steps:

- 1 ♦ On the **Object Instance** region of the Container Builder, select the **Add** button. The layout grid shows all the available **cells** where the object can be placed, as shown below:



- 2 ♦ Complete the necessary fields:
 - a) **Object lookup** — Select the object to be placed.
 - b) **Name** — The instance name of the object, defaults to the actual object name.
 - c) **Description** — Defaults to the short description of the object.
- 3 ♦ Complete the optional fields (only available for visual objects):
 - a) **Resize horizontal/vertical** — Gives you the option of allowing an object to dynamically resize on either plane at runtime.
 - b) **Justification** — Allows you to define the alignment of every object.
- 4 ♦ Select the relative position of the object by clicking in any one of the available cells, indicated by a white background in contrast to the chosen background color.
- 5 ♦ Click **Save**.

8.5 Advanced container features

These features allow for additional modification or customization of containers and their properties. They are enabled or disabled according to the selected object.

8.5.1 Container properties

This option opens the Dynamic Properties window, shown in [Figure 8–14](#), as described in [Chapter 6, “Using the AppBuilder in Progress Dynamics.”](#)

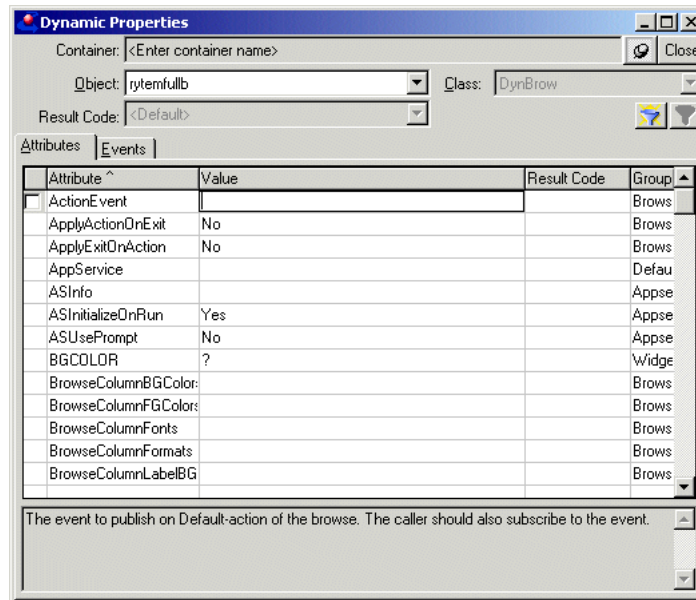


Figure 8–14: Dynamic Properties window

Another convenience is the ability to open a container’s super procedure in the Section Editor by right-clicking in the Container Builder and selecting **Edit Custom SmartObject**.

8.5.2 Page maintenance

The Page Maintenance tool is provided to manage page-related data. The tool consists of a browser to select the desired page, a maintenance toolbar, and a viewer for actual data maintenance, as shown in [Figure 8–15](#). You can perform standard IO functions.

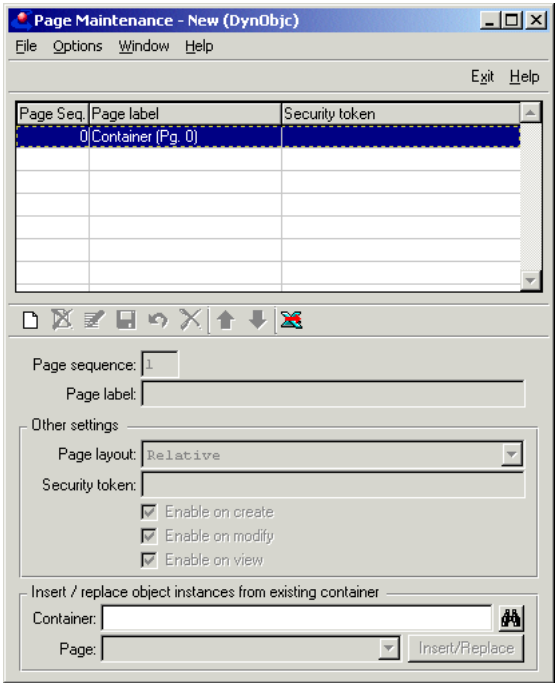


Figure 8–15: Page Maintenance window

Page sequencing

Two up/down arrow icons are included on the toolbar to allow movement of the selected page to change the sequence in which tab folders appear on a container. Page0 cannot, however, be moved because it is the **background** page that is always there. If a folder exists on Page0, it cannot be re-sequenced.

Inserting instances for pages on existing containers

This tool allows you to insert object instances from any page of another container into a page.

After selecting a container from the list, the container builder determines whether the selected container has multiple pages or not, and if so, the combo below the list is enabled for you to select the specific page that you want to use as a template. If not, the selection defaults to Page0. The **Insert/Replace** is then enabled. If object instances exist on the page, you are prompted as shown in [Figure 8–16](#).

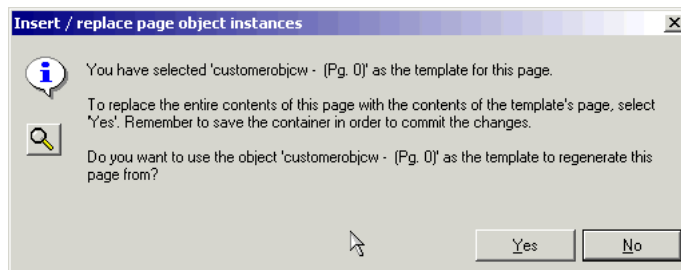


Figure 8–16: Insert / replace warning

After reading the message, select **Yes** or **No** to continue.

8.5.3 Links maintenance

SmartObjects communicate by means of named connections that in the ADM are called SmartLinks. A SmartLink is nothing more than a collective name for a set of named events. They can be published (using the Progress™ 4GL PUBLISH statement) by the initiator, or Source, of the event. They are subscribed to (using the Progress 4GL SUBSCRIBE statement) by one or more Targets for the event. As the designer of a window (or in this case, a container for many windows of a type), you need only name the Source and Target objects and the name of the link between them. The standard ADM code handles all the setup needed for the objects to communicate. If you are not familiar with the Version 9 ADM, you can read more about this mechanism in the Version 9 documentation. Some new links defined for the Progress Dynamics framework are described in the [“New Progress Dynamics SmartLinks”](#) section.

After you select all the objects that are in the window or page, you need to define all the links between objects in the Container. The links will be made between the objects in the container; when a specific window or page is built from the container, these same links will be created between the actual application objects the application assembler chooses.

The standard Progress Dynamics Repository includes a number of template objects, and templates for window and page layouts with the links already defined, to satisfy many common kinds of application needs. You only need to go through the process of creating a new layout if none exists already for the combination of objects you need. Once you have defined it, any developer who wants to assemble an application can build windows from it without understanding or modifying the specifics of the object positions, links, and so on.

Links Maintenance tool

The Links Maintenance tool is used to manage ALL links on the container between all instance objects and the container itself. Filtering and sorting of links can be done quite intuitively using the tools provided. By clicking on any one of the column headers, a browser's details will be sorted by that particular field. A special filtering tool has been provided to allow only a selected list of links to be displayed.

Filtering links is made possible by a combination of combos. This works exactly like the standard Appbuilder Links window works. Besides being able to filter on source, target, and link type, you can also see links pertaining to a specific object—whether it is the target or source—using the **To / From** field. Figure 8–17 shows the Links Maintenance Window. Another useful option is to filter links by folder page.

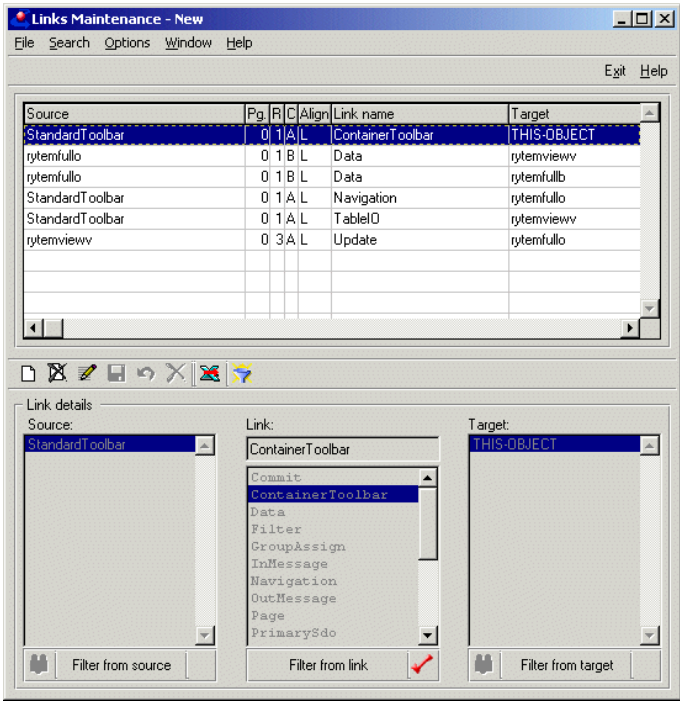


Figure 8–17: Links Maintenance window

Adding/assigning links

To add or assign a link, follow these steps:

- 1 ♦ Click the **Add** button.
- 2 ♦ Select the source, link, or target for the link. If, for example, the required object is not in the source list, click on the **Filter from source**, **Filter from link**, or **Filter from target** button to reveal all possible sources for links. This works for all three columns. Whichever item is selected from any column, the other two are dynamically updated to only show allowable items. This is based on the data in the `ryc_supported_link` table. Again, the red check mark indicates which component is currently **active**.
- 3 ♦ Select the other two components of the link.
- 4 ♦ Click **Save**. The browser is updated and the latest record is added.

Maintaining links

Links can be edited and deleted using the maintenance tool. By simply selecting the link from the browser, the columnar link details section is dynamically updated with the specific link's details. Select the **Modify** button to enable the columns. Here, the source, target or link type can be changed. Remember to click on **Save** for the change to take effect.

To delete a link, again, simply select it from the browser and click the **Delete** button.

New Progress Dynamics SmartLinks

Most of the links used by objects in Progress Dynamics are standard Version 9 ADM SmartLinks. A few new ones have been added beyond those described in the Version 9 documentation. This section describes the following new SmartLinks:

- **ContainerToolbar link** — Progress Dynamics supports many functions initiated from its toolbars in addition to those that are standard to Version 9 SmartObjects. To initiate those events, the Toolbar event published along the ContainerToolbar link passes the name of the function to its container or to a contained object. So for a toolbar that supports any of these additional functions (including invoking Notepad, Word, Excel, a Calculator, an Internet connection, e-mail, a Print Setup routine, the Suspend function, the Re-Logon function, inline Translation, Help, etc.) there is routinely a ContainerToolbar link from a toolbar to its container. Whenever the user invokes any such function in the toolbar by choosing the appropriate button, the Toolbar event is published by the toolbar, and it passes the name of the specific function as an input parameter. Code in the standard super procedure for Containers, `src\adm2\container.p`, receives the Toolbar event and runs the correct code to handle the function.

With earlier versions of Dynamics, both browsers and containers could have used the toolbar link and could have been toolbar-targets. The ContainerToolbar link is now used to prevent ambiguity and make defining enable rules for toolbar actions easier. Toolbar links to containers now use the ContainerToolbarLink. The ContainerToolbar link has four new attributes:

- ContainerToolbarTargetEvents (toolbar class)
- ContainerToolbarTarget (toolbar class)
- ContainerToolbarSourceEvents (container class)
- ContainerToolbarSource (container class)

You can also create Toolbar links from datavisual objects (browsers and viewers) in the same window. The standard support procedure for datavisual objects, the super procedure `src\adm2\datavis.p`, likewise has a toolbar procedure to receive the Toolbar event, along with a parameter naming a specific function to be processed by the Browser. The functions are supported by several different toolbar bands described in the “[Standard toolbar objects](#)” section. They include access to Comments, Audit, Data Export, Print preview, Data Filter, and Find. If the toolbar you use supports any of these operations on the current record in a Browser or Viewer, then there should be a Toolbar link from the Toolbar to the Browser or Viewer in your layout.

- **Toolbar link** — The toolbar link is in the `datavis` class to support browse toolbar actions such as **Filter**→**Find**→**Export to Excel** for viewers.

Action rules for toolbars follow this logic:

- Browse toolbar `tableio` actions have the same record state enable rule as standard toolbar `tableio` actions plus an additional rule to check whether `FolderWindowToLaunch` is not blank.
- Comments/Auditing/Status/Print/Export have record available enable rules.
- `AuditEnabled` attribute added to SDO and used in audit action enable rule.

A toolbar linked to a data visual object refreshes often to apply these new rules.

- **ToggleData link** — Use the ToggleData link to enable records in a parent Browser in one window to be browsed independently of the display of records in a dependent child Browser. It effectively disables the Data link from parent to child so that data for the dependent query is not constantly fetched from the server as the user browses through different records in the parent query. When a parent record is explicitly selected (by double-clicking on it, for example) a new child window with the Browser for that record's child records is displayed. It keeps these two windows independent once the user initiates the child window. If the user explicitly selects a new parent record, a new child window is displayed for it. It is also made between the Browser in the child window and its container window. However, the ToggleDataTargets property should generally be used in preference to this link, as described in the [Progress Dynamics Programming Handbook](#).
- **Primary SDO link** — Where there is more than one SmartDataObject™ in a window or page, the PrimarySDO link tells its container which SDO is to be used as the Target for a Data pass-through link from another window or page. The pass-through link establishes a connection between an SDO and its query in one window and a dependent SDO and its query in a second window. Although the PrimarySDO link is strictly necessary only when a window or page actually contains two potential Data Targets for a Data Source in another container, it is customary to define the PrimarySDO link so as to eliminate any chance of confusion as to where the path of the Data links should go. The PrimarySDO link goes from a contained SDO to its container.
- **TreeFilter link** — The TreeFilter link was created to allow the Dynamic TreeView object to have filtering capabilities. This link is added automatically when creating a new Dynamic TreeView with a filter viewer associated with it.

Based on the filter selections, the browser displays the objects that satisfy the criteria.

The Foreign Fields Mapping dialog

In the instances section of the Container Builder, there is a button (...) to the right of the Foreign Fields field. Click this to display the Foreign Fields Mapping dialog, as shown in [Figure 8–19](#).

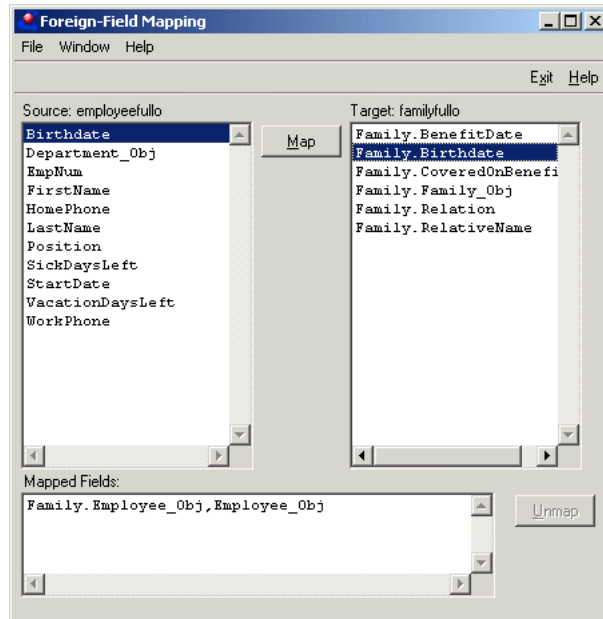


Figure 8–19: Foreign Fields Mapping dialog

Use this dialog to explicitly map SDO and SBO source and targets. Select one field in each list and choose Map. The list at the bottom shows the fields that are currently mapped. Choose one and click unmap to undo the mapping.

8.5.4 Object menu structure

This tool allows the addition and maintenance of menu structures associated with the container, as shown in [Figure 8–20](#). This is typically used for Dynamic Menu Controllers and allows menu structures (a grouping of one or more menu items) to be merged into the containers menu structure.

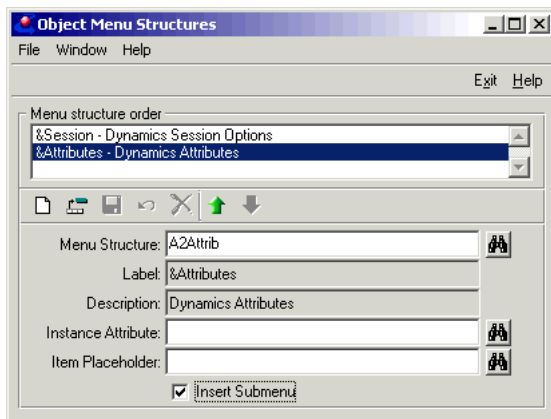


Figure 8–20: Object Menu Structures window

8.5.5 Object Initialization sequencer

This tool allows you to specify the order in which objects are initialized **per page**. It simply filters the objects per page and allows you to move objects using up/down arrows, as shown in [Figure 8–21](#).

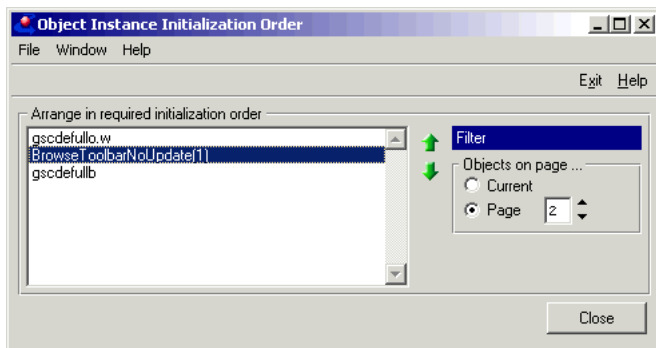


Figure 8–21: Object Initialization sequencer

NOTE: Due to Repository constraints, Page0 objects cannot be re-ordered.

8.5.6 Run

This option launches the current container for previewing purposes. It is only enabled if any changes that have been made have been saved.

8.6 Standard toolbar objects

The Progress Dynamics Repository comes with a number of standard Toolbar objects predefined. These should be satisfactory for many of the kinds of windows you need to build. This section discusses some of the standard toolbars.

8.6.1 FolderPageTop toolbar

In addition to a tab folder page, you can also use the FolderPageTop toolbar in a single-page window. (You can place any of the Toolbars at the top of a window or page, at the bottom, or in between other objects in the Main section of the window or page). The FolderPageTop Toolbar includes a full update band of buttons (including delete, cancel, and reset) and a band of **Navigation** buttons (**First**, **Prev**, **Next**, and **Last**). [Figure 8–22](#) shows this toolbar.



Figure 8–22: FolderPageTop toolbar

[Table 8–1](#) describes the links you need to create for this Toolbar.

Table 8–1: FolderPageTop toolbar links

For this functionality . . .	You must include . . .
To enable the Navigation buttons	An SDO in the layout and a Navigation link from Toolbar to SDO
To enable the Update buttons	A TableIO link from the Toolbar to a Viewer or other updateable object in the window
To pass records to the Viewer for display	A Data link from the SDO (which of course you do not see at run time) to the Viewer
To handle updates of records	An Update link from the Viewer back to the SDO

As with all Toolbars, the functions of the buttons are duplicated in the menu, as shown in [Figure 8-23](#).

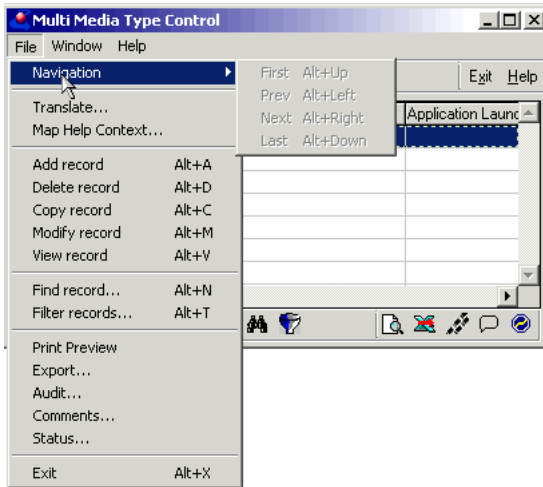


Figure 8-23: File Menu that duplicates toolbar functions

8.6.2 ObjcTop toolbar

[Figure 8-24](#) shows the ObjcTop Toolbar.

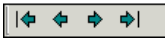


Figure 8-24: ObjcTop toolbar

The ObjcTop Toolbar contains:

- **Navigation** and **Exit/Help** buttons.
- A Window menu item to let the user select whether each selection of a record should bring up a separate maintenance window.
- A Help menu.
- If the window contains an updatable object and a TableIO link, there are also **OK** and **Cancel** buttons.

8.6.3 FolderTop toolbar

The FolderTop Toolbar includes an **Update** band without **Delete**, a **Navigation** band, a **SpellCheck** button, and the set of **OK/Cancel/Exit/Help** buttons, as shown in [Figure 8–25](#).



Figure 8–25: FolderTop toolbar

8.6.4 Browser toolbar

The browser toolbar looks like the one shown in [Figure 8–26](#).



Figure 8–26: Browser toolbar

It has a special band of **Update** buttons. These are designed not to edit a record in place, but to launch a separate container window for record maintenance. To enable this, you must define the Launch Container property for the Browser in the window. Likewise, double-clicking on one of the records in the Browser brings up the separate container window to edit the selected record. The Toolbar also has a **Find** and **Filter** band. When the user chooses the **Find** or **Filter** button, a separate dynamic window appears, as shown in [Figure 8–27](#). Use this window to create a subset of the records in the query or locate a particular record.

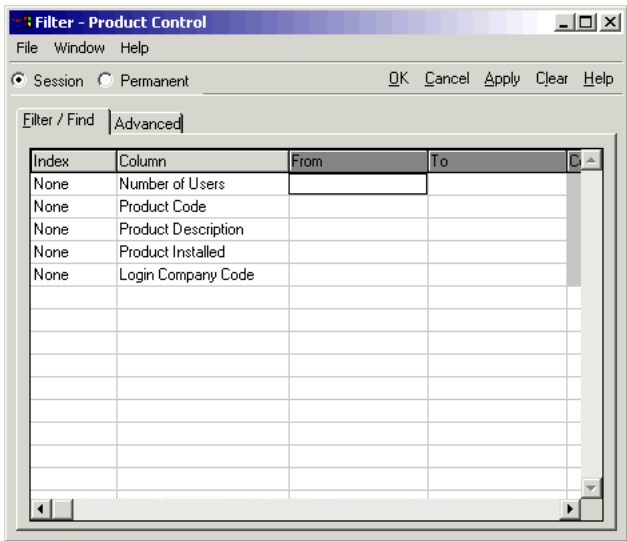
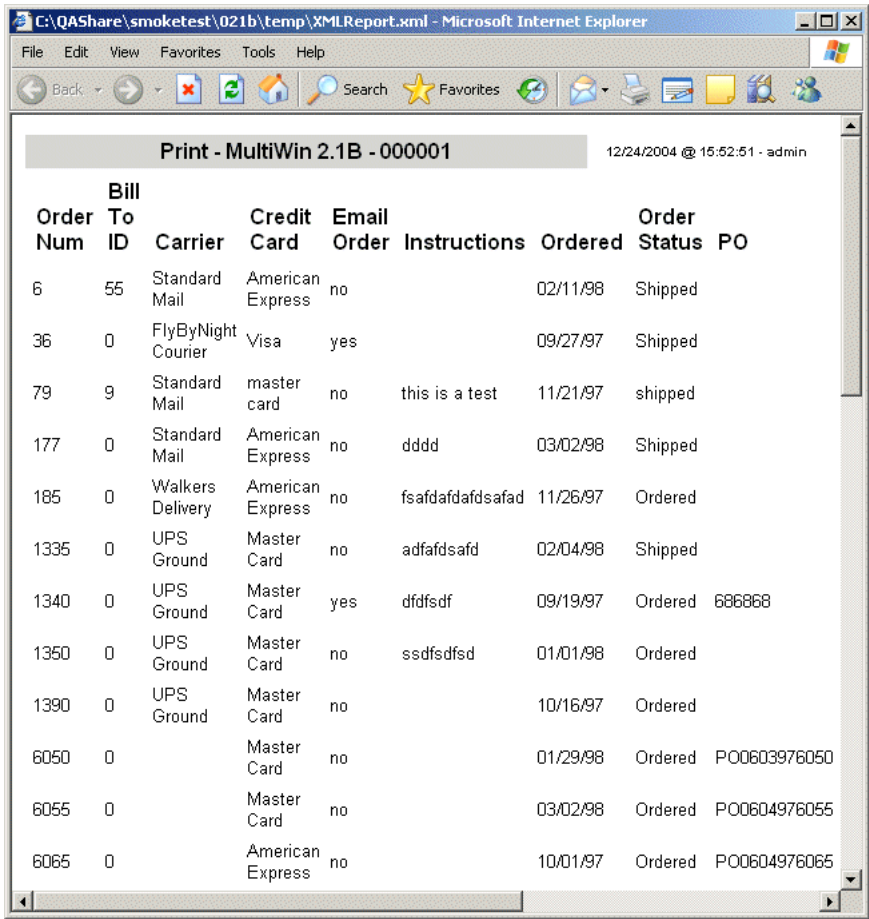


Figure 8–27: Filter window

There is also a band of buttons that:

- Allow records in the Browser to be exported to a Print Preview window or to Excel.
- Associate an Audit record or a Comment with the currently selected record.
- Provide status information for the record.

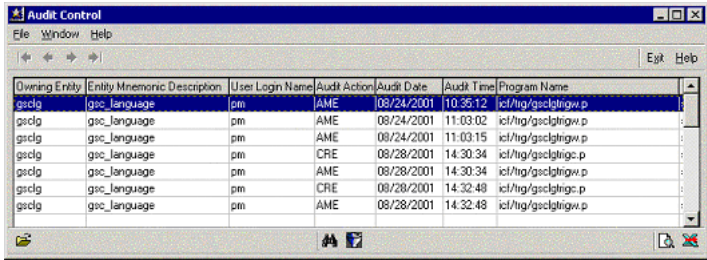
For example, when you choose **Print Preview**, the contents of the Browser are sent in XML format to the default Windows Internet browser, as shown in [Figure 8–28](#).



Order Num	Bill To ID	Carrier	Credit Card	Email Order	Instructions	Ordered	Order Status	PO
6	55	Standard Mail	American Express	no		02/11/98	Shipped	
36	0	FlyByNight Courier	Visa	yes		09/27/97	Shipped	
79	9	Standard Mail	master card	no	this is a test	11/21/97	shipped	
177	0	Standard Mail	American Express	no	dddd	03/02/98	Shipped	
185	0	Walkers Delivery	American Express	no	fsafdafdafdsafad	11/26/97	Ordered	
1335	0	UPS Ground	Master Card	no	adfafdsafd	02/04/98	Shipped	
1340	0	UPS Ground	Master Card	yes	dfdf sdf	09/19/97	Ordered	686868
1350	0	UPS Ground	Master Card	no	ssdf sdfsd	01/01/98	Ordered	
1390	0	UPS Ground	Master Card	no		10/16/97	Ordered	
6050	0		Master Card	no		01/29/98	Ordered	PO0603976050
6055	0		Master Card	no		03/02/98	Ordered	PO0604976055
6065	0		American Express	no		10/01/97	Ordered	PO0604976065

Figure 8–28: Print Preview Example

When you choose **Audit**, an audit history for the record appears, as shown in [Figure 8–29](#).



The screenshot shows a window titled "Audit Control" with a menu bar (File, Window, Help) and a toolbar (Exit, Help). Below the toolbar is a table with the following columns: Owning Entity, Entity Mnemonic, Description, User Login Name, Audit Action, Audit Date, Audit Time, and Program Name. The table contains several rows of audit data.

Owning Entity	Entity Mnemonic	Description	User Login Name	Audit Action	Audit Date	Audit Time	Program Name
gsclg	gsc_language		pm	AME	08/24/2001	10:35:12	icl/tig/gscldtgrw.p
gsclg	gsc_language		pm	AME	08/24/2001	11:03:02	icl/tig/gscldtgrw.p
gsclg	gsc_language		pm	AME	08/24/2001	11:03:15	icl/tig/gscldtgrw.p
gsclg	gsc_language		pm	CRE	08/28/2001	14:30:34	icl/tig/gscldtgrw.p
gsclg	gsc_language		pm	AME	08/28/2001	14:30:34	icl/tig/gscldtgrw.p
gsclg	gsc_language		pm	CRE	08/28/2001	14:32:48	icl/tig/gscldtgrw.p
gsclg	gsc_language		pm	AME	08/28/2001	14:32:48	icl/tig/gscldtgrw.p

Figure 8–29: Audit Control

When you choose **Comments**, the Comments Control window appears. Here you can view, add, or edit comments of any kind for this record.

8.6.5 BrowseToolBarNoUpdate toolbar

This toolbar has the same bands as the BrowseToolBar, except for the **Update** band.

8.6.6 FolderTopNoSDO toolbar

This toolbar just has the **OK/Cancel/Exit/Help** band of buttons.

8.6.7 DynToolBar

This toolbar has a full band of **Update** buttons, **Navigation** buttons, and a **Filter** button.

8.6.8 Other toolbars

Other toolbars provided for more specific uses include:

- MenuController
- LookupToolBar

8.7 Preferences

Figure 8–30 and Figure 8–31 show the Container Builder Preferences dialog tabs. The dialog is available on the Options menu and allows you to specify UI-behavior and -properties for the Container Builder.

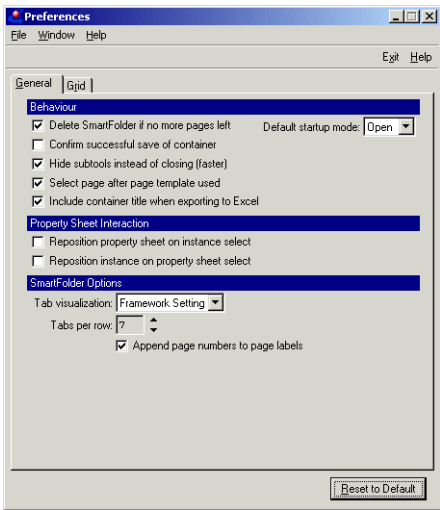


Figure 8–30: Container Builder Preferences - General tab

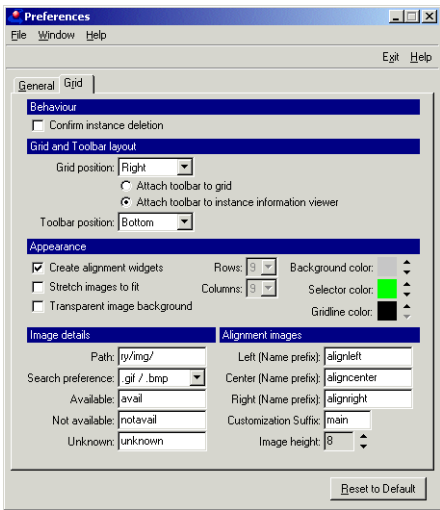


Figure 8–31: Container Builder Preferences - Grid tab

8.8 Customization option for dynamic viewers and browsers

When you open a dynamic viewer or browser in the AppBuilder, the **Layout** menu contains this option: **Custom Layout**. This feature lets you define and edit custom layouts based on result codes. For each existing result code, you can define one custom layout.

When you create a dynamic viewer or browser, the layout you define is known as the *default layout*. Every instance of the object uses this layout. For example, suppose your application has a result code named LargeFontUI that applies when your application identifies a vision-impaired user. You can define a custom layout named LargeFontUI. When the application applies the LargeFontUI result code, every instance of the object uses the new custom layout instead of the default layout.

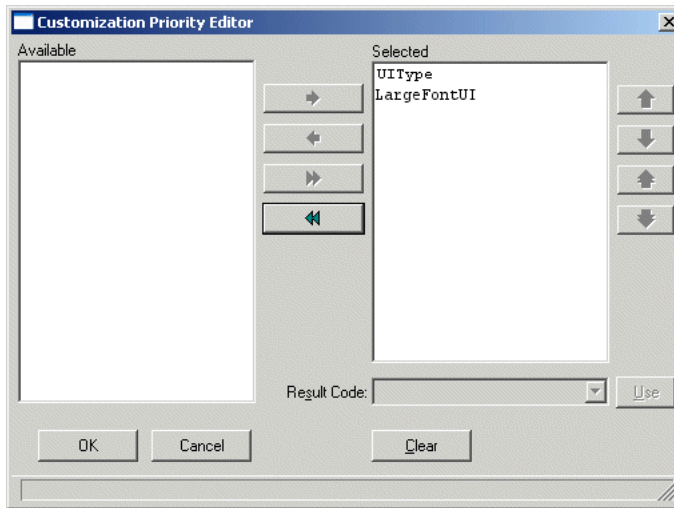
Some essential facts about this feature:

- The **Custom Layout** menu option is only available when a dynamic browser or viewer is in focus in the AppBuilder.
- From the **Layout** menu, select the **Custom Layout** option to bring up the **Custom Layout** dialog box. (Consult the online help for assistance with the dialog box fields.)
- The **Custom Layout** option works exclusively with the object in focus.
- Only a single layout (default or custom) for a particular object is available in the AppBuilder at any one time.
- You can have custom layouts of different objects open at the same time.
- When multiple custom layouts are open, switching the result code on one object (through the **Custom Layout** dialog box) has no effect on the custom layouts of the other objects.
- The title bar of open objects shows the result code associated with that custom layout.
- If an open viewer has no layout previously defined for a selected result code, then AppBuilder creates a new custom layout initialized to be identical to the layout that was in focus immediately before the result code was selected.
- Choose the **Save** icon in the AppBuilder's main toolbar to save changes to all layouts of the focused object.
- The Re-Logon feature does not reset active custom layouts. Exit your session and start a new one to reset to default layouts.

8.8.1 Customization priority

You can have a layout scheme that includes multiple layouts associated with result codes of varying customization types. When you run the browser or viewer, the Framework needs to know which of these result code based customizations takes precedence over the other. It will try to apply all customizations, but in the case of conflict, it needs to know which customization receives priority.

Make sure the object you are working with is currently selected in AppBuilder, and select **File**→**Customization Priority** to display the **Customization Priority Editor**:



Here you can create a list of the appropriate result codes by moving them to the right-hand column. Prioritize the result codes by ordering the list. Put the highest priority code at the top of the list, and so on.

Click OK when you are finished. Notice that the AppBuilder main window now has a Customization button on the right-hand side, as shown below:



This button reminds you that you have customizations in effect. Click the button to return to the **Customization Priority Editor**.

8.8.2 Synchronization

Each custom layout is saved as a variation of the default layout. This means that only those attributes that are different from the default layout are stored with the custom layout. Unchanged attributes in the custom layout are in synchronization with the default layout. When you make a change in the default layout, that change will be reflected in an existing custom layout if, and only if, the affected attributes are *in sync* (the same as) with their default layout counterparts.

8.8.3 Adding and removing widgets

All widgets that you want to appear in any custom layout of the object must be part of the default layout, although they can be hidden. When working with custom layouts, you might delete and add widgets in AppBuilder. However, in reality, the widgets in the custom layouts are not added or deleted, they are simply made visible or hidden by your paste or delete actions.

8.8.4 Supported attributes

The attributes supported for custom layouts in the static property dialogs are in the following table:

BGCOLOR	GRAPHIC-EDGE	REMOVE-FROM-LAYOUT
COL	HEIGHT	ROW
CONVERT-3D-COLORS	LABEL	SEPARATORS
EDGE-PIXELS	NO-BOX	THREE-D
FGCOLOR	NO-FOCUS	WIDTH
FILLED	NO-LABELS	–
FONT	NO-UNDERLINE	–

All additional attributes are supported in the dynamic property sheet.

NOTE: The REMOVE-FROM-LAYOUT attribute is a pseudo-attribute that hides an object in a custom layout.

8.9 Dynamic SmartFrames

This release of Progress Dynamics introduces dynamic SmartFrames™. Dynamic SmartFrames belong to the DynFrame class and are container objects that do not have an associated window. You build dynamic SmartFrames in the Container Builder and add the dynamic SmartFrame to a window to render it. A window object can contain from zero to many dynamic SmartFrames.

The dynamic SmartFrame:

- Can contain other DynFrame objects.
- Can contain folder objects.
- Can be included on a folder page.
- Is intended to contain objects like SDOs, browses, and viewers.
- Is *not* intended to contain field-level widgets, such as DataFields and fill-in fields.
- Acts as data sources and data targets.
- Is supported by the Links Maintenance tool.
- Is a self-contained object that is rendered using the physical object `ry/uib/rydynframw.w`.
- Has `ry/app/rydynframp.p` as a super procedure, and inherits from the ADM2 Container class.

Building Progress Dynamics TreeView Windows

This chapter describes how to create Progress Dynamics windows with a TreeView layout. It details how to use property sheets to define nodes of the tree and the overall tree structure. The nodes are represented in an ActiveX-based TreeView object on the left side of the window. The dynamic windows launched from the selection of the nodes appear as frames on the right-hand side. You can use a TreeView to represent either hierarchical data or a menu structure.

This chapter discusses the following topics:

- [Introduction](#)
- [Building a TreeView window](#)
- [Setting up structured nodes](#)
- [Using an extract program for a node](#)
- [Creating a filter viewer for a TreeView window](#)
- [Running the TreeView window](#)
- [Building a menu structure TreeView window](#)
- [Summary](#)

9.1 Introduction

The standard Progress Dynamics dynamic window gives you many options for the appearance of your application windows. As discussed in previous chapters, you can combine Viewers, Browsers, Toolbars, Folders, and other objects in a variety of layouts to provide many different looks to different parts of your application.

The Progress Dynamics TreeView gives you another option. You can build a dynamic window with a TreeView control to the left, which can represent a hierarchy of related data, a nested menu structure, or some other multi-level data with which you populate the TreeView. To the right, you can relate application pages and sub-windows to different levels in the TreeView, to show details or allow updates of any record at any level.

The TreeView window described in this chapter is referred to as a *dynamic* TreeView, because it is a largely data-driven object that is controlled by a single physical procedure file (rydyntreew.w). This file is a variation of the rydyncontw.w procedure file. The rydyncontw.w file generates all the varieties of dynamic windows discussed in earlier chapters that describe the Container Builder and the kinds of objects with which you can populate it. Two property sheets allow you to define what to display at each node level in the tree, as well as the overall structure of the TreeView itself.

The dynamic TreeView window uses the underlying Progress Dynamics object called the TreeView Object. The TreeView object is a 4GL procedure that wraps a TreeView ActiveX Control. It provides a standard interface from other objects to the methods in the TreeView control that populate and manipulate the nodes of the TreeView.

NOTE: Progress Dynamics uses the standard Microsoft TreeView control.

You do not need to understand this object in detail to build dynamic TreeView windows. Generally speaking, you need to learn more about the TreeView object if you want to use the TreeView to build a kind of window that the dynamic TreeView window does not provide for you. This might include:

- Achieving a fundamentally different look and arrangement of objects on the screen
- Using the TreeView to coordinate data from objects the dynamic TreeView window does not handle yet, such as Progress B2B objects for communication to other applications using JMS
- Assembling a larger compound object from the TreeView object, for example to manage SDOs or SBOs, etc. in a different way than how the dynamic TreeView works

To build a TreeView for this kind of purposes however, you must do considerably more programming of calls to the TreeView APIs.

9.1.1 Components of the dynamic TreeView window

You can build a TreeView window from many different objects, but the standard window will always have roughly the appearance of the window shown in [Figure 9–1](#).

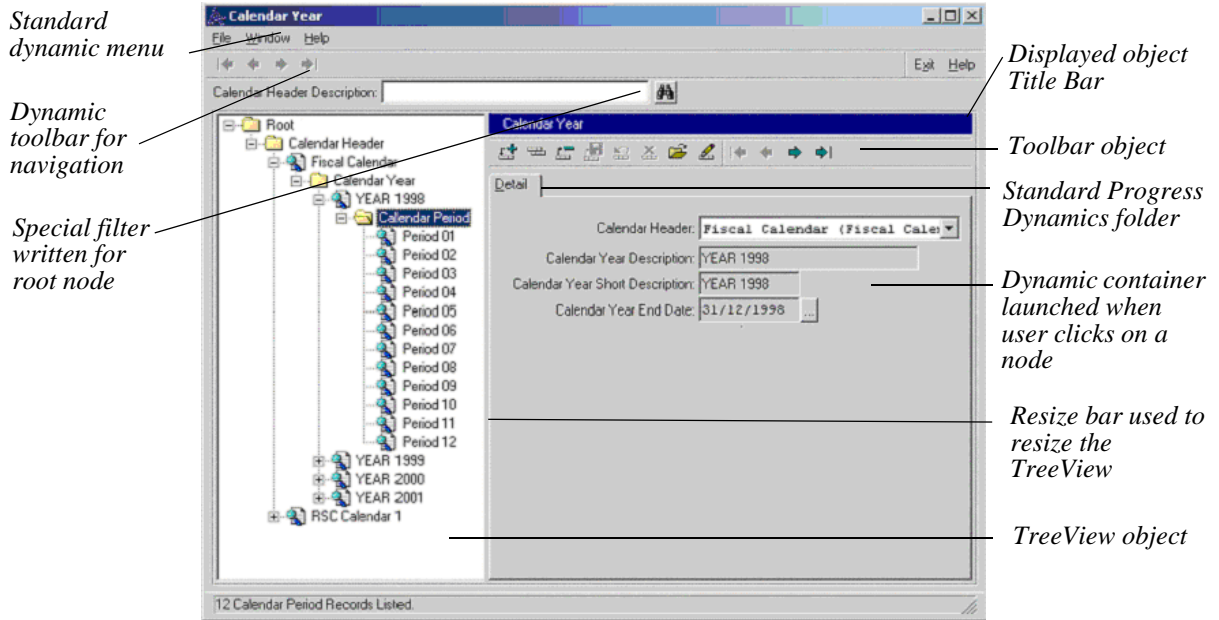


Figure 9–1: Folder toolbar linked to current displayed logical object

The Dynamic sub-window manages data in the panel on the right side of the TreeView window. These can be any windows you have built using the framework that can be launched by the TreeView. Though originally constructed as independent windows of one or more pages, because of the dynamic nature of the framework, these objects can be made frames of the TreeView window, providing a single-window interface with a number of different containers managed by a single main container window. You can associate a separate detail or maintenance window with each node level in the tree. The TreeView window automatically displays the appropriate frame and data for the current node. You can include in each of these sub-windows Viewers, Browsers, and other objects appropriate to the current level of the tree. You can make the sub-window itself a multi-page Tab Folder, if that is appropriate.

The TreeView window has an extensive API that you can program to customize its behavior in a number of ways. However, standard TreeView operations are provided for you just by filling in the TreeView and Node property sheets, so the API is not discussed here.

9.2 Building a TreeView window

Building a dynamic TreeView is basically a two-step process. First you must define the nodes of the TreeView, the different levels within the tree. Each of these nodes determines the data displayed at that level and its relationship to the level above. Then you define a TreeView window itself, which can incorporate any number of nodes to form a multi-level tree.

Before you define tree nodes, you must make sure that the objects referenced in the node definitions are already defined and registered in the Repository. This includes SDOs or SBOs that will provide data for the nodes, and container windows that will be displayed in the right-side pane when data nodes in the TreeView are selected.

As an example, you are going to build a TreeView window with three levels of data from the Sports2000 database: Departments, Employees in each Department, and Family members for each Employee. Before you start, follow these steps:

- 1 ♦ Run Entity Import to import table and field definitions for the tables involved.
- 2 ♦ Place these definitions into a new Employee (EMP) Product Module, so that they are grouped together and easy to locate.

NOTE: If you have worked through the *Getting Started with Progress Dynamics*, book you have already done this.

- 3 ♦ Run the Object Generator to create SDOs, dynamic Browsers and Viewers for these tables.

For this example, you will not be using Browsers. You will use Viewers to provide more exact control over the Viewer layout, but the Browsers and Viewers generated for you will still be useful in other windows.

- 4 ♦ Build a static SmartDataViewer in the AppBuilder for each of the Department, Employee, and Family SDOs.
- 5 ♦ In the Container Builder, create windows where the Department, Employee, and Family records can be maintained. For each of the three tables, follow these steps:
 - a) In the AppBuilder, select **New→Dependent Window (DynFold)**. The Container Builder displays.
 - b) Enter a name for the container. (For this example, use the table name or dump name if that is shorter, plus foldwin).
 - c) Select **rywinFolder** as the container template (Create from existing container).
 - d) Enter a **Description** and add the WindowName in the Property Sheet.

- e) Select **Page 1**.
- f) Select the **Pages** tool and set the Page Label to an appropriate name, then choose **Save and exit from the Pages** tool.
- g) On the Object Instances section, choose the **Lookup** on the Object field to select a Viewer to replace the template viewer.
- h) In the Lookup window, select the Viewer you created for the table.
- i) Choose the **Save** button and exit the Container Builder window.

When you have finished all the preliminary steps, you can then proceed to define the tree nodes and the TreeView window itself, as described in the “[Defining TreeView nodes](#)” section.

9.2.1 Defining TreeView nodes

This section provides step-by-step instructions for working through the Department TreeView example. It also explains the elements of the dynamic TreeView. [Figure 9–2](#) shows a picture of what you are building so you can better understand the following pieces as you create them:

- The Department Filter Viewer at the top lets you select a Department Code or Name.
- The five node levels of the Tree View itself (Department data node, Employees text node, Employee data node, Family member text node, and Family data node).
- A maintenance folder in the right pane of the window.

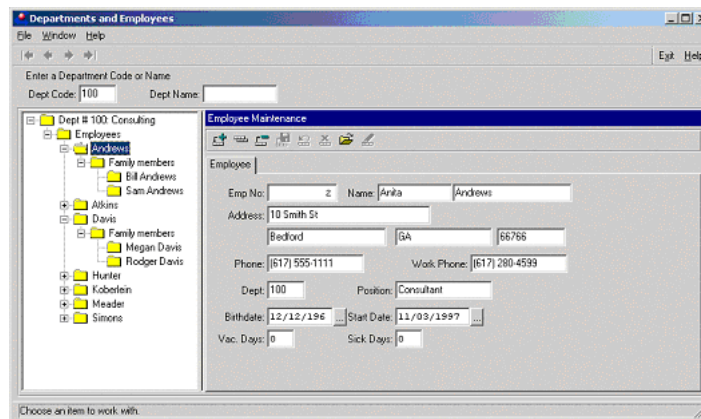
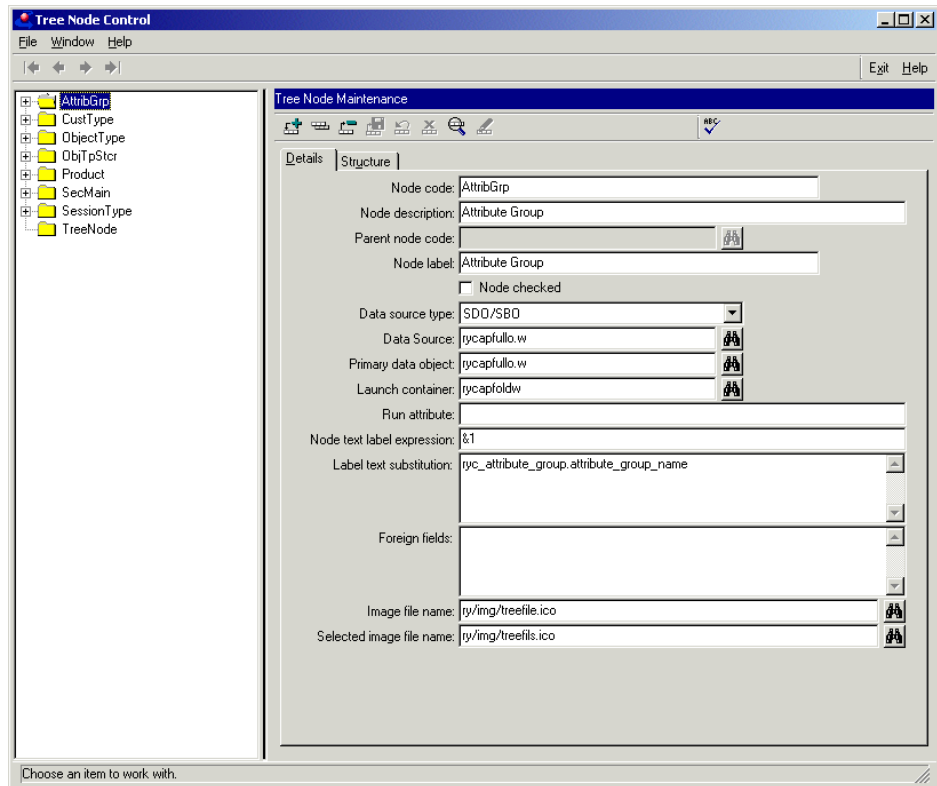


Figure 9–2: Completed example TreeView

Follow these steps to build the descriptions and to learn what they are for and how they relate.

- 1 ♦ In the AppBuilder, select **Build→Tree Node Control**:



- 2 ♦ Choose **Add** to create your first tree node. (It doesn't matter if something is currently selected, since you're creating a top-level node.)

As shown in [Figure 9–2](#), the three-level Department example requires you to define a total of five tree nodes. As you build a TreeView there is a node to represent the set of records at the next level down, and then another node to represent the individual records at that new node. So you will define nodes for each of the following records:

- Department
- Employees in a particular Department as a group
- Each Employee

- Family Members as a group
- Each Family member

Because each tree node defines both data at a certain level within the tree and also its relationship to the next level up, you must define a distinct node for each combination of a type of object or data that will be displayed at a level, and its relationship to the next level up. You will define a node for Departments with no parent relationship, because that will be the top of this TreeView. The Employee nodes will define themselves specifically as being for Employees of a Department, and the Family nodes as Family of an Employee. Other types of relationships for Employees or Family Members would require you to define new nodes.

Figure 9–3 shows an example of the TreeNode Maintenance frame.

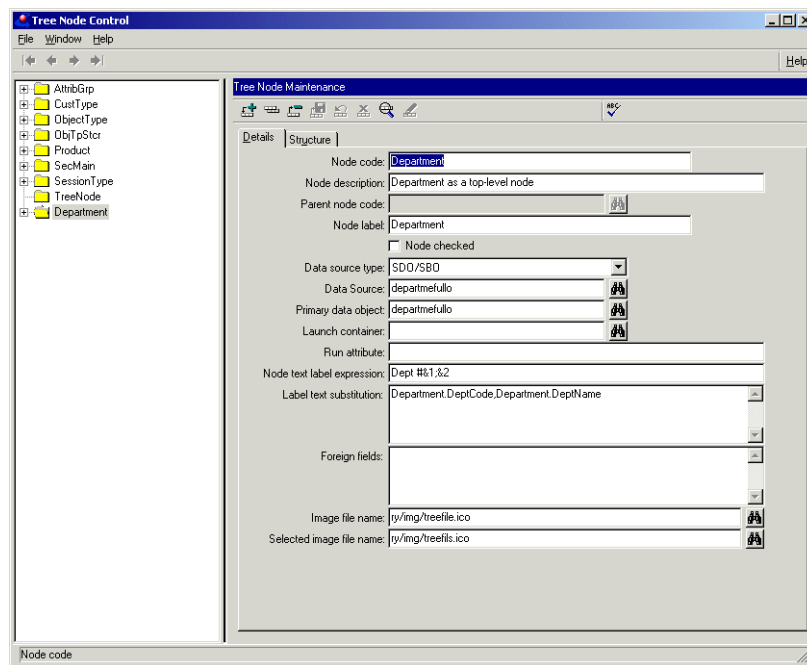


Figure 9–3: Tree Node Maintenance frame

Continuing the Department example

To complete the Department example, you need to define four more tree nodes, alternating data and text nodes. The next node down from Department is a text node that just displays the label “Employees.” It acts as a header for the individual Employee nodes of the selected Department.

To add the node, follow these steps:

- 1 ♦ Save the Department node.
- 2 ♦ Choose **Add** in the Tree Node Control window, then follow these steps:
 - a) For the Node Code, enter **EmployeeTx** (format is currently limited to ten characters).
 - b) For the Parent Node Code, enter **Department**, the node you defined earlier.
 - c) For the Node Label, enter **Employees**.
 - d) For the Data Source Type, select **Plain Text**.
 - e) For the text to be displayed next to the node, select **Employees**.
 - f) The Primary SDO names the same SDO used by the level above, the one that will be providing key values to the next node down. For the Primary SDO, enter **departmefullo**.

This node will not launch a container window on the right side because it does not show individual record values. So none of the remaining fields apply, except for the Image File Names, which would normally be the same choices as for any other node.

- 3 ♦ Save this node, then choose **Add** again.
- 4 ♦ Follow these steps to define the node for Employee records:
 - a) For the Node code, enter **Employee**.
 - b) For the Parent Node Code, enter **EmployeeTx**.
 - c) For the Node Label, enter **Employee**.
 - d) For the Data Source Type, select **SDO/SBO**.
 - e) For both the Data Source and Primary SDO, select **employeefullo**.
 - f) For the Launch Container, select the name of the window defined for Employee maintenance, **EmployeeFoldWin**.

- g) To display the Employee's last name next to the node, enter **&1** for the Node Label Expression and **Employee.LastName** for the Label Text Substitution.

Because this node uses the Department table as a parent, you list the Foreign Fields that define the foreign key relationship. Note that because the EmployeeTx Text node is placed in between the Department data node and the Employee data node, that intervening EmployeeTx node repeats the name of the parent Data Source. It is then the next node down, the Employee node that actually defines the foreign key relationship between the parent and itself.

- h) For the Foreign Fields value, enter **Employee.DeptCode,DeptCode**.

5 ♦ Save this node, then choose **Add** again.

6 ♦ Follow these steps to define another Text node that goes between Employee and Family:

- a) For the Node Code, enter **FamilyTxt**.
- b) For the Parent Code, enter **Employee**.
- c) For the Node Label and Plain Text, enter **Family Members**.
- d) For the Data Source Type, select **Plain text**.
- e) For the Primary SDO, select **employeefullo**.

7 ♦ Save this node, then choose **Add** again.

8 ♦ Follow these steps to define the Family data node:

- a) For the Node Code, enter **Family**.
- b) For the Parent Node Code, enter **FamilyTxt**.
- c) For the Node Label, enter **Family Member**.
- d) For the Data Source Type, select **SDO/SBO**.
- e) For the Data Source and Primary SDO, enter **familyfullo**.
- f) For the Launch Container, enter **FamilyFoldWin**.
- g) For the Node Label Expression, enter **&1**.

- h) For the Label Text Substitution, enter **Family.RelativeName** to display the name of family member.
- i) For the Foreign Fields, enter **Family.EmpNum, EmpNum**.

9 ♦ Save each of the tree nodes in turn, then exit the Tree Node Control.

Once you have defined all the tree nodes needed by a TreeView window, you can define the window itself.

NOTE: These tree nodes define a particular hierarchy. Each node level defines its relationship to the next level above. Thus to define a different TreeView window using some of the same tables, you will need to define a new set of tree nodes.

9.2.2 Defining the TreeView window

Now you are ready to assemble these nodes into a completed dynamic TreeView window.

Figure 9-4 shows the various styles of trees that you can build.

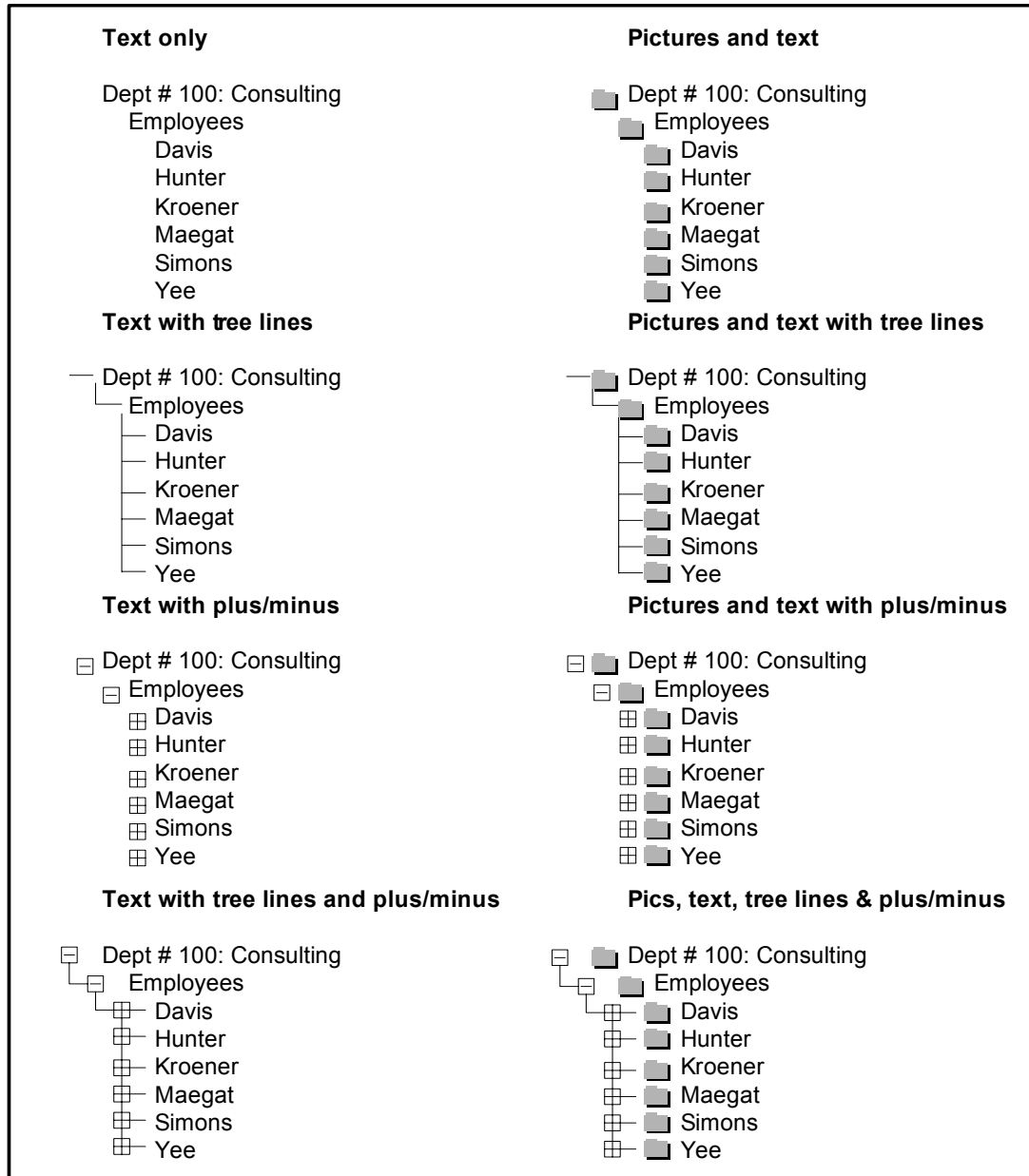
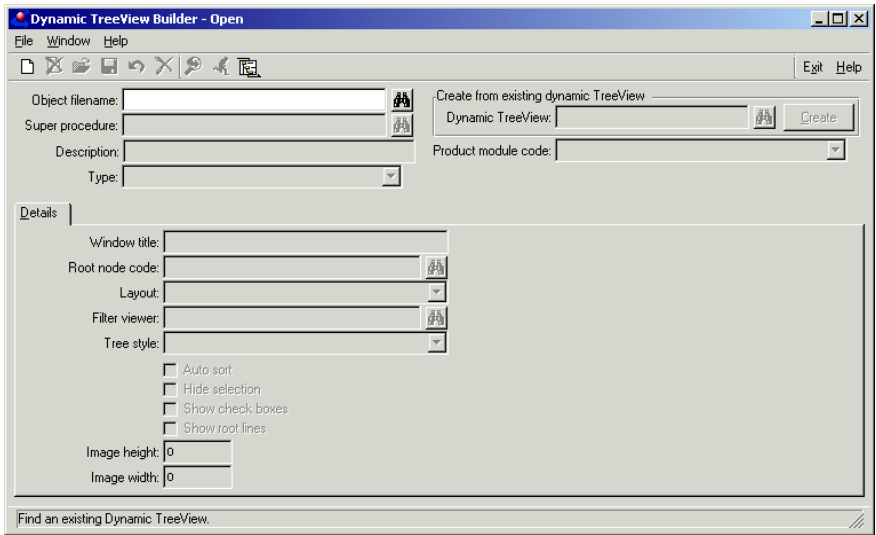


Figure 9-4: Examples of TreeView styles

In the example, you will use the style that shows all the available elements (Pictures and Text with Tree Lines and Plus/Minus).

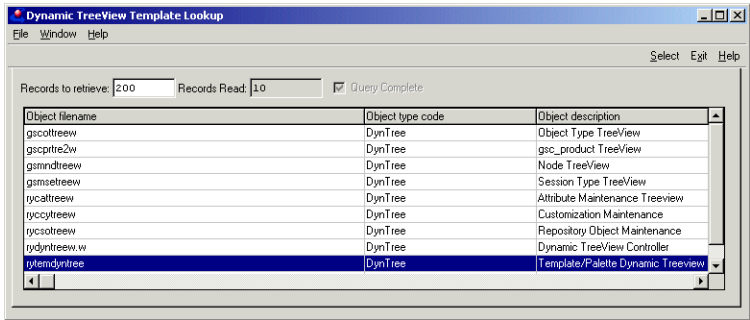
Follow these steps to build a TreeView window:

- 1 ♦ From the AppBuilder, select **Build→Dynamic TreeView Builder**:



- 2 ♦ Click the **New** button.

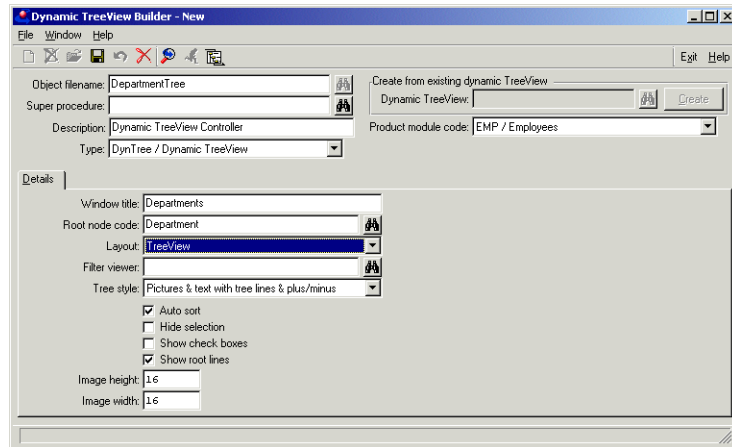
- 3 ♦ Choose the lookup button in the **Create from existing dynamic TreeView** section:



- 4 ♦ Choose **rytemdyntree** and click the Select button. This is the standard template for dynamic TreeViews.

- 5 ♦ Click **Create**.

- 6 ♦ Type a **Name** for the new TreeView, add a **Description**, and specify the EMP **Product Module**.
- 7 ♦ Type a **Window Title** and specify Department for the **Root Node Code**:



- 8 ♦ Click **Save**.

NOTE: The Type field is provided for specifying custom classes that extend the Dynamics TreeView class. If you create an extended class, make sure the **Layout Supported** flag is set to YES in the Object Type maintenance viewer.

9.3 Setting up structured nodes

A *structured node* is a node where each new level is created recursively from the same SDO. This means that you can expand a node any number of times and you do not have to set up individual nodes for each level in the tree.

This section explains the process of setting up a node for a structured table using the `gsm_node` table, illustrated in [Figure 9–5](#), from the Progress Dynamics Repository database (ICFDB) as an example.

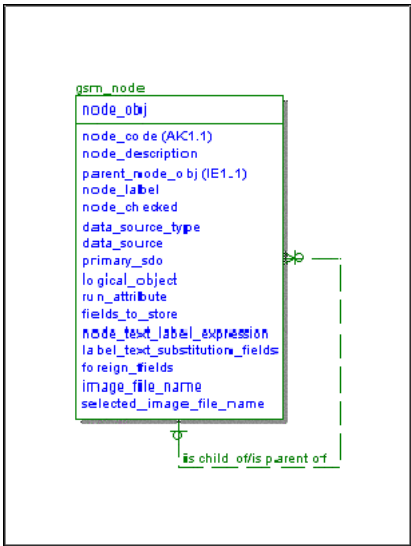


Figure 9–5: Repository `gsm_node` table

As you can see from this diagram, the `gsm_node` table is linked to itself and using the `parent_node_obj` and `node_obj` fields.

You can define a single node record to identify a node to expand indefinitely depending on its data. To do this you create a node record that indicates that the SDO can expand into child nodes. To illustrate, create a node in the Tree Node Control window using the values shown in Figure 9–6.

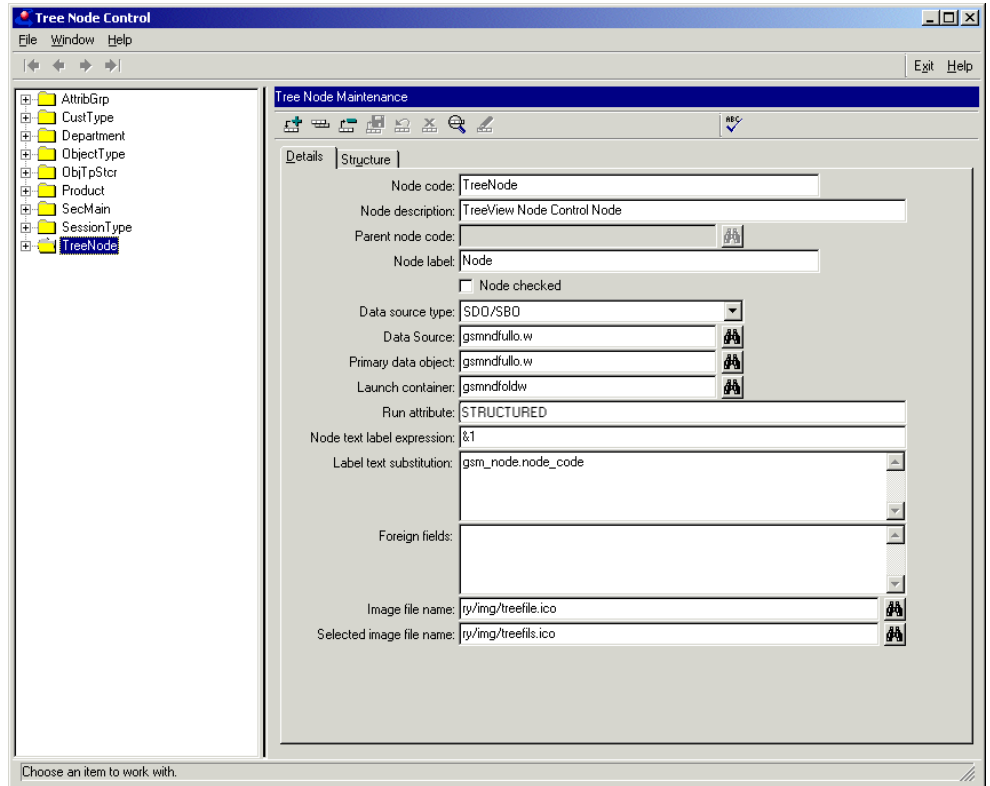


Figure 9–6: Tree Node Control window Details tab

Note the information in the Structured tab, as shown in [Figure 9–7](#).

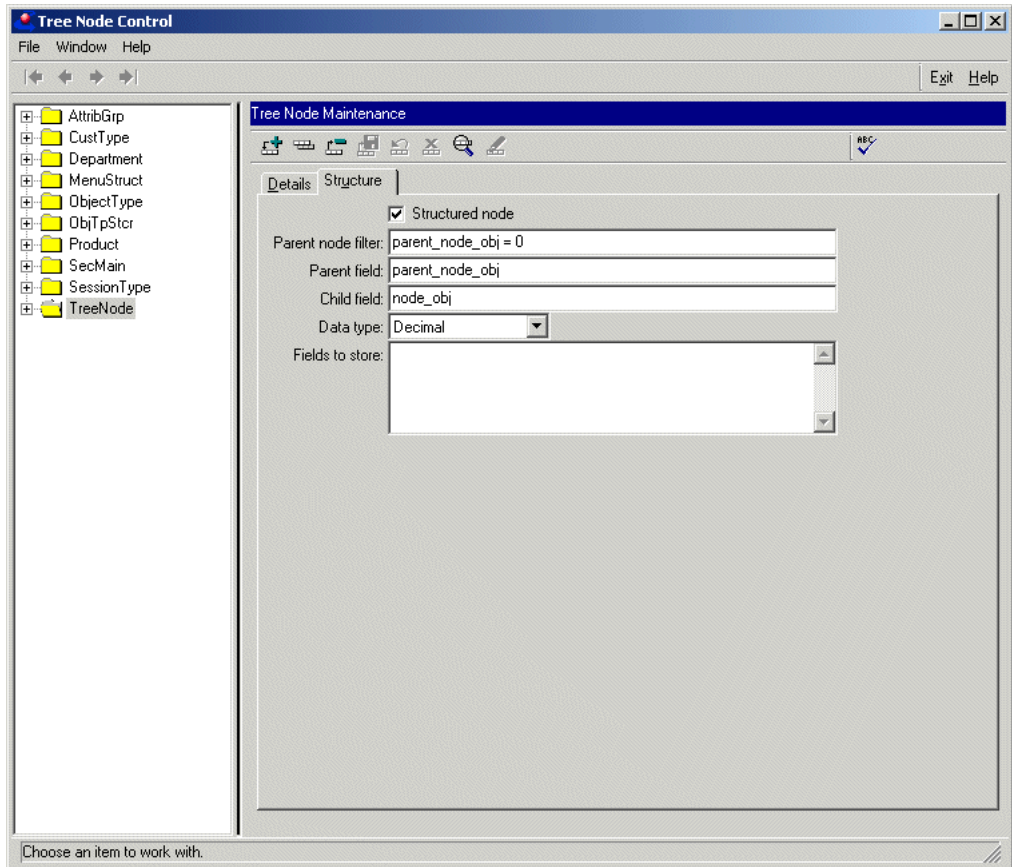


Figure 9–7: Tree Node Control window Structure tab.

The Structured node check box is enabled and identifies this node as a structured table node and allows related child nodes to be created.

When specifying values for a structured node, you must specify the following values in the following order:-

- Parent node filter query
- Parent key field
- Child key field
- Parent and child key field data type

For example, if you want to list only those nodes that do not have a parent (that is, nodes that are the highest nodes in the structure), the root node would filter only on records where the `parent_node_obj = 0`. Since a node only has a value in the `parent_node_obj` field if it is a child of another node, you can specify the `parent_node_obj` as the parent key field. A child node is linked to its parent by checking that the parent's key field value (`node_obj`) is equal to the `parent_node_obj` field for any other nodes. This makes the `node_obj` field the table's child key field. Where both of these key fields have a data type of DECIMAL, the fourth value would be DECIMAL. Thus, the Fields To Store value would be:

`0^parent_node_obj^node_obj^DECIMAL`

Once you have set up all the other standard information, you can create a dynamic TreeView in the normal way by creating a record in the Dynamic TreeView Builder.

9.4 Using an extract program for a node

This section explains the requirements and precautions to take when you use an extract program to populate a dynamic TreeView's nodes from an extraction program.

The extraction procedure mechanism was designed to allow a developer to extract complex data from the database by allowing them to execute a procedure on a Progress AppServer and send this back to the client side, thus meaning that the extraction program is dependant on being connected to a database, and the database tables could be directly referenced.

This example uses the customer table from the Sports2000 database.

To create an extraction procedure, you can create a new Structured PLIP from the AppBuilder menu, as shown in [Figure 9–8](#), or add a procedure to an existing Structured PLIP.

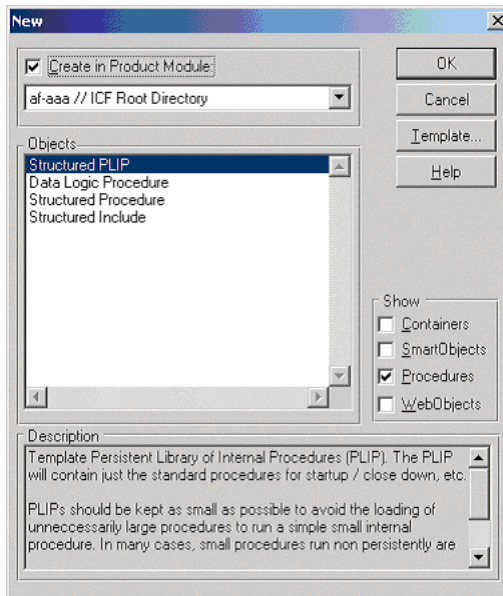


Figure 9–8: New dialog box

In this PLIP you need to add a procedure called `loadData`, shown in [Figure 9–9](#).

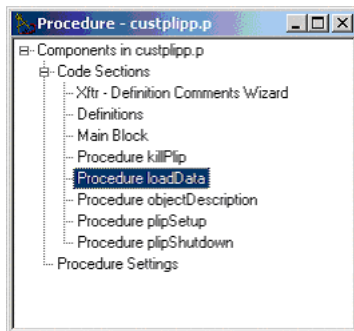


Figure 9–9: loadData procedure

This procedure accepts three input parameters and one input-output table handle to get and send the temp-table that is responsible for creating the nodes on the TreeView.

For this example, the extract program is run for the Root node of the TreeView and it will return the first 10 customers in the database. The procedure will look as follows:

```

/*-----
Purpose:      Extracts the first 10 customer records from the DataBase
Parameters:   <none>
Notes:
-----*/
DEFINE INPUT  PARAMETER pcParentNodeKey AS CHARACTER NO-UNDO.
DEFINE INPUT  PARAMETER pcPrimarySDO   AS CHARACTER NO-UNDO.
DEFINE INPUT  PARAMETER pcFilterValue   AS CHARACTER NO-UNDO.
DEFINE INPUT-OUTPUT PARAMETER TABLE-HANDLE phTable.

DEFINE VARIABLE hBuf                AS HANDLE NO-UNDO.
DEFINE VARIABLE hParentNodeKey      AS HANDLE NO-UNDO.
DEFINE VARIABLE hNodeKey            AS HANDLE NO-UNDO.
DEFINE VARIABLE hNodeLabel         AS HANDLE NO-UNDO.
DEFINE VARIABLE hPrivateData        AS HANDLE NO-UNDO.
DEFINE VARIABLE hRecordRef          AS HANDLE NO-UNDO.
DEFINE VARIABLE hRecordRowid        AS HANDLE NO-UNDO.

/* Grab the handles to the individual fields in the tree data table. */
ASSIGN hBuf                = phTable:DEFAULT-BUFFER-HANDLE
      hParentNodeKey      = hBuf:BUFFER-FIELD('parent_node_key':U)
      hNodeKey            = hBuf:BUFFER-FIELD('node_key':U)
      hNodeLabel         = hBuf:BUFFER-FIELD('node_label':U)
      hPrivateData        = hBuf:BUFFER-FIELD('private_data':U)
      hRecordRef          = hBuf:BUFFER-FIELD('record_ref':U)
      hRecordRowid        = hBuf:BUFFER-FIELD('record_rowid':U).
FOR EACH Customer
  WHERE Customer.CustNum >= 1
  AND   Customer.CustNum <= 10
  NO-LOCK
  BY customer.custNum:
  hBuf:BUFFER-CREATE().
  ASSIGN hParentNodeKey:BUFFER-VALUE = pcParentNodeKey
        hNodeKey:BUFFER-VALUE      = STRING(Customer.CustNum, "999999":U)
        hNodeLabel:BUFFER-VALUE    = STRING(Customer.CustNum) + " (" +
  TRIM(Customer.NAME) + ")"
        hRecordRef:BUFFER-VALUE     = TRIM(STRING(Customer.CustNum))
        hRecordRowid:BUFFER-VALUE   = ROWID(Customer)
        hPrivateData:BUFFER-VALUE   = pcPrimarySDO.
END.
END PROCEDURE.

```

Parameters

pcParentNodeKey

If the node level being created is a child of another node, this parameter will contain the unique identifier of its parent node. If the node being created is the Root node, it will be blank. This value **must** be assigned to the new record's `parent_node_key` to ensure that the build routine knows where to put this node.

pcPrimarySDO

Since an SDV needs a data source—normally in an SDO you need to tell it what SDO needs to be launched to allow this functionality. This SDO must contain the tables used in your extraction procedure and **must** have—as the first table in its query—the table whose ROWID was stored in the `record_rowid` field. This value can be assigned to the `private_data` field of the new record being created.

pcFilterValue

If a filter viewer is used on your Dynamic TreeView the value being passed from the filter viewer will be received through this parameter. You will need to dissect this field if you want to use these values in your query.

phTable

This is the temp-table's handle used to populate the TreeView with its nodes. You can use this handle to query any existing records in the TreeView as well as adding new records for nodes to be created. If your extract procedure is not the root node in the Dynamic TreeView you will need to find the parent node record using the `pcParentNodeKey` value. You get the value from either the `record_ref` field, which is the table's key field, or you use the `record_rowid` to find your parent record and the only filter on its values when building its child nodes.

As you can see from the example provided, you must populate the following fields:

- `parent_node_key`
- `node_key`
- `record_ref`
- `record_rowid`
- `private_data`

CAUTION: When assigning the value of the `record_ref` field, you must make sure that you specify the value of the entity key field/entity object field as specified in `gsc_entity_mnemonic` table. If you do not, an incorrect record in the SDO is selected. When creating a node record, be sure that you specify a valid SDO name in the Primary SDO field to accompany your extract program.

9.5 Creating a filter viewer for a TreeView window

You can create a custom Viewer to display at the top of the TreeView window. This Viewer can contain any combination of fields appropriate to let the user select one or more records to start the TreeView at the top level. It is important to allow some sort of filtering if the number of records at the top level is not small (ideally one in most cases). The TreeView control is an ActiveX control independent of the Progress 4GL. To populate it, a TreeView method must be called repeatedly for each new node. If there are hundreds of nodes in the tree, it will be very slow to initialize. Also, a TreeView with many nodes is unlikely to provide an effective user interface, since the user will have to scroll around in the TreeView so much that it will be difficult to locate the nodes of interest. In a sense, this applies to **all** levels of the tree. If any node defines a data set that could be many records for a single record at the next level up, then the TreeView might not be the appropriate visualization to use.

As described earlier, the TreeView can receive foreign key values from another object and use them to identify the starting record at the top level. If this is not the case, then a filter Viewer is likely the best mechanism to use to identify where to start.

Create the filter Viewer as you would any other static Viewer in the AppBuilder.

Follow these steps to build the Viewer:

- 1 ♦ Select the SDO for the top-level node.
- 2 ♦ Choose one or more fields from the table to use in filtering. In some cases, you might not wind up using **any** fields from the table itself. This might happen if, for example, you want to define two fill-ins to use in a From-To range check or some other mechanism that does not directly use the fields in the database table. If this is the case, then select one field in the Viewer wizard, and then delete it when you are done with the Wizard.
- 3 ♦ Back in the design window, you can define whatever fill-ins and other objects you need.

For the example, you build a Viewer on the Department table that has the DeptCode and DeptName fields in it. When the user enters either a Department Code **or** a (possibly partial) Department Name and leaves that field, the Viewer notifies the TreeView procedure and the TreeView is populated. The notification is done by means of a special published event defined for the TreeView, `filterDataAvailable`.

Figure 9–10 shows what the Viewer looks like (with a text prompt added to it).

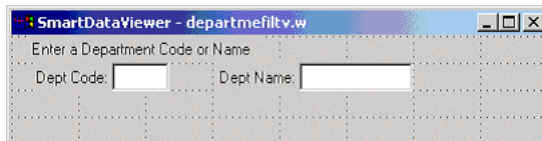


Figure 9–10: Example SmartDataViewer

First you must ensure that the filter fields will be enabled when the TreeView comes up. To do this, you need to intercept the initialization of the Viewer and add a statement to enable the fields, as described in the following steps:

- 1 ♦ In the Section Editor of the AppBuilder, select **Procedures**→**New**→**Override**.
- 2 ♦ Select the procedure name `initializeObject`. This is the standard named event that is run whenever this or any other SmartObject is initialized. By default, the procedure runs its standard code by executing a RUN SUPER statement, as the standard behavior is defined in a set of super procedures associated with this object.

- 3 ♦ Add the ENABLE statement to this procedure. The standard Progress frame name for all visual SmartObjects is encoded in the preprocessor value {&FRAME-NAME}, so you qualify the reference with that name. Also, because the Viewer is using the same temp-table definition to represent the data as it is sent from server to client by the SDO, you need to qualify at least the first field reference with that temp-table name, RowObject as shown in the following code example:

```

/*-----
Purpose:  Make sure the fields in the Viewer are enabled
Parameters:
Notes:
-----*/

/* Code placed here will execute PRIOR to standard behavior. */

RUN SUPER.
ENABLE RowObject.DeptCode DeptName WITH FRAME {&FRAME-NAME}.
/* Code placed here will execute AFTER standard behavior. */

END PROCEDURE.

```

NOTE: If you are not already familiar with building applications with SmartObjects, you can learn a great deal more from the Version 9 documentation.

Next you must add the code to respond to the data the user enters and pass it on to the TreeView. The TreeView is subscribed to the `filterDataAvailable` event in this Viewer. All you need to do is publish that event with the proper parameter. The way you have defined the filter Viewer, the user can enter either a Department Code or a Department Name. So you must define a LEAVE trigger for both of those fields. First, the DeptCode trigger checks that the user entered a value and publishes `filterDataAvailable`, passing a single input parameter. This parameter is a string consisting of the following values, in a comma-separated list:

- The name of the field in the SDO set to filter on. If there is more than one field, you must delimit the list of field names with the `CHR(1)` character (because the comma is already used to delimit the elements of the parameter as a whole).
- The SCREEN-VALUE (that is, the value in Character form) of the fields. Likewise, you must use `CHR(1)` as a delimiter if more than one value is involved.
- The comparison operator to use to compare the field to the value. This operator can be `=`, `>`, `>=`, `<`, `<=`, `BEGINS`, etc.

Follow these steps to improve the user interface at run time:

- 1 ♦ Before publishing the event, invoke the SESSION method SET-WAIT-STATE, which changes the mouse cursor. The General setting changes it to an hourglass to let the user know that the TreeView is processing the request.
- 2 ♦ After the event, set the wait state back to "", which restores the mouse cursor to what it was before. The following example shows the DeptCode trigger:

```
DO:
  IF RowObject.DeptCode:SCREEN-VALUE NE "0" AND
     RowObject.DeptCode:SCREEN-VALUE NE "" THEN
  DO:
    SESSION:SET-WAIT-STATE("General").
    PUBLISH "filterDataAvailable"
      (INPUT "DeptCode," + RowObject.DeptCode:SCREEN-VALUE + ",=").
    SESSION:SET-WAIT-STATE("").
  END.
END.
```

- 3 ♦ Define a similar trigger for the DeptName field. Specify BEGINS as the operator so that the user can enter a partial value for the Department Name:

```
DO:
  IF RowObject.DeptName:SCREEN-VALUE NE "" THEN
  DO:
    SESSION:SET-WAIT-STATE("General").
    PUBLISH "filterDataAvailable"
      (INPUT "DeptName," + RowObject.DeptName:SCREEN-VALUE +
        ",BEGINS").
    SESSION:SET-WAIT-STATE("").
  END.
END.
```

There are several different places in the Viewer code where this trigger might belong, depending on the objects in the Viewer:

- If the user can enter values in two or more fields, and the filter should use all of them, then you might add an Apply button to the Viewer. The Progress Dynamics tools often use an Apply button to respond to all the field settings together. In this example, the `filterDataAvailable` event is published in the CHOOSE trigger for the button.
- If you use a dynamic Lookup to allow the user to select a value, then place the publish statement in a special procedure defined for Lookups called `lookupDisplayComplete`. In this case, the code for a Lookup would look like this:

```

/*-----
Procedure: lookupDisplayComplete
Purpose: Apply the selection of a value in a Lookup to the TreeView.
-----*/
DEFINE INPUT PARAMETER pcFieldNames AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER pcFieldValues AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER pcDataValue AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER phLookup AS HANDLE NO-UNDO.
PUBLISH "filterDataAvailable"
    (INPUT "DeptCode," + RowObject.DeptCode:SCREEN-VALUE + ",=").
END PROCEDURE.

```

NOTE: The input parameters to the procedure are not used in this case but need to be there to satisfy the calling sequence.

- If you used a Combo object to let the user select a value, add the PUBLISH event in the VALUE-CHANGED trigger for the Combo.

9.6 Use Rows To Batch with large numbers of nodes

Instead of using a filter viewer, you can use an alternate technique to handle treeviews with large numbers of nodes. The dynamic TreeView batches large data sets by using the RowsToBatch property set at the dynamic TreeView level. If more than the set rows to batch records could be found a **...More node** will appear as the last node in the TreeView to allow the user to get the next batch of records from the server. The default for this property is set to 50 records. To change it, access the RowsToBatch property using the Dynamic property sheet in the Dynamic TreeView Builder. If the RowsToBatch property is set 0 (zero), then the Dynamic TreeView will revert back to the retrieving all the records for the TreeView.

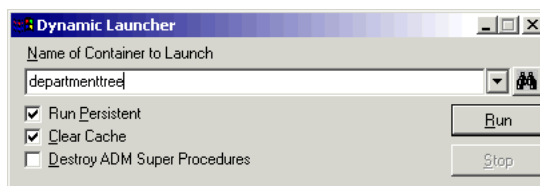
CAUTION: If the RowsToBatch property is set (other than zero), the AutoSort option is automatically turned off and the sorting of the nodes will depend on the SDO's sort order. If the AutoSort option was checked when the TreeView was built, the framework attempts to sort the nodes manually, but there is a possibility that the nodes might not always be sorted in the correct order. If you set the RowsToBatch to zero and the AutoSort option is checked on, the TreeView will execute standard sorting behavior.

9.7 Running the TreeView window

You can run the TreeView window as you would any other dynamic window in your application. You could, for example, invoke it from a menu item.

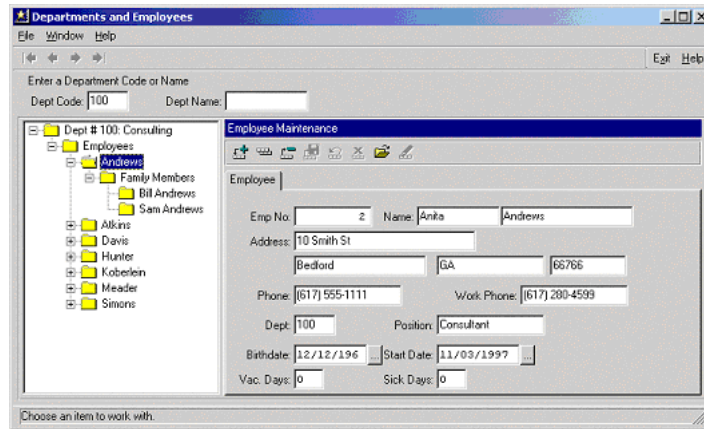
To test your window out of the context of your application, follow these steps:

- 1 ♦ From the AppBuilder, select **Compile→Dynamic Launcher**:



- 2 ♦ Run the Department TreeView window. It comes up empty.

- 3 ♦ Enter a value in the filter Viewer. It populates itself. For example, if you select Department #100, you will see a single node representing that record.
- 4 ♦ Expand that node by selecting the plus sign or double-clicking on the folder icon.
- 5 ♦ Expand the Employees node to show a list of all employees for that department.
- 6 ♦ Expand an employee's node to show Family Members for the employee:



At each level the pane to the right shows the maintenance window for the data at that level. Remember that you can reapportion the space between the TreeView on the left and the maintenance pane on the right by positioning the cursor between the two until it shows a left-right cursor, and then dragging the dividing line between the two parts of the window.

9.8 Building a menu structure TreeView window

You can also use the dynamic TreeView layout as a way to represent a menu structure. In this way the hierarchy of your menu, with potentially multiple levels, appears represented as a tree on the left side of your application window. The dynamic windows that are launched from the items in your menu structure are all realized as frames in the panel on the right side of the window, just as you saw in the earlier example.

If these windows in turn launch other separate windows, then these will still appear as separate windows rather than as frames in the TreeView window. You can control this if you wish by representing the multiple frames as tabs in a single tab folder rather than as separate windows.

Creating a TreeView window for a menu is essentially the same as creating one for a node representing data. As an example, follow these steps to construct a TreeView window for the Progress Dynamics Administration's Security menu:

- 1 ♦ From the AppBuilder, select **Build→Tree Node Control**. The node you create will represent the menu structure.
- 2 ♦ Enter a Node Code and Description for the node. The example calls the node SecurityNd.

This example has no Parent Node Code. Also, because the menu item label will be displayed as the identifying label for the item in the TreeView, there is no need to enter a Node Label; it will not be displayed.

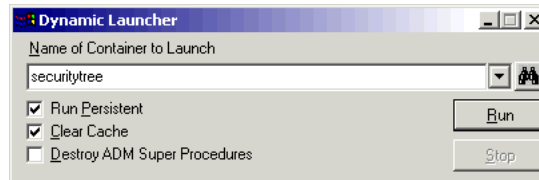
- 3 ♦ For the Data Source Type, select **Menu Structure**.
- 4 ♦ Select the **Menu Structure Code** for the menu structure you want to represent. In this example, it is ICFAF-Secu, the Administration Security menu.

None of the other fields for the node maintenance apply, except possibly the Image File Names, if you want to use them. At this time, only open and closed folder images are available, and there is not yet a maintenance utility to add images to the Repository. For this example you can leave the Image File Names blank.

- 5 ♦ Choose **Save** to save this node description.
- 6 ♦ Exit the Tree Node Maintenance.
- 7 ♦ Exit the Tree Node Object Control.
- 8 ♦ From the AppBuilder, select **Build→Dynamic TreeView Builder**.
- 9 ♦ Choose the **Add** button to display the Dynamic TreeView Builder Maintenance window.
- 10 ♦ Select a **Product Code** and **Product Module Code**. For the example, you could select the General module of the Sports application, if you have that definition available from the *Getting Started with Progress Dynamics* book's exercise.
- 11 ♦ Give the TreeView object a name and enter a description. For the example, use SecurityTree.

- 12 ♦ Enter a Window Title to appear at the top of the TreeView Window. For the example, it is Progress Dynamics Security.
- 13 ♦ Enter the name of the Node Code you assigned in the node maintenance. The example uses SecurityNd.
- 14 ♦ Because the example does not use the folder images for the Security node, select **Text Only with Tree Lines** as the Tree Style.
- 15 ♦ Save this information.
- 16 ♦ Exit the Dynamic TreeView Builder.
- 17 ♦ In the Dynamic TreeView Builder Control, select **Options→Generate Dynamic Object**, then confirm that you want to save this object and that you want to delete any earlier version of it.
- 18 ♦ Exit the Dynamic TreeView Builder.
- 19 ♦ Now you can run your Security Menu TreeView:

- a) Bring up the Dynamic Launcher:



- b) Choose **Run**.

Note that no images appear with the items because you defined the TreeView that way. Also note that the labels of the menu items appear as the labels of the individual items in the tree.

NOTE: When programming with dynamic treeviews, note that the content of the treeview sits in a dynamic frame. To get the handle of the treeview when working with an object within it, get the ContainerSource of the object. This gives you the dynamic frame. Get the ContainerSource of the dynamic frame, and you have the treeview handle.

9.9 Summary

As you have seen, the dynamic TreeView window lets you represent a wide variety of application information in a single window, managed by a hierarchical visual representation of the data or the application structure.

You can use the API of this object to incorporate into any type of application window for which you would like to use it.

Building Basic Business Logic in a Progress Dynamics Application

The Progress Dynamics framework provides a structure in which you can write the business logic that is the heart of your application. The framework enables you to write your logic in a consistent and reusable way that supports running your application efficiently in a distributed environment. This structure is an extension of the structure in Progress Version 9 ADM2 SmartDataObjects (SDOs) and Progress SmartBusinessObjects (SBOs). You can use existing Version 9 SmartObjects with Progress Dynamics without modification. The extensions provided for the first time in Progress Dynamics increase the reusability of your business logic. They provide a standard way to incorporate existing complex data management procedures into the framework.

The ADM2 documentation provides detailed reference material on the SDO and its properties. Even if you are not yet familiar with the Version 9 objects, this chapter provides sufficient background so that you can work successfully with them in Progress Dynamics. An SDO is an object that manages a database query, transfers data between server and client, and provides hooks for data validation logic. An SBO is a compound object that manages related data for two or more SDOs, in a one-to-one or parent-child relationship.

This chapter discusses general programming and design issues for a distributed application. In particular, it covers business logic procedures that go beyond the table validation logic normally written into SDOs. It also discusses the Progress Dynamics Messaging system, which lets you define messages in the Repository, translate them, and use them in your applications.

This chapter includes the following sections:

- [Writing distributed applications](#)
- [The Progress Dynamics SmartDataObject](#)
- [Strategies for SmartDataObject query definition](#)
- [Standard validation procedures for SDOs](#)
- [The SDO logic procedure](#)
- [Message handling in Progress Dynamics](#)
- [Summary](#)

10.1 Writing distributed applications

The basis of the logic structure of the ADM2, which carries over into Progress Dynamics, is to support running applications in a distributed environment using the Progress AppServer technology. There are several reasons for this approach.

First, consider a client/server system with an application interface running on an MS Windows machine and a database running on a separate server machine. This kind of system can have serious performance limitations. All the records involved in running the application are passed from server to client, including records read to execute schema trigger procedures. Every CAN-FIND or other database access operation requires a message between client and server. The major goals of the Progress AppServer are:

- To help you minimize this traffic, by allowing procedures that require no user interface to execute entirely in a Progress session running on the server machine.
- To give you control over exactly when data is transferred between client and server sessions, and when other requests from client to server occur.

The Progress Dynamics framework handles this data transfer and most other interactions between client and server in a standard way. The framework optimizes the interaction so that you do not need to modify it in most cases.

Another reason for using the AppServer is that it makes the application environment more flexible. In particular, using the Progress WebClient product, the client application can run without any database connection. Progress Dynamics supports the use of AppServer to enable you to deploy your application in this configuration or in new ones yet to be developed, without having to change your fundamental business logic.

However, building an application that uses AppServer involves a very different mind-set from developing applications that run host-based or client/server. In fact, some useful features of the Progress language and how it interacts with the database are not appropriate in a distributed application. This can be a painful realization. You need to think about it when designing or converting applications for this new environment.

Some of these differences center around record access and transaction scoping. Others arise because database data is not directly available on the client. Still others arise because an application that uses AppServer involves two or more completely independent Progress sessions running on different machines. These are all things that you have to consider in your development. Progress Dynamics is designed to handle as many of these kinds of issues as possible for standard situations. But, there are always cases where you have to design and program with the characteristics of a distributed application in mind. The following sections discuss a few of these issues.

10.1.1 Client access to the database

It is entirely possible and “legal” to maintain a direct client/server connection to the application database while using AppServer for some kinds of operations. Many applications have been built this way, some using the direct connection to read data in bulk, to populate browses for instance, while using the AppServer connection for updates, so that the execution of schema trigger procedures and other server-side logic does not require passing records to the client. In some cases, this can be effective. However, even without considering the future deployment options for your application, this technique can lead to serious performance limitations.

The Progress Dynamics model maintains the principle that there is never a database connection on the client. Observing this constraint will prepare your application so that you can deploy it using WebClient without any changes. WebClient lets you distribute and maintain your application worldwide without the need to deal with run time licensing issues and code maintenance for all your application’s users.

Observing this constraint also prepares your application for alternative client types, so that your application can be accessed through Progress WebSpeed®, or from a non-Progress client interface, or from future client platforms for which you might not yet be consciously programming.

But what restrictions are you facing when you give up your client database connection? Most obviously, your client-side code can no longer have explicit statements in it that access the application database. You cannot write FIND and FOR EACH statements against the database and cannot construct and open database queries. Your access to the database must be through temp-tables passed between client and server. Progress Dynamics handles this for you in most cases, allowing SmartDataObjects and SmartBusinessObjects to be divided transparently between client and AppServer sessions, and passing data as needed in both directions.

But there are subtler requirements as well. Most significantly, perhaps, the Progress compiler will automatically compile any field validation you have defined in the database schema into any frame containing those fields. This includes, of course, Progress SmartDataViewers, which are just Progress frames with supporting code attached. If this validation requires database access, such as a CAN-FIND on a foreign key value, then the procedure will generate an error at run time if there is no database connection. There are other issues with this as well, which are discussed in [Chapter 2, “Database Design Principles in Progress Dynamics.”](#) The recommendation is that you not use validation defined in your Progress schema, or at least not rely on it executing in your Progress Dynamics user interface. The Progress Dynamics Object Generator, by default, removes validation from the SDOs it generates, to help you avoid problems in this area.

The CAN-FIND validation is just an example of a client-side bit of business logic intended to provide immediate feedback to the user if a data value is not valid. Sometimes application developers, observing the guideline that there should be no direct client-side database connection, but still wanting to maintain the same level of user feedback, will insert calls to the AppServer on LEAVE of many fields in the application screens, to do a database lookup of some kind. This is not a good idea in most cases. As noted, a big part of the purpose of AppServer is to minimize the number of calls between client and server. Adding lots of extra calls defeats the purpose and works against good performance. The Progress Dynamics Lookup and Combo objects satisfy the need for this feedback. In addition, they provide the user with a list of valid values from which to choose. The values are brought over from the server in a single call to maximize performance. You can define other kinds of validation that do not require a database lookup in the SDO for the table in a way that executes that logic on the client. These kinds of validation are discussed in the “[Client-side data validation](#)” section. By default, this logic executes when the data entry for the record is complete. If you want to add code to run the client-side field validation in the SDO on LEAVE of a field, you can do so without forcing a call across the AppServer connection.

So in short, write your client application code so that it does not require any direct connection to a database. This goal will maximize your performance in a distributed environment and prepare you for successful deployment on the widest variety of client platforms. The Progress Dynamics objects are designed to support you in every way in achieving this goal.

10.1.2 Record locking and transaction scoping

Another significant feature of the Progress 4GL that loses much of its value in a distributed world is the whole area of record locks and transaction scoping. When your client application is looking only at data in temp-tables passed over from the server, you cannot hold record locks on the server while the user works with that data. You cannot easily scope a transaction from the client side that will update multiple records on the server in a single transaction. The whole way you think about and plan for data management changes somewhat because of this.

Data access in Progress Dynamics is always done based on “optimistic” locking. In a distributed application, you really have no other choice. Records are read NO-LOCK on the server, because you cannot hold record locks on the server as you pass the data to the client. In a stateless AppServer configuration, when the client session sends back an update, it will probably be executed in a different AppServer session anyway. So there is no built-in protection that someone else will not update the data while your user is looking at it. When an update comes back from the client, the server code that supports SDOs checks whether the data has been changed since you read it, and (unless you set a property indicating that you do not care) rejects the change.

In many cases, this might be satisfactory. Depending on the nature of the data, it is unlikely that two users will attempt to update the same record concurrently. Having an occasional update rejected, such that the user has to enter changes again, might be acceptable. Or with some data, it might not matter if two users change different parts of a record concurrently. In this case, you would set the `CheckCurrentChanged` property of the SDO to `NO` to indicate this.

But in other cases, you will want to maintain something like the record lock of a host-based or client/server application. To do this, you must build flag fields into your database that signal that a record or a set of related records is being updated, and write your application logic to observe this flag.

10.1.3 Minimizing AppServer calls from the client

Regardless of what programming technique you use, you are well advised to minimize direct extra calls to the AppServer wherever possible. If the code you write forces a lot of calls from client to server beyond those that are already done for you to retrieve data and return updates, it affects the performance of your application in a distributed environment. When you go to write such a procedure, consider whether it is really necessary to have the database access occur immediately, or whether it is satisfactory (or necessary for performance reasons) to allow the code to be executed when values go back to the server anyway. Many calls from client to server are a likely sign of poor application design, and one that will not handle a distributed application or one with a non-Progress client as effectively as you wish.

The whole discussion of client-side and server-side logic in this chapter is based on the optimization of AppServer access. What can be done on the client without reference to the database should be done on the client, if this allows the application to provide feedback to the user without an AppServer call. What requires a database connection, or what does not specifically benefit from being executed on the client in terms of immediate feedback to the user, should be left in server-side procedures. This will not only reduce the number of calls to the server, but also reduce the amount of compiled code that must be deployed to client machines. The following section on the SDO and its logic procedure, as well as the section on business logic procedures, amplifies this basic principle.

10.2 The Progress Dynamics SmartDataObject

The SmartDataObject or SDO is at the heart of Progress Dynamics data management. This is the object that presents data to other client objects, accepts updates, performs application-specific validation, and gets data back to the database. It is the one standard SmartObject that is AppServer-aware. That is, it knows how to run itself on an AppServer and manage the passing of records back and forth between client and AppServer.

This section presents an overview of the SDO in case you are not already familiar with it. Since the SDO has been modified and enhanced in various ways for Progress Dynamics, some material will be of interest even to those who have already worked with Version 9 SmartObjects.

As of the first release of Progress Dynamics, the SDO was a static, or procedural object. Several Progress source files were generated for each SDO. The design was such, however, that it would be straightforward to migrate these static SDOs to become strictly dynamic, data-driven objects in this latest release. The only source procedure that has remained is the procedure where any custom logic goes. Keep this in mind as you read through this material.

10.2.1 SmartDataObject basics

This section reviews some of the basic characteristics of the SmartDataObject. You can create SDOs either by running a wizard in the AppBuilder or by running the Progress Dynamics Object Generator. Generally this section presumes that you have created SDOs initially using the Object Generator, although you can then edit and customize them in the AppBuilder.

The Object Generator creates an SDO for each table, with a database query selecting all the rows in that table. Customizing the SDO in the AppBuilder allows you to modify the query, which might involve a join of two or more tables, along with a specific list of fields from those tables to be used in the SDO. You can modify names and characteristics of these fields (such as format) in the SDO. You can define additional calculated fields to store values not directly represented by single database fields.

NOTE: Properties of calculated fields in Static SDOs are stored in the include file, not the Repository.

This field list is turned into a Progress temp-table definition whose name is RowObject. At run time, records are read out of the database tables into the RowObject temp-table. All other client-side objects (Viewers and Browsers and the like) access the values in this temp-table rather than reading and writing database records directly.

There are two primary reasons for using this temp-table as an intermediary for data access:

1. It allows you a degree of freedom in defining SDOs whose definitions are not tied directly to a particular database table definition. This means that if you change data definitions at a later time, you do not necessarily have to change application components to match them. You can rename fields between the database and the temp-table, and the SDO support code automatically makes sure that values are transferred properly between the database field and the temp-table field (and back again if you change the value). Fields in the database tables that are not needed or wanted in the application can be left out of the SDO definition. The SDO lets you mask joins between tables by presenting the join as a single logical table in the SDO.

2. SDOs are designed to support distributed applications. The Progress AppServer allows greatly increased efficiency of data access compared with client/server in many cases. In particular, this is true when the application's business rules require many side effects to an update operation, including reading additional database records to validate the data being modified, or reading and writing other records to reflect changes (updating totals or other calculations in other parts of the database, for example). Using the Progress AppServer moves all of that update logic onto the same machine with the database. This can greatly improve performance compared with using a client/server database connection to return all of the related records to the client for processing.

The SDO is specifically designed to run on a stand-alone client, or transparently divided between the 4GL client and AppServer, or on an AppServer accessed from a non-Progress client. If you follow the guidelines discussed in this guide, as well as in the ADM documentation and course material, your applications will be well positioned to run successfully in a wide variety of deployment environments.

This section outlines the process of running an SDO from the perspective of the RowObject temp-table and its update counterpart, RowObjUpd. This description presumes that the SDO is divided between client and AppServer. If the SDO is running entirely on the client, essentially the same steps occur, but all in a single procedure in a single Progress session.

NOTE: You do not need to understand all these steps or the procedures that are called before you begin building and using SDOs. However, as you begin to write data validation logic for your SDOs, it will be useful to understand how the data is moved around and the order in which things happen. The TransactionValidate procedures mentioned are internal procedure names to which you can attach validation logic to be executed at various points during the update process. These procedures are discussed in detail in the [“Server-side validation”](#) section.

10.2.2 Setting instance properties in SDOs

There are references in this material to instance properties that you can set for your SDOs. *Instance properties* are SmartObject properties or attributes (the terms are synonymous as used in Progress Dynamics). You can assign them as initial values of the Object to cause it to act in a particular way, either by default or when it is run in a specific container. In Progress Dynamics, default values for these property values are written to the Repository when the object is created, and then inherited for each occurrence of that object within a container window. You can modify these property values for the object itself, in which case they will be inherited by any use of that object in an application container window. Or you can modify the values for a particular instance, that is, as used in a specific window.

In either case, you can modify these values through the AppBuilder or using the Dynamic Properties sheet. This gives you access to all the records in the Repository that define objects.

Even though SDOs were static objects in the first release of Progress Dynamics, information about them was stored in the Repository. In particular, instance property settings are defined in the Repository for each occurrence of an object in a container window, and this includes static objects such as SDOs.

Object attributes are defined at several levels. Every Object Type (such as SDO) defines basic attributes and default values for them where appropriate. Each specific object of that type (such as the Sports database Customer SDO called `customerfull0`) defines values for other attributes that might not be given values at the Object Type level. Each instance of an object in a specific container (such as the instance of `customerfull0` in the dynamic window called `custbrowsewin`) can override any of those attribute values that are defined as instance properties.

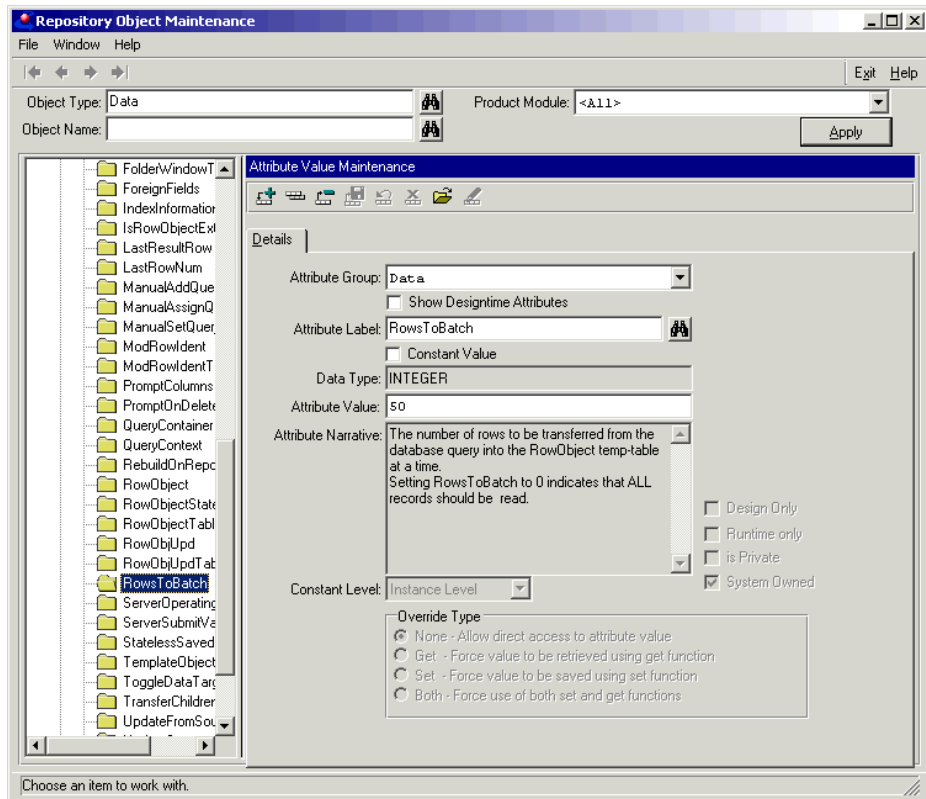
In this section you will go through examples of setting instance properties at all three levels: first for the Object Type SDO, then for a specific SDO, then for an instance of that SDO in a container.

Changes you make to class attribute values will automatically cascade down to existing instances of that class. Where attributes have been explicitly set at object or object instance level, they will not be overwritten. This is very important to keep in mind if you need to modify attributes yourself. If you want to change, for example, the number of rows sent from server to client in each batch for an existing window where that SDO is used, you must change the attribute in that specific instance. If you change the attribute in the object definition itself, it will affect all object instances where the attribute has not explicitly been set for that object or object instance.

Follow these steps to set attributes for an Object Type, a specific object, and an instance of that object:

- 1 ♦ From the Progress Dynamics Administration window, select **Object→Object Type Control**.
- 2 ♦ Select **Data** as the Object Type, then choose **Find**.
- 3 ♦ Expand to the next level and select **Attributes→RowsToBatch**.

- 4 ♦ This property has a default value of 50, which means that every SDO you create will have a **RowsToBatch** attribute with the value 50, unless you modify it. If you change this value to **20**, it changes that default for every SDO created after you make the change:



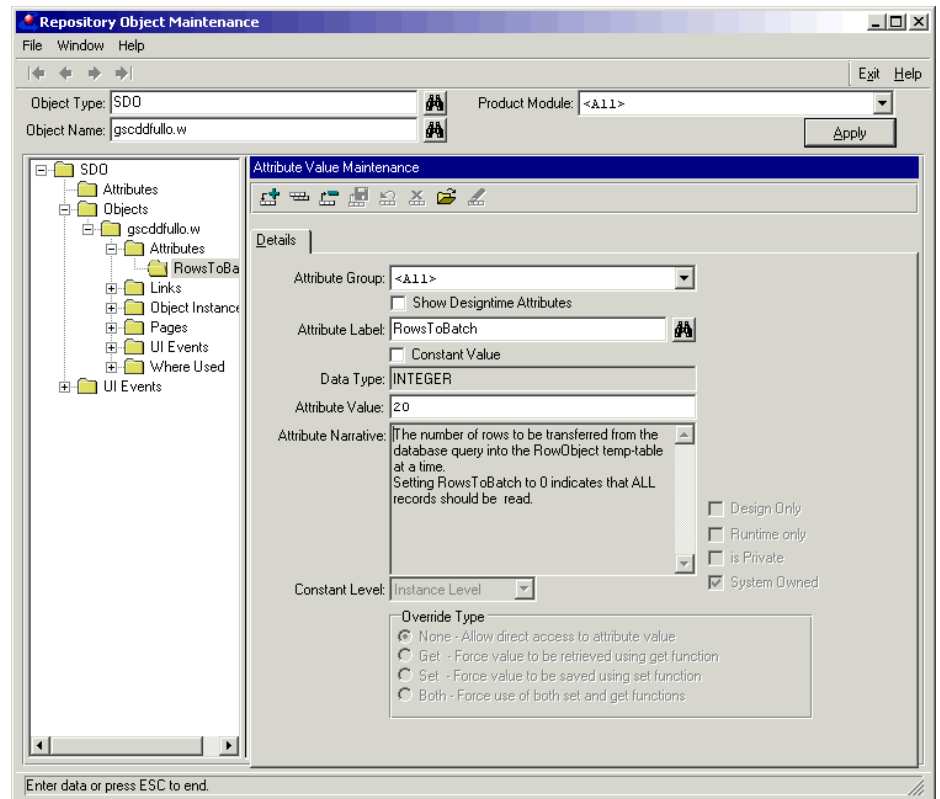
This change is shown mostly to demonstrate the levels at which attributes are defined. Exercise extreme care in making changes at this level because it could change the behavior of your entire application.

CAUTION: Never rename the instances of datafields in your SDOs.

Follow these steps to edit the same RowsToBatch attribute for a specific SDO:

- 1 ♦ From the Appbuilder, select the **File→Open Object** and open the specific Object to open from the Open Object window.
- 2 ♦ Click on the **Dynamic Properties** icon to open the Property Sheet for the selected SDO.

3 ♦ Find the attribute RowsToBatch and change the value to **20**:

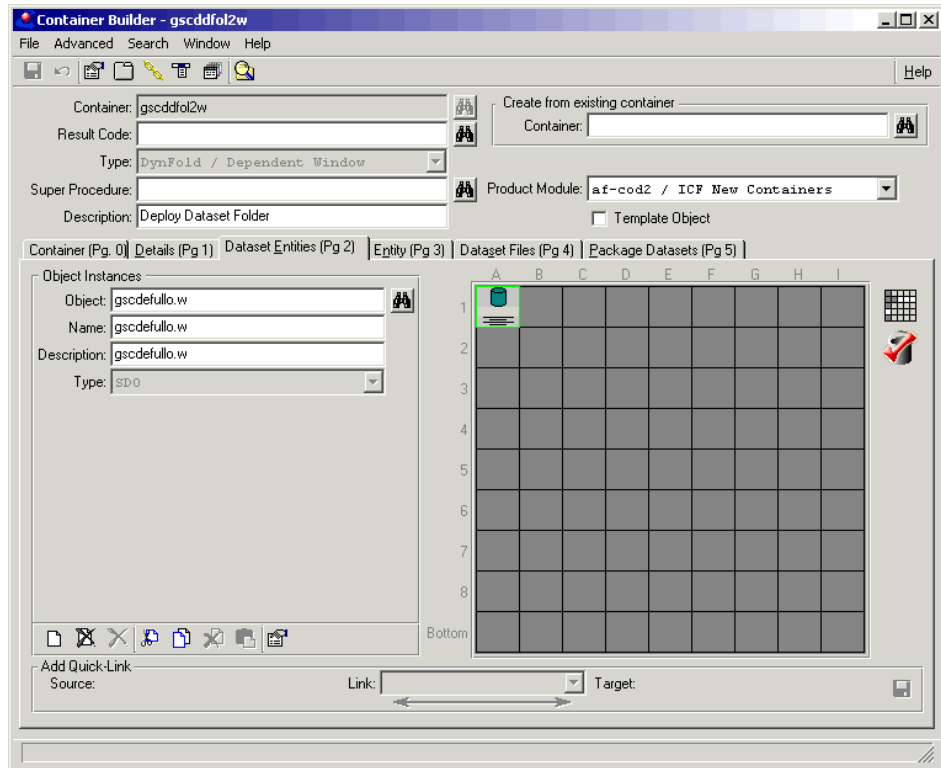


4 ♦ Close the Property Sheet.

Follow these steps to edit the same RowsToBatch attribute for a specific instance of an SDO:

- 1 ♦ From the Container Builder, open a Folder Window where an instance of the SDO exists.
- 2 ♦ Select the SDO and click on the **Dynamic Properties** icon for that SDO.

3 ♦ Change the attribute value for RowsToBatch to **20**:



4 ♦ Close the Property Sheet.

SDO startup

When a SmartDataObject is started from a Progress Dynamics window, the code in the SDO's `constructObject` procedure checks to see whether the databases required by the SDO are connected on the client. If they are, then the full SDO (`<SDOname>.r`) file is run on the client. If the databases are not connected, then the client proxy form of the SDO file (`<SDOname>_c1.r`) is run on the client. For more information on how the various forms of the SDO are generated and how they are constructed, see the [Progress Dynamics ADM2 API Reference](#). This client proxy form of the SDO contains all of the SDO procedures and functions that do not have any database references, and therefore do not require a database connection. You should view this as the standard arrangement: that there is no Progress database connection **of any kind** on the client. Using this standard will put you in the best position to deploy your application in a variety of situations in the future.

When the client SmartDataObject is run, its code checks to see if it has an AppService Instance Property defined. It then checks to see if that AppService is running locally (meaning that procedures in that AppService should be run in the client process) or remotely. If it is local, then the ASHandle property is set to the local SESSION handle. In later calls, running an internal procedure in the ASHandle will run it in THIS-PROCEDURE, the client-side SDO. If it is remote, the SDO runs the full version of itself (<SDOname>.r) on the AppServer as a persistent procedure and sets the ASHandle property to that remote persistent procedure handle. The SDO now coordinates passing database rows back and forth between client and AppServer.

10.2.3 Data management in the SDO

When the database query is opened, a batch of rows is read into the RowObject temp-table on the server-side SDO, as illustrated in Figure 10–1. This temp-table is then returned to the client-side SDO as an output parameter to the serverSendRows procedure call. This client-side temp-table can then be browsed by SmartDataBrowsers, and individual row values are displayed in one or more SmartDataViewers.

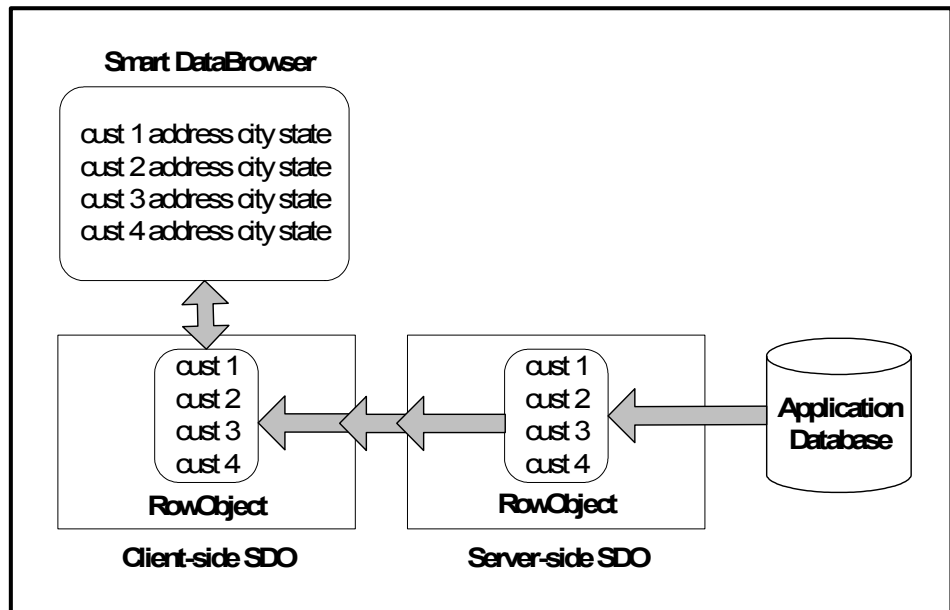


Figure 10–1: Reading the first batch of rows into the SDO

Once the RowObject table has been passed to the client, the server-side copy is emptied in preparation for reading another batch of rows from the database. When the SDO is running in a stateless AppServer session, the norm for deployed Progress Dynamics applications, the server-side SDO is deleted to free the session for another user. If another batch of data is requested, the server-side SDO restarts, the query reopens, and repositions. The data is read from the database into the server-side RowObject table, passed to the client, and appended to the client-side RowObject table. The client-side query on the RowObject table is reopened and repositioned to the new current row (the first row of the new batch). If there is a SmartDataBrowser, the reopening of the RowObject query automatically refreshes the browser.

Remember that all records are read from the database NO-LOCK. This is because there is simply no way to keep record locks on a set of records that is being passed to another Progress session. The SDO code checks to make sure that another user has not modified any updated row, since it was read. Also, there are functions to refresh the data set or the current row if that is desired.

Figure 10–2 shows this process.

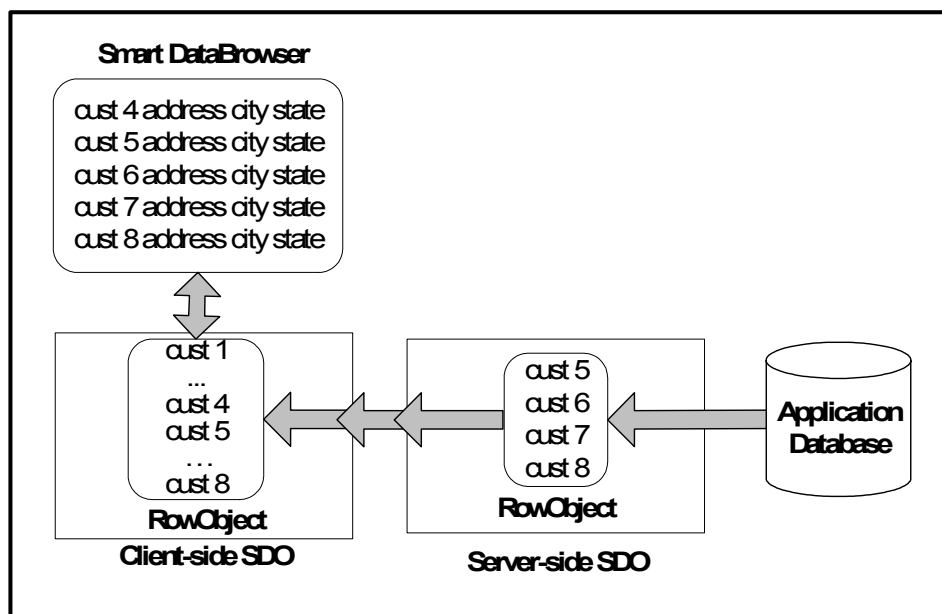


Figure 10–2: Reading the second batch of rows into the SDO

To collect changes to be sent back to the server, there is a separate copy of the RowObject temp-table called RowObjUpd. When the user updates an existing row two things happen. First, a row is created in the RowObjUpd table and the original values of the modified row are copied there. This before image of the row allows the SDO to determine later in the process whether another user has changed the row since this user read it. Second, the changes are saved to the RowObject table on the client, and a special flag field called RowMod is set to “U” (for Update) to indicate that the row has changed.

The RowMod field in the “before” image record in the RowObjUpd table is set to blank, as shown in [Figure 10-3](#).

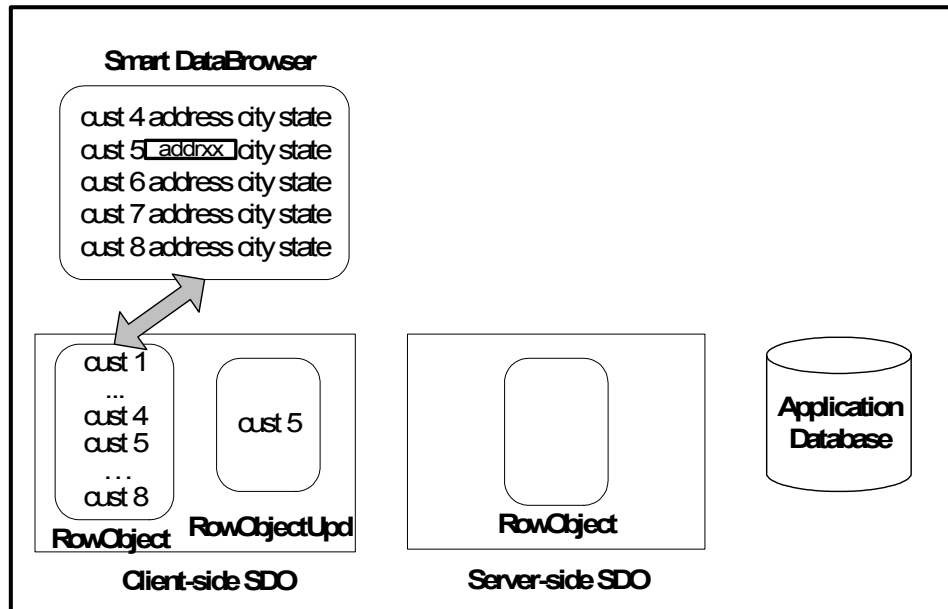


Figure 10-3: Modifying customer 5 In the client SDO

If the user adds (or copies) a new row, the data is written to the RowObject table, with a RowMod code of “A” (or “C”). It might seem puzzling that the changes are first written to the RowObject table on the client rather than to the RowObjUpd table. The reason for this is that any changes need to be visible to the user before they are committed. Thus, they must be made to the table the client objects are browsing (the RowObject table). As discussed below, all the changes are moved into the RowObjUpd table on Commit.

When a user deletes a row, it is also necessary to reflect this in the RowObject table, so that the row appears to be truly gone from the user’s session. For this reason, when a Delete occurs, the row to be deleted is moved to the RowObjUpd table with a RowMod of “D”, and it is deleted from the RowObject table.

The diagram shown in [Figure 10–4](#) reflects an Add and a Delete done within the same transaction as the first Update.

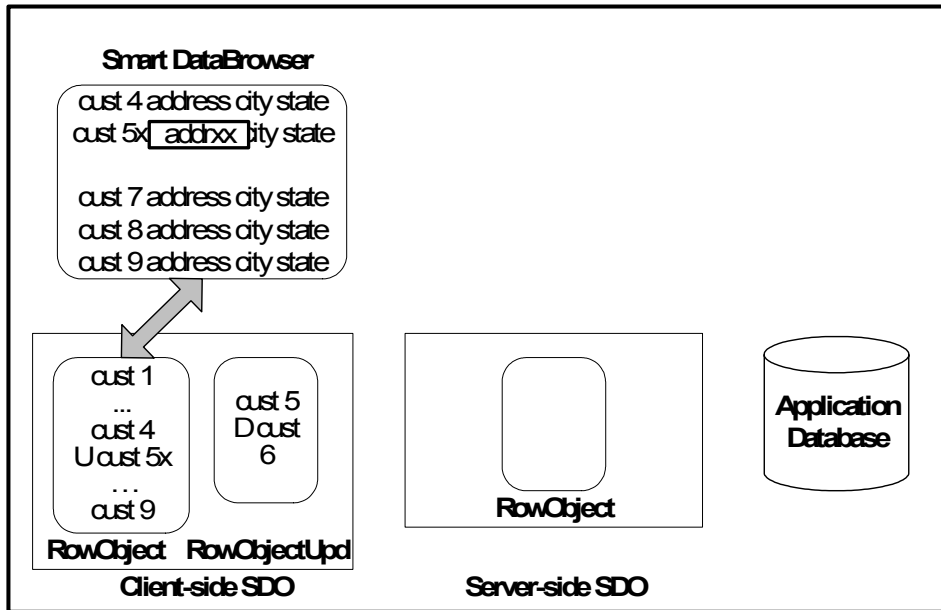


Figure 10–4: Adding row 9 and deleting row 6 after updating row 5

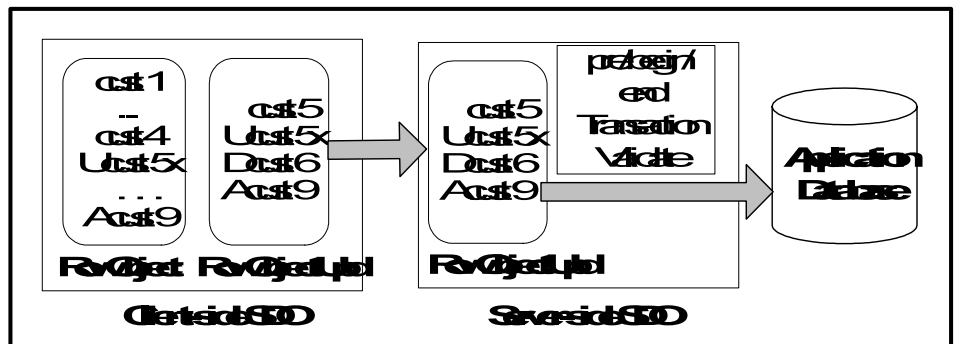
By default, each individual change (a Save of an Update, or Save of an Add/Copy, or a Delete) goes back to the server-side SDO to be committed to the database. If there is a Commit Panel, however, the AutoCommit SDO property that controls this causes changes to be accumulated in the client-side SDO until Commit is done. Then they all go back to the server together.

When a Commit occurs (either immediately when the user chooses **Save** if there is no Commit Panel or Commit band in the Toolbar, otherwise when the user chooses), the following happens on the client:

1. Updated rows are copied from the RowObject table to the RowObjUpd table; thus there is a before image of each modified row (with a RowMod of “”) and an after image (with a RowMod of “U”).
2. Added or Copied rows are copied from the RowObject table to the RowObjUpd table.
3. The RowObjUpd table is passed to the server-side SDO as an argument to the serverCommit procedure.

The following then happens on the server-side SDO:

1. Any `preTransactionValidate` procedure is run to handle business logic that should occur before the actual database transaction begins.
2. The database transaction is started.
3. Any `beginTransactionValidate` procedure is run to handle business logic that should occur inside the transaction but before the changes themselves are written to the database.
4. The Updated row is written back to the database. Each updated row has a `RowIdent` field in the `RowObjUpd` table that holds the `ROWID`(s) of the database records the row was derived from. The database records are read `EXCLUSIVE-LOCK` and compared with the “before” record in the table. If another user has changed the database record, the current change is rejected, unless the `CheckCurrentChanged` SDO Instance Property has been set to `NO`. If nothing has failed so far, those fields that were modified are `ASSIGNED` back to the corresponding database records.
5. The Added row is Created in the database and its values Assigned.
6. The row marked to be Deleted is read from the database and deleted.
7. Any `endTransactionValidate` procedure in the SDO is executed to handle business logic that should occur inside the database transaction but after all the modifications are made to the database, as follows:



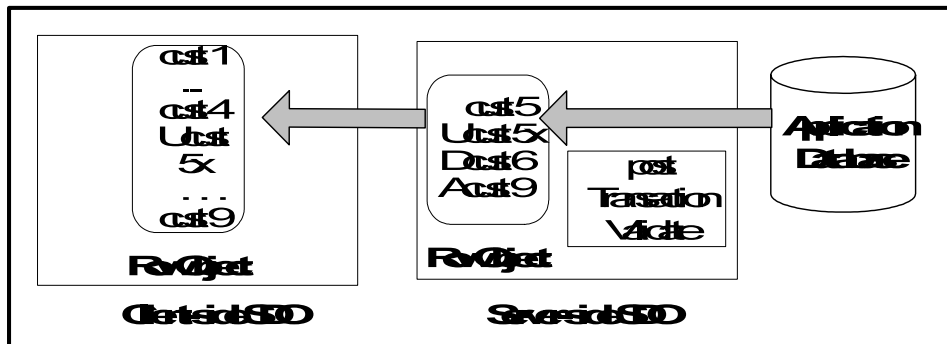
8. The Updated and Added rows are re-read from the database, to capture any changes made by database triggers or other code (such as the assigning of a key field value by a `CREATE` trigger, or the calculation of a field by a `WRITE` trigger, or by one of the `TransactionValidate` procedures).
9. Any `postTransactionValidate` procedure is executed on the server.

10. The final versions of the possibly modified Updated and Added rows are passed back to the client, and the final versions are copied back into the RowObject table to be displayed in the client.
11. The server-side RowObjUpd table is emptied.

NOTE: When AutoCommit is enabled and changes in one SDO trigger a Commit action, all unsaved changes in all the SBO's SDO's are saved.

The following happens on the client:

1. The RowMod flags in the client-side RowObject table are cleared.
2. The client-side RowObjUpd table is emptied:



10.2.4 Determining the proper batch size

The number of rows read at a time is a configurable Instance Property of the SDO called RowsToBatch. This notion of batching rows is important because it can be very time-consuming to read all of the records that satisfy the SDO's query definition into the RowObject table at once, before any of them are viewed by the client SmartObjects.

Here are some things to keep in mind when setting the RowsToBatch property (whose initial value, somewhat arbitrarily, is 50):

It is highly advisable to limit the number of rows that satisfy the database query to the smallest number possible. Although users might think that they want to browser through large numbers of database rows, there is almost never a need to do this. Use of the filter button on an Object Controller-style browse window, or using the SmartFilter object, can direct the user more efficiently to the exact record or subset of records really needed.

NOTE: The SmartFilter object is described further in the Progress Version 9 documentation.

Remember that you should usually define SDOs with the most general possible WHERE clause (typically no WHERE clause at all beyond what might be needed to describe a join, if there is one). You can then use this one SDO in many situations where the data set is restricted by a WHERE clause defined at run time to present to the user only those rows that really need to be seen. If the result set of the query is small, it is usually best to set the batch size large enough so that all the rows can be retrieved at once. (To put this another way, it is probably a good idea to restrict the result set size wherever possible so that the 50 row default is more than adequate to retrieve all of it at once.)

Also keep in mind that if a user applies a filter to a data set using the Progress Dynamics filter button, that filter can be saved permanently in the Repository, so that the query will be filtered the same way the next time the user starts that part of the application. Encouraging the user to choose the **Filter** button can help cut down on unnecessary traffic between client and server.

There are several advantages to getting the entire (possibly filtered) result set at once. First, if there is a Browser browsing the RowObject query, it will have certain unavoidable quirks if the result set is retrieved in multiple batches. If the user scrolls to the end of the first batch by clicking and holding the down arrow on the vertical scrollbar, he must release the mouse from the scrollbar before the OFF-END trigger can fire. This trigger retrieves another batch of rows from the server, adds them to the client-side temp-table query, and reopens and repositions the query to allow the user to continue scrolling. Also, the thumb size of the scrollbar will not reliably reflect the total number of rows in the result set when it not retrieved all at once.

Another factor is that it is easy and frequently useful to re-sort or further filter the RowObject query after it has been retrieved. This clearly makes sense only after the entire result set has been retrieved. Sorting a Progress Dynamics Browser by clicking on a column header, for example, will only sort the records already retrieved from the database. This will not be very useful if that is not the whole data set.

However, there are many cases where it is necessary to allow a large data set to be browsed (or if your users simply insist on doing this). RowsToBatch lets you determine how many rows at a time to get. The larger the batch size, the longer it takes for the application screen to come up; the smaller the batch size, the more frequent the interruptions when the user scrolls from one batch to another. You must balance these two considerations. One compromise that can be effective in many cases is to set the initial batch size to be very small (for example, just enough rows to fill the view port, if there is a browser). Then programmatically reset it to a much larger number if the user wants to see other rows. This minimizes waiting time before the application window appears. The appropriate place to do this is in a local `initializeObject` procedure either in the logic procedure for the SDO itself (if this behavior should occur wherever the SDO is used) or in a local `initializeObject` procedure in a custom super procedure for the dynamic window where the SDO is used.

In the following example, the custom code explicitly sets `RowsToBatch` to a small number before the standard initialization code is executed. (It overrides any other setting of the property, either to its default value or some other value). The code then resets the value to a larger number after the initial batch has been retrieved. This code would go into the logic procedure (which is the custom super procedure) for the SDO. Alternatively, your code could simply allow the Instance Property setting to be used initially, and then reset the property value after the `RUN SUPER` statement:

```
/*-----
Purpose:  Local Override of initializeObject in an SDO,
         to reset the RowsToBatch property following the
         initialization of the SDO, so that
         up to 2000 additional rows will be retrieved if the
         user scrolls out of the initial batch.
Parameters: none
-----*/
/* Code placed here will execute BEFORE standard behavior. */
DYNAMIC-FUNCTION('setRowsToBatch':U IN TARGET-PROCEDURE,
  INPUT 20).
RUN SUPER.
/* Code placed here will execute AFTER standard behavior. */
DYNAMIC-FUNCTION('setRowsToBatch':U IN TARGET-PROCEDURE,
  INPUT 2000).

END PROCEDURE
```

Another property to consider where larger result sets are concerned is `RebuildOnRepos`. This logical SmartDataObject property is false by default. If the user chooses the **Last** button on a Navigation Panel or otherwise repositions to some row not currently in the client-side RowObject table, the SDO will retrieve one batch of rows after another until the desired row is reached. This can take a very long time if the result set is large. If this is the case you should check on the **Rebuild On Reposition** toggle box in the design time Instance Property dialog box for the SDO to reset it to true.

In this case, when the user repositions to a row outside the current client-side data set, the client-side temp-table will be thrown away, the database query will be repositioned directly to the desired row, and the RowObject temp-table will be rebuilt backwards or forwards from that new starting point. This makes doing a Last much faster, but it can force the SDO to retrieve the same rows multiple times if the user alternates back and forth between the First and Last rows, for example. A good rule of thumb here is to set `RebuildOnRepos` to true if the number of rows in the result set is likely to be much larger than the batch size. Otherwise leave it false, so that the overhead of getting all the rows into the client temp-table happens only once.

10.3 Strategies for SmartDataObject query definition

A major purpose of SDOs is to make it easier for you to put all the appropriate business logic for dealing with a database table or set of related database tables in one place, so that it is always executed whenever that table is used. For this reason, it is important not to define more SDOs than you need. Keep in mind that you can attach many different visual objects to an SDO under different circumstances. Different fields can be displayed or enabled, and other aspects of the SDO can be customized. There is generally no need to have a number of different SDOs for these different situations. Though no set of guidelines will cover all application situations, the following suggestions will give you something to think about as you design your application.

Consider these two basic principles:

1. Ideally there should be just one SDO for a database table.
2. That SDO should contain all the fields that any client object would ever use from that table.

If there is just one SDO for each database table, then there is no question about where the update logic for the table goes, and it will be executed whenever that SDO is used.

In most cases there should be only one table in each SDO, as well.

Joins are problematic. You should never use an SDO to update multiple tables in a one-to-many relationship, such as an Order and its OrderLines. A one-to-many join will not provide a useful display, since the parent information will be repeated for every child record. It is also not practical to update the parent information in this kind of setup, again because it is repeated for each child record. For example, if you define an SDO with the following query, the Order information will be repeated for every OrderLine an Order has:

```
FOR EACH Order, EACH OrderLine OF Order
```

This result will not look attractive, and it makes the Order information more or less impossible to update. If you allow changes to Order fields, then a modification to an Order record through the SDO will appear to have changed just one of many instances of that Order information. However, there is only one Order record in the database, and that one record has been changed.

So updateable joins should be restricted to one-to-one joins, between tables that are almost always referenced together. [Chapter 11, “Building Advanced Business Logic in a Progress Dynamics Application,”](#) discusses alternatives for logic that goes beyond a single table.

Another problem with joins is that if both tables in a joined SDO are updateable and an Add operation is done, a new row is created for both tables even when this might not be desired.

So, if an SDO has a join, you should enable only one table in the join so that the SDO can have the update logic for that one table only. The one reasonable exception to this would be if the two tables are really always updated (and created) together, such as header and detail information for a Customer.

You might immediately (and reasonably) object that these guidelines are too restrictive. For one thing, a join is often necessary because the main table in a query contains key values for other tables. These key values are not meaningful for users looking at the data. For example, an Order SDO that just shows CustNum will not be helpful to the user without the Customer Name. This is not a problem: you can go ahead and join Order to Customer and include the Name field from Customer in the Order SDO. However, **do not** enable that field for update. That way the Customer Name field can be displayed without you having to worry about Customer update logic in that object. An object can contain read-only joins in several directions if necessary to display meaningful information in this way. For example, the Order SDO could also include SalesRep OF Order in addition to Customer OF Order to show (but not update) the SalesRep RepName field.

As discussed in [Chapter 7, “Building Progress Dynamics Lookups and Combos,”](#) you can add Progress SmartDataFields in the form of dynamic Lookups and Combos to any Viewer in your application. You can use them not only to display all the possible valid values for a field, but also to display other fields that help identify the value in a meaningful way. For example, you can use them to show the Customer Name for an Order as well as the CustNum field, or the Sales Rep name field in addition to the Sales Rep initials. It is important to understand in this regard how Progress Dynamics supports these special fields, and also what the product direction is.

In Progress Dynamics, a dynamic Lookup or Combo is populated from the database when it is displayed. This means that when you display a record in a Viewer that includes a dynamic Lookup or Combo, the values from the related table that are displayed along with the foreign key value are retrieved from the database (therefore from the server in a distributed application) at that time.

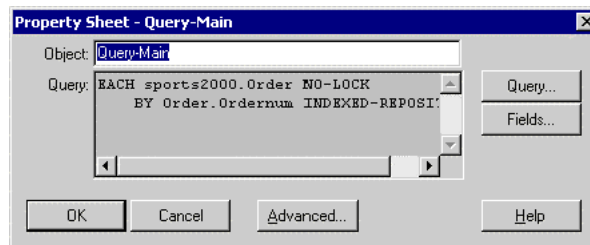
For example, you can avoid a server access to obtain the Customer Name for display in an Order Viewer if that field has already been added to the Order SDO through a read-only join to the Customer table. For this reason, you should include such fields in your SDOs, so that they will be available as the product is optimized, without changes to existing applications being required.

In addition, you might want to show linked fields such as Customer Name or SalesRep Name in a Browser, where the Lookups and Combos are not available. This is another reason to include them in the SDO definition.

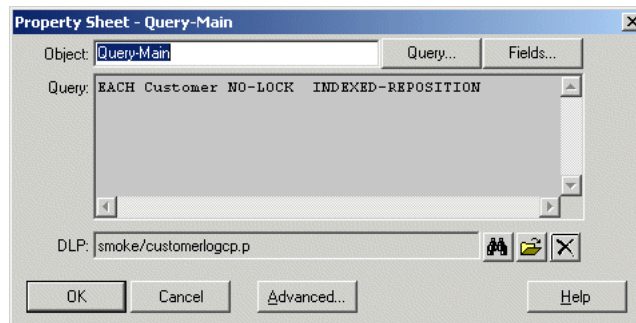
The time to modify SDO definitions in this way is after you have generated SDOs in the Object Generator. As was described in [Chapter 5, “Using the Object Generator,”](#) you can specify that you only want to generate SDOs, not dynamic Viewers and Browsers, when you run the Object Generator. This gives you the ability to modify the query definitions and field lists before generating other objects. The Object Generator will add joins to other tables automatically if your schema definition conforms to the naming conventions described in [Chapter 2, “Database Design Principles in Progress Dynamics.”](#) For an existing database, you must define the relationships and the appropriate descriptive fields to add to SDOs yourself.

The following steps describe how to modify an Order SDO example, to show its use and also its limitations, and then show you an alternative:

- 1 ♦ After you have generated SDOs in the Object Generator, open them in the AppBuilder to edit them.
- 2 ♦ After opening an SDO, double-click on its design window to bring up its property sheet:



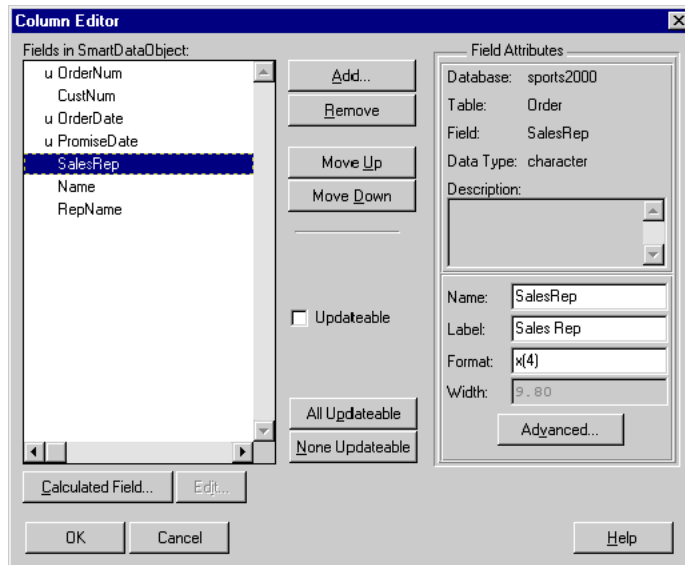
- 3 ♦ Choose the **Query** button to display the Query Builder for the Order SmartDataObject.
- 4 ♦ Select the **Order**, **Customer**, and **SalesRep** tables:



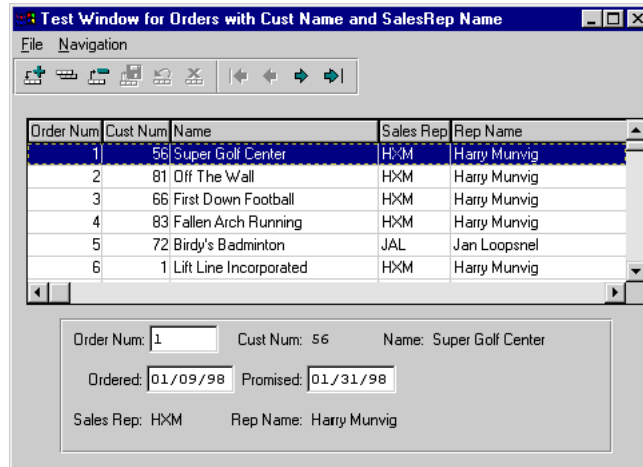
NOTE: Because SDOs nearly always run in stateless environments like Progress Dynamics, always use the INDEX-REPOSITIONED keyword.

- 5 ♦ Choose **OK**.

- 6 ♦ Choose the **Fields** button to bring up the Column Editor for the SDO. Note that properties of the fields reflect the data field definitions (Entity Maintenance) rather than the database schema.
- 7 ♦ Add the **Customer.Name** and **SalesRep.RepName** fields to the SDO's column list, but make them non-updateable. Note that the SalesRep and CustNum fields are also marked non-updateable. This is to prevent a user from modifying these key fields. However, if you allow these foreign key fields to be modified, the SDO will properly refresh itself to join to the new related record. Whether the key fields are updateable or not is up to you as the application designer:



When you build a Viewer for this SDO and then add it to a window along with a dynamic Browser and Toolbar, you can see the Customer and SalesRep names along with their key values. Because they are not updateable, the SDO remains clearly an order SDO for update purposes, and the customer and salesrep information can never be modified using it:



Another objection to having just a single SDO for each table could be that retrieving the entire SDO data set (and especially the entire main table plus various fields from other joined tables) is inefficient when the user wants to browser just a few fields to select a particular record. In this case it is perfectly reasonable to define an additional read-only SDO for the table, with no joins to other tables, and with the field list restricted to just the fields needed for the browser or other uses to which this Object is put. You can then use this object in a browse window or other situation where an efficient browser is needed.

Note, however, that the dynamic Lookup can often satisfy the requirement for browsing records without the use of an SDO at all.

Another point mentioned earlier with regard to the field list in an SDO is that it should generally include all the fields from the primary table **that are used on the client**. Sometimes tables contain a very large number of fields. This can lead to problems defining SDOs for them because of limits to the statement length in Progress, when the DEFINE TEMP-TABLE statement for the SDO becomes excessively long. In some cases a table can have fields that are used by the business logic (in calculations, for example) but never viewed or updated on the client. Such fields do not normally need to be part of the SDO definition. The SDO will find the database record for an SDO temp-table record being updated when the update is returned to the server. You can modify or look at any other fields in the database as part of the SDO's update logic.

In other cases where the field list is exceptionally long, it might be the case that different parts of the record are used in one kind of operation and other parts of the record in another.

NOTE: If this is the case, then it is very likely that splitting up the table into multiple tables would improve the database design and the efficiency of the application, but it might not be practical to do this with an existing database.

If this is true, then you can define multiple SDOs to view and update different parts of the record so that the whole record is not transferred across the AppServer connection. With the logic procedure that is part of the SDO in the Progress Dynamics architecture (see the “[Standard validation procedures for SDOs](#)” section for more details), you can define a single logic procedure to be the custom super procedure for multiple SDOs, if that is appropriate, to share the business logic between SDOs.

These guidelines will not satisfy every need, but you should keep them in mind when you are designing SmartDataObjects for your application. Perhaps the best general rule is that the smallest number of SDOs that will do the job is the best number. Extra SDOs for specialized purposes are likely to cause headaches later on, when you are trying to sort out why the application's behavior is not consistent.

10.4 Standard validation procedures for SDOs

SmartDataObjects support a set of standard validation hooks where you can place or call your business logic. This section discusses the standard hooks for SDOs, including extensions to the set of validation procedures available for the first time with Progress Dynamics.

10.4.1 Client-side data validation

This section discusses the standard validation procedures to run on the client, without a database connection.

Column validation

For each field in the RowObject record of an SDO, you can optionally provide a validation procedure. Such procedures are named `<column>Validate`, where `<column>` is the name of the field in the RowObject record.

The action of a `<column>Validate` procedure should be to determine the validity of its input parameter (the same data type as the corresponding column in RowObject). On successful validation the procedure should RETURN. On unsuccessful validation the procedure should RETURN *stringExpression*. The string expression is added to a list of error messages for later presentation to the user.

Under no circumstances should a <column>Validate procedure RETURN ERROR. These procedures are run NO-ERROR for every column with a modified value, and the setting of the error condition is presumed to mean that no corresponding validation is defined. If your <column>Validate procedure returns an ERROR then its validation is ignored.

Here is a simple example of a column validation procedure for the Customer Name field:

```
PROCEDURE nameValidate:
  DEFINE INPUT PARAMETER pcValue AS CHARACTER.
  IF LENGTH(pcValue) < 2
  THEN RETURN "Name must be longer than 2 characters.".
END PROCEDURE.
```

Although the RETURN value can be a simple string as in this example, you should generally define messages for every type of message that can occur in the application so that these messages can be stored in the Repository and translated into other languages. The Message utilities are discussed in the [“Message handling in Progress Dynamics”](#) section.

You can control on a column-by-column basis whether or not the validation is to occur on the client. If the <column>Validate procedure contains no database references (and this is indeed usually the case), then you can unselect the **DB-REQUIRED** flag in the AppBuilder code section editor. Doing so causes that procedure to be compiled into the SDO client proxy sdoname_c1.w (or its associated logic procedure proxy) and executed on the client.

Checking **DB-REQUIRED** on prevents the procedure from becoming part of the client proxy. Generally, any validation that requires the database connection should be placed into one of the pre-Transaction procedures discussed in the [“Server-side validation”](#) section. The advantage of using client-side logic at all is that validation not requiring a database access can be done without an AppServer call. In the event of an error, Progress Dynamics can return a message to the user without ever sending the update to the server.

All <column>Validate procedures, whether DB-REQUIRED is true or false, are compiled into the server-side portion of the SDO (<sdoname>.w or its associated logic procedure).

The fact that <column>Validate procedures are defined does not mean that these will be executed on the server, even though they are compiled into the server-side portion of the SDO. The server functions only perform <column>Validate (and rowObjectValidate—see later) if the attribute ServerSubmitValidation is set to **TRUE**.

In Progress Dynamics <column>Validate and rowObjectValidate procedures (discussed below) are **always** executed on the server, unless one of the validation checks has already failed on the client and returned a message to the user. Note that although this might at first appear to be an extra overhead, it is fundamental to ensure that all validation fires correctly for non-Progress (for example, Java) clients.

By default, `<column>Validate` procedures have access to only one piece of data—the value of the column in question. Although it is possible to gain access to other data through the direct access of the `RowObject` buffer, this is not recommended.

You should use `<column>Validate` when a column can be validated in isolation from other columns in the SDO. If the validation can be performed without reference to database objects, then this is an ideal candidate for column Validation with **DB-REQUIRED** unchecked. (For example, verifying that a mandatory field has a value or that simple range criteria are observed.) Such validation will execute on the client without involving any `AppServer` requests should the validation fail. In particular, it can be useful to create such a procedure to validate an individual column when you anticipate that other client-side objects will want to execute that validation separately from validating the record as a whole. For example, if it is essential that validation be done on `LEAVE` of the field in a `SmartDataViewer`, without waiting until the rest of the update is done. Then the `Viewer` code can easily invoke the right `<column>Validate` procedure in its `Data-Source` without executing anything else.

Do not use `<column>Validate` if the validation must be performed on every record write. These procedures are only executed if the value of that particular column has been changed during the course of updating the record, or if the value has been changed from its default initial value in the case of an `Add` or `Copy`.

Additionally, avoid `<column>Validate` if the validation logic must consider the current settings of other fields—`rowObjectValidate` would be a better choice in this case.

Finally, be aware that `<column>Validate` executes **prior** to the commencement of a transaction. And always remember that the `<column>Validate` procedure will be compiled into the client-side object only if it involves no database access and **DB-Required** has been unchecked. Use `preTransactionValidate` and other server-side procedures to do other checks that occur before the transaction begins but that need access to the database.

rowObjectValidate

The `rowObjectValidate` procedure is intended for validation that must consider more than one field in the `RowObject` record.

If the `rowObjectValidate` procedure is defined and has **DB-REQUIRED** checked off, then it fires on the client after any client-side `<column>Validate` procedures have completed. Note that `rowObjectValidate` executes even if one or more of the `<column>Validate` procedures have failed and returned a validation message. Thus all client-side validation is completed before the full list of errors is returned to the user.

The following code is an example:

```
PROCEDURE rowObjectValidate:
  IF RowObject.Balance > RowObject.CreditLimit THEN
    RETURN "Balance may not exceed credit limit of " +
      STRING(RowObject.CreditLimit).
  END PROCEDURE.
```

All of the validation procedures should return an error message to signal error. Progress Dynamics supports a mechanism for returning a list of information about the error, including the table and field name, as well as a Group and Error Code (error number) for each error, using the include file `aferro.txt.i`. This include file and other message-handling files are described in the [“Message handling in Progress Dynamics”](#) section.

If `rowObjectValidate` is defined with **DB-REQUIRED** unchecked, then it will execute on the Client. If **DB-REQUIRED** is checked on, then it will **not** execute on the Client.

As with the validation procedures for individual columns, you should avoid **DB-REQUIRED** code for `rowObjectValidate`. The intention here is that as much of your validation should execute client-side as possible to reduce AppServer traffic. Move validation that might cause `rowObjectValidate` to require a database connection to `preTransactionValidate` so that it executes on the server-side.

Use `rowObjectValidate` whenever your validation logic can be performed without any database access, and prior to the start of a database transaction, but requires access to more than one column value in the SDO.

10.4.2 Server-side validation

Additional validation procedure hooks are designed to be used on the server, where there is free access to the database. This section discusses these entry points. For each of these procedures, remember that the client-side validation entry points described above apply only to a single SDO record. If multiple records are updated, then the client-side validation procedures will be called a number of times. By contrast, the first set of server-side, transaction-level validation routines described below could potentially apply to multiple record updates, and you should write them as **FOR EACH** blocks on the `RowObjUpd` table. As a result, when using the procedures defined below, you must write not only a **FOR EACH** loop on the `RowObjUpd` table, but also (if the logic applies only to specific operations such as Add, Copy, Update, or Delete) a check on each record within the loop that examines the value of the temp-table field `RowObjUpd.RowMod`. The field can have a value of ‘A’ for Add, ‘C’ for Copy, ‘D’ for Delete, ‘U’ for an Updated row, and “(blank) for the before image of an updated row (the field values as read from the database).

In the case of an update, another standard RowObjUpd field, RowNum, holds a unique row number for each update. The updated row with RowMod = 'U' and the before image of the same row with RowMod = "" will have the same RowNum field. Thus a block of code that needs to look at both before and after values will generally be of the form:

```
DEFINE BUFFER OldRowObj FOR RowObjUpd.  
FOR EACH RowObjUpd WHERE RowMod = 'U':  
    FIND OldRowObj WHERE RowObjUpd.RowNum = OldRowObj.RowNum AND  
        OldRowObj.RowMod = ''.  
    /* Now you can compare values in OldRowObj and RowObjUpd. */  
END.
```

Because of the nature of this SDO-specific logic that you must add to the validation, an alternative set of procedure hooks has been defined for Progress Dynamics to handle updates a row at a time, and with buffer names based on the table names. These alternative hooks are discussed in the [“New Progress Dynamics validation procedures”](#) section.

preTransactionValidate

preTransactionValidate always executes on the server, immediately prior to the commencement of a transaction. As such, it can freely read the database, but should do so NO-LOCK. To maximize performance, preTransactionValidate should be restricted to the type of Referential Integrity check that can be implemented as a simple CAN-FIND statement. For example:

```
PROCEDURE preTransactionValidate:  
    FOR EACH RowObjUpd:  
        IF NOT CAN-FIND(...) THEN RETURN "...".  
    END PROCEDURE.
```

Within this loop, use the RowObjUpd.RowMod field to determine the type of change for each row ('A'dd, 'C'opy, 'U'pdate, or 'D'elele).

Within preTransactionValidate you can physically alter the values that will be written back to the database, if so desired. That is, any changes made to the records in the temp-table will be seen by later procedures within the transaction. Generally, preTransactionValidate is appropriate for code that makes changes to the updated records, and for code that needs to do comparisons with existing records in the database, but without making changes to them.

Keep in mind that you do not want to place code here that might need to look at updated and modified records in one table. That is, some records are in the RowObjUpd table as updates and some are unmodified records in the database. For example, if you need to total the OrderLines of an Order in a validation procedure for the OrderLine table, you do not want to have to total records in the update temp-table and records in the database together. This will complicate your logic unnecessarily. Such logic should be left to the end of the transaction so that all the records are in the database together.

Also, you also do not want to put logic here that does any other database updates. The transaction block is not yet open, and those updates would not be undone if the transaction fails. For example, if the validation logic in an OrderLine SDO needs to total all the OrderLines for an Order and adjust the Balance of the associated Customer accordingly, and compare that against the CreditLimit, this validation should **not** be done in preTransactionValidate.

beginTransactionValidate

The beginTransactionValidate procedure executes immediately after the start of the transaction, but before any database updates have been performed. There is little to do here except perhaps to gain an EXCLUSIVE lock on a parent table to prevent other user sessions from updating the same records (for example, in an OrderLine SDO, lock the Order table). In addition, if it is important to do some other validation before anything is actually written to the database, you should do that here.

For example, if you want to assure that no other user will attempt to make changes to an order (an Order record and its OrderLine records) while you are updating any part of the order, you can read the Order record with an EXCLUSIVE-LOCK at the beginning of the transaction in the OrderLine object. Then allow the transaction to make changes to one or more OrderLines.

endTransactionValidate

After all updates have been written to the database, but before the transaction is committed, endTransactionValidate is called. This is the ideal place to perform database cascade operations.

One example: Now that all OrderLines have been updated, recalculate the total amount owing. If done in the write trigger, then this behavior would execute once for every order line. By placing it in endTransactionValidate the logic is performed once only, whether one, some, or all of the order lines were updated.

Another example: On delete of a specific entity you might want to **conditionally** enforce cascade deletion of child records. It is not possible to put conditional cascade deletion into Delete triggers.

postTransactionValidate

The `postTransactionValidate` procedure executes once the transaction has been committed, and only if the commit was itself successful. This could be a useful place to write log records if required. Log records are often written from Write triggers, and require the triggers to execute within the context of a specific user. If the log were to be written to a database, then `postTransactionValidate` would have to open a transaction of its own and manage that carefully.

10.4.3 New Progress Dynamics validation procedures

Progress Dynamics also supports additional standard internal procedure hooks that let you write validation logic for specific operations (Create, Write, and Delete), one record at a time. This spares you from writing a FOR EACH loop in each validation procedure to allow for the possibility that more than one row is being updated in a single transaction. It also removes the need to check the value of RowMod or in any other way write code specific to the structure of the RowObjUpd table. Among other things, this structure makes it possible to use the same logic procedure to validate a buffer outside the context of an SDO.

For each stage in the transaction as described above (`preTransaction`, `beginTransaction`, `endTransaction`, and `postTransaction`), three procedure names are defined which, if they exist, will be called to deal with specific operations a row at a time. This then gives you a total of twelve additional hooks that you can use as needed:

- `createPreTransValidate`
- `createBeginTransValidate`
- `createEndTransValidate`
- `createPostTransValidate`
- `writePreTransValidate`
- `writeBeginTransValidate`
- `writeEndTransValidate`
- `writePostTransValidate`
- `deletePreTransValidate`
- `deleteBeginTransValidate`

- deleteEndTransValidate
- deletePostTransValidate

NOTE: The names are shortened by changing Transaction to Trans.

These hooks are called exactly when you would expect, based on their names. The Create group are called instead of their pre/begin/end/post counterparts for each newly created row, that is, a row that results from either an Add or Copy operation.

The hooks in the write group are called for each Update, Add, or Copy operation, the delete group for each Delete operation.

To make the logic as general as possible, and to support calling this validation logic from outside of SDOs, the buffer name of the table being updated is not RowObjUpd but rather the table name preceded by b_. This buffer name is available to the internal procedure. You do not need to define it there or pass it in as a parameter. (These internal procedures take no parameters at all.) In the case of the Write group of procedures, the unchanged version of the record as it was originally read from the database is also available in a buffer with the table name prefixed by old_.

In addition, because the Write procedures will be called for Adds and Copies as well as Updates (since you will want to place business logic there to validate values being written to the database for all of these cases), there is an additional logical variable named isCreate. The isCreate variable is available within the validation procedures. It will be TRUE if the operation is an Add or Copy, and FALSE otherwise.

Note that the client-side validation procedure rowObjectValidate has not been subdivided into multiple other procedure hooks. This is partly because it is called for each individual Save operation, and therefore never needs to refer to multiple records as the server-side procedures must be prepared to do. Also, the principal goal of the logic procedure is to make the server-side logic, where most code is expected to go, as flexible and reusable as possible. However, the b_ + <tablename> buffer is defined for rowObjectValidate and available to it, so that the developer does not have to write references to the RowObject buffer.

The framework gives you the option of using one or the other of these two sets of procedures in a single SDO, but not both. That is, an SDO (and its supporting logic procedure if it has one) can use the pre/begin/end/postTransactionValidate procedures as in the standard Version 9 ADM. Alternatively, it can use the new create/write/delete set of procedure names. It cannot use both (this is largely to avoid confusing issues of the exact order in which all the procedures would be executed).

In general, the recommendation is that new application objects should use the new procedures. They are more fine-grained. That is, they give you logic for a more specific update type and free you from the RowObjUpd and RowMod references of the other procedures. Also, they are more reusable from elsewhere in your application. The TransactionValidate procedures provide compatibility with existing Version 9 applications.

[Chapter 5, “Using the Object Generator,”](#) shows some examples of the code for these new procedures, as automatically generated for you by the Object Generator. Here are a couple of other examples:

```
PROCEDURE deletePreTransValidate:
/*-----
Purpose:  Procedure used to validate records server-side before the
          transaction scope on delete of Order
Parameters: <none>
Notes:    Restrict delete of Order with OrderLines.
-----*/

IF CAN-FIND(FIRST OrderLine
            WHERE OrderLine.OrderNum = b_Order.OrderNum) THEN
DO:
  ERROR-STATUS:ERROR = NO.
  RETURN {aferrortxt.i 'OE' '8' 'Order' '' STRING(b_Order.OrderNum)}.
END.
END PROCEDURE.
```

This code checks to see if there is an existing OrderLine for the Order being deleted. If there is, the delete is rejected. The use of the standard error formatting include file `aferrortxt.i` is explained in more detail in the [“Message handling in Progress Dynamics”](#) section. In this case, error message number 8 from the Order Entry (OE) group is returned with a substitution argument indicating the Order Number. So the message as stored in the message table in the Repository is something like “Can’t delete Order Number <&1>, which has Order Lines.”

The standard Progress ERROR-STATUS is reset as a matter of course, because the error handling code will rely on the message being returned as the error flag.

Note the use of the database table name OrderLine to refer to the database lookup, and the SDO buffer name b_Order to refer to the record being deleted.

This next example shows the use of the Old buffer holding the original record as read from the database:

```

PROCEDURE writePreTransValidate:
/*-----
Purpose:   Procedure used to validate records server-side before the
           transaction scope on write
Parameters: <none>
Notes:
-----*/

DEFINE VARIABLE cMessageList AS CHARACTER NO-UNDO.
DEFINE VARIABLE cValueList AS CHARACTER NO-UNDO.

IF b_Customer.CreditLimit - Old_Customer.CreditLimit > 10000 THEN
DO:
  ASSIGN
    cValueList = STRING(b_Customer.CreditLimit)
    cMessageList = cMessageList +
  (IF NUM-ENTRIES(cMessageList,CHR(3)) > 0 THEN CHR(3) ELSE ':U) +
    {aFerrortxt.i 'CM' '8' 'Customer' 'CreditLimit' cValueList }.
END.

ERROR-STATUS:ERROR = NO.
RETURN cMessageList.

END PROCEDURE.

```

In this example, the CreditLimit of the original version of the record as read from the database is compared with the modified value. If this has been increased by more than \$10,000, the change is rejected, using message 8 in the Customer Maintenance (CM) message group.

Note the accumulation of messages here. More than one message can be returned to the client, delimited by CHR(3). In this case, the ellipsis represents additional error checks that might be done. If the intent is to check for all possible errors and return a list of them, rather than returning the first error encountered, then this coding style accomplishes that. Each error text is added to a variable, and the entire list is returned to the client where it will be properly parsed and presented to the user.

NOTE: If you want to implement initialization functionality for data-handling objects (SDOs, SBOs), you can override the createObjects() method of the container. Override with a check for Valid-Handle and access contained instances. This technique allows you to add functionality before the main initialization.

It is advisable to check all forms of validity in application code, even if there exists either a database trigger procedure or, as in this case, a built-in database check to perform the validation for you. If your code catches an error such as this, you can return a meaningful (and translatable) message to the user rather than waiting for Progress to attempt to display what might be a much less meaningful message, and one that might not make it back to the client successfully. And your code can react to the error exactly as you want, rather than trapping default Progress behavior.

10.5 The SDO logic procedure

As described in the standard ADM2 product documentation, table maintenance logic should go into the SDO defined for each table. In this way it will be executed whenever the user makes an update through the SDO, regardless of the nature of the interface. However, you do not need to take this dictum literally. Even in standard Version 9 SDOs, you can easily put your table maintenance logic in a separate procedure (perhaps leave it in an existing procedure from your current application if the calling mechanism can be worked out), and run it from inside the standard hooks in the SDO.

For example, the `preTransactionValidate` procedure run just before the database transaction starts could simply turn around and call something else. Here is one example of what such code could look like. Note that if you were going to do this on a wide scale, it would be wise to develop a standard technique, perhaps using an include file, to wrap the old code in the new call. The standard SDO update buffer, `RowObjUpd`, is copied to a standard Customer buffer on the assumption that the existing procedure expects that buffer with no additional fields. The `bOldCustomer` buffer gets the original version of the record, as it was read from the database:

```
PROCEDURE preTransactionValidate:

DEFINE BUFFER bCustomer LIKE Customer.
DEFINE BUFFER bOldCustomer LIKE Customer.
DEFINE BUFFER bRowObjUpd LIKE RowObjUpd.
FOR EACH RowObjUpd WHERE RowObjUpd.RowMod = 'U':
    BUFFER-COPY RowObjUpd TO bCustomer.
    FIND bRowObjUpd WHERE bRowObjUpd.RowNum = RowObjUpd.RowNum AND
        bRowObjUpd.RowMod = ''. /* Blank indicates the Before copy. */
    BUFFER-COPY bRowObjUpd TO bOldCustomer.
    RUN CustomerLogic.p (INPUT bCustomer, INPUT bOldCustomer,
        OUTPUT cStatusMessage).
    IF cStatusMessage NE "" THEN RETURN cStatusMessage.
END.
```

This simplified example illustrates a couple of the shortcomings of the standard SDO logic definition. First, it requires a bit of extra work to associate the standard validation hooks with logic that is actually defined in another procedure. And second, the “wrapper” code that makes the association has some SDO-specific conventions in it.

If you define your logic directly in the SDO, your code has these same conventions:

- Using the SDO buffer name RowObjUpd, and using the extra fields RowMod and RowNum to identify which type of update operation (Update, Add, Copy, Delete) has occurred.
- Identifying the before version of each Updated row.
- Being prepared to loop through the table in the event that more than one record was changed in a single call.

The Progress Dynamics Version of SDOs is entirely compatible with existing ones, but it contains some enhancements to remove these SDO-specific logic elements. The goal is to make your logic more reusable and more flexible in how it is called.

When Dynamic SDOs are generated in the Object Generator, a logic procedure is created that contains all validation logic that is associated with the SDO and becomes a super procedure.

This makes it easier to call the same table maintenance logic from elsewhere in your application, where you might not be using SDOs as a data management mechanism. In this way you can truly use the same logic from everywhere regardless of the nature of the code.

The following illustrations might help to explain the position of the logic procedure in the Progress Dynamics architecture. The client application starts the SDO along with other client-side objects. Because there is no database connection on the client, the proxy version of the SDO is run. It, in turn, identifies the logical name of the partition in which its server-side counterpart should run, and starts the full SDO in that AppServer session.

Figure 10–5 shows this standard SDO mechanism.

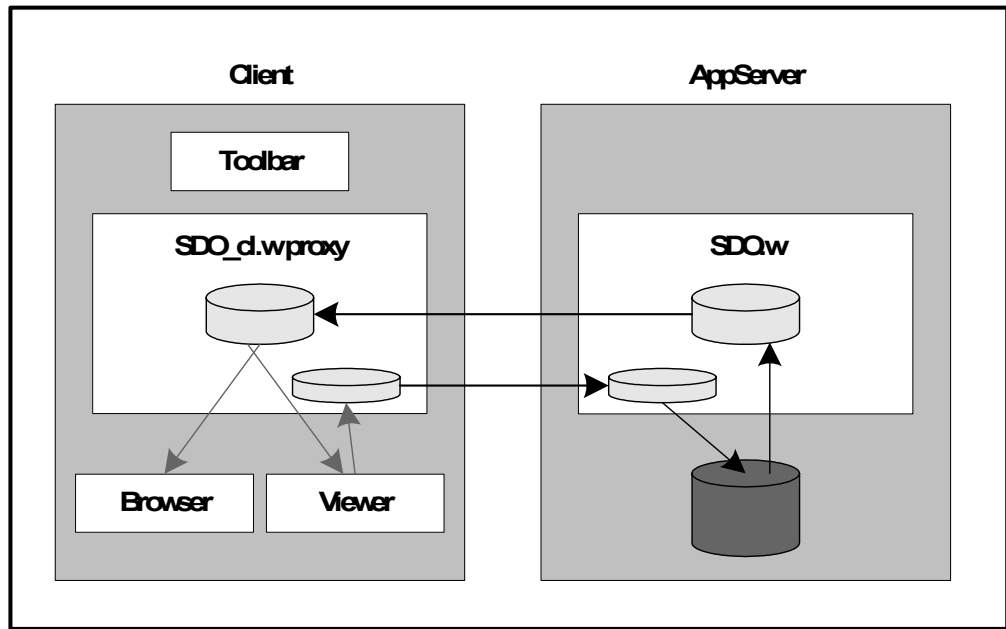


Figure 10–5: SmartDataObject on client and server

Data is read from the database into the server-side RowObject table and passed across the AppServer connection to the client, where it is made available to other client objects. Updates made on the client are collected in a separate copy of the SDO temp-table called RowObjUpd and passed back to the server, where the data is validated and written back to the database. You can code validation logic into the SDO procedure. Using the DB-REQUIRED technique described above determines whether a particular logic procedure is compiled into the proxy and server-side SDOs or into the server-side full SDO only.

In Progress Dynamics, a new pair of procedures is generated to hold specific business logic for the table. As with the SDO, there is really only one procedure that has any code in it: the same kind of client proxy wrapper procedure is used to allow the same source procedure to be compiled two ways, with or without the internal procedure and functions that require the database connection. Since the logic procedure is started as a super procedure of the SDO, its contents more or less transparently become part of the SDO, so that the same validation procedures can go into the logic procedure as previously went into the SDO. If you create a logic procedure for an existing SDO and simply copy all of the standard validation procedures into it, the SDO will work exactly the same as it did before.

The resulting setup looks like the illustration in [Figure 10–6](#). The logic procedures are started automatically by the SDOs on each side. The name of the logic procedure is a new property of the SDO. All validation logic goes into the logic procedure to be run on one side of the AppServer connection or the other. (As with SDOs themselves, this all works correctly if there is no AppServer at all. In that case, the server-side full SDO and logic object run on the client, with a local database connection.)

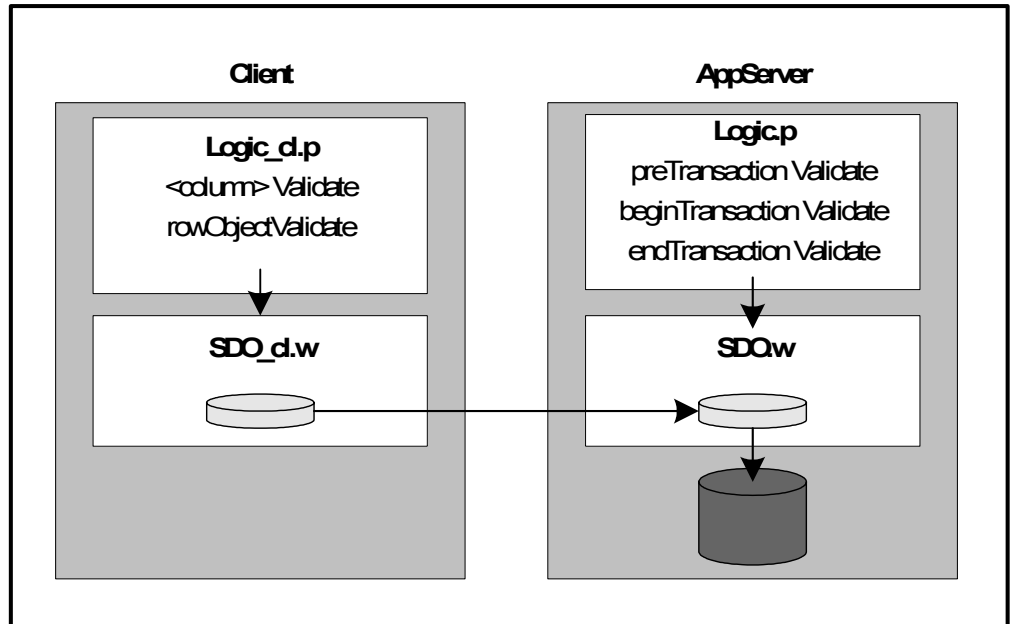


Figure 10–6: SmartDataObject with business logic procedure

Updates are collected on the client in the RowObjUpd table as before. New supporting code for the SDO makes these records available to the logic procedure.

One of the principal reasons for separating out the logic procedure is to allow the SDO itself to become a run time instance of a generic SDO, both on client and server. With no table-specific logic in the SDO itself, this becomes a relatively easy matter. [Figure 10–7](#) shows this process.

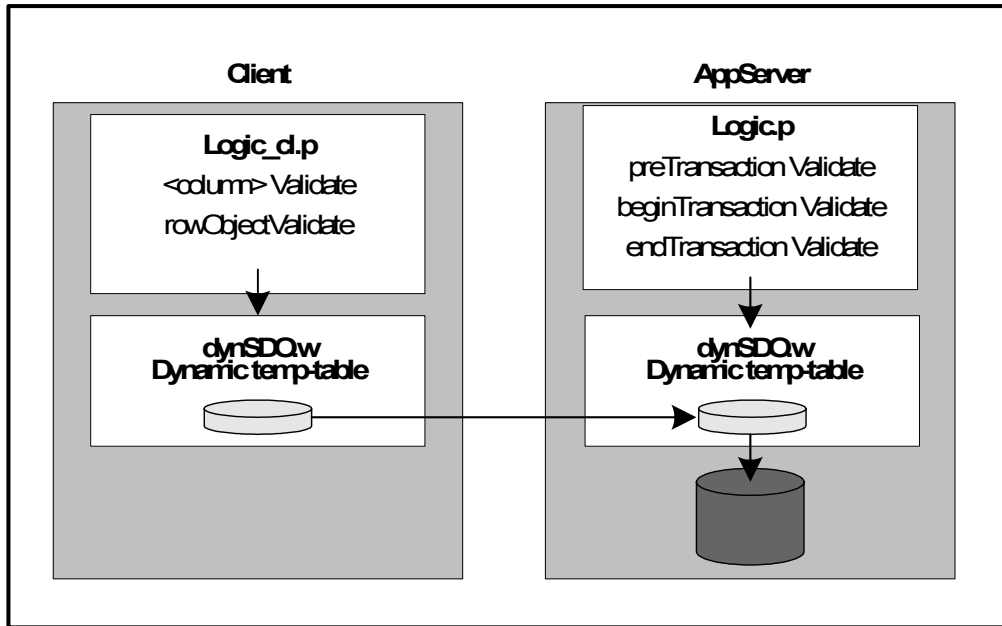


Figure 10–7: Dynamic SDO with business logic procedure

10.5.1 Calculated field support

Dynamic SmartDataObjects (and dynamic viewers) support calculated fields. When you add a calculated field to a dynamic SDO with the Calculated Field button, you can set the following attributes with the column editor: Name, Label, DataType, Format, and Help.

NOTE: A CalculatedField class extends the DataField class to support placement of calculated fields onto dynamic SDOs and dynamic viewers. This class should not be used and is not supported for any purpose other than calculated fields. Dynamic SDOs support calculated fields by invoking a function that follows a specific naming convention to calculate the values.

The Name attribute must be the same as the calculatedField object filename. You cannot specify an expression for the calculated field; therefore the framework disables the Edit button. You must write code to calculate the value for the field in a function of the dynamic SDO's data logic procedure and then name the function according to the required naming convention. The dynamic SDO invokes the function at runtime to get the calculated value into its temp-table.

The framework creates a master calculatedField object in the Repository for each calculated field added to a dynamic SDO. If a master calculatedField already exists in the Repository, the framework raises an error and you must change the name of that calculated field in the SDO. The calculatedField master object has the same product module as the SDO. Master calculated fields cannot be reused on dynamic SDOs. You can create a calculatedField instance object for the dynamic SDO in the same way as you create a DataField instance object.

Calculated field functions

Dynamic SDO calculated fields must have functions defined in the SDO's logic procedure called "Calculate" plus the field name. For example, a function called "CalculateNewLimit" is required in the data logic procedure to calculate the value for a calculated field called "NewLimit". The function does not have access to the RowObject buffer and needs to make appropriate calls to the SDO to access the buffer if it requires information from the buffer for the calculation. A function to calculate the NewLimit in the data logic procedure might look like the following example:

```
FUNCTION CalculateNewLimit RETURNS DECIMAL:
DEFINE VARIABLE hRowObject AS HANDLE NO-UNDO.
DEFINE VARIABLE hCreditLimit AS HANDLE NO-UNDO.
    hRowObject = DYNAMIC-FUNCTION('getRowObject':U IN TARGET-PROCEDURE).
    hCreditLimit = hRowObject:BUFFER-FIELD('CreditLimit':U).
    RETURN hCreditLimit:BUFFER-VALUE * 1.10.
END FUNCTION.
```

Dynamic Viewer calculated fields

You can add calculated fields from static or dynamic SDOs as instances onto dynamic viewers. When you create a dynamic viewer with a static SDO as its data source, a calculatedField master object might not exist for calculated fields of that SDO. When this occurs, the framework creates master calculatedField objects for the calculated fields and sets their product modules to that of the dynamic viewer.

Saving static SDOs with calculated fields as dynamic SDOs

When you save a static SDO that contains calculated fields as a dynamic SDO, the framework creates master calculatedField objects for your calculated fields and saves them in the dynamic SDO. The names of these calculated fields must be unique within the Repository. To complete the change over to a dynamic SDO, you must move the expressions for the calculated fields to calculation functions in the dynamic SDO's data logic procedure and edit the code as necessary.

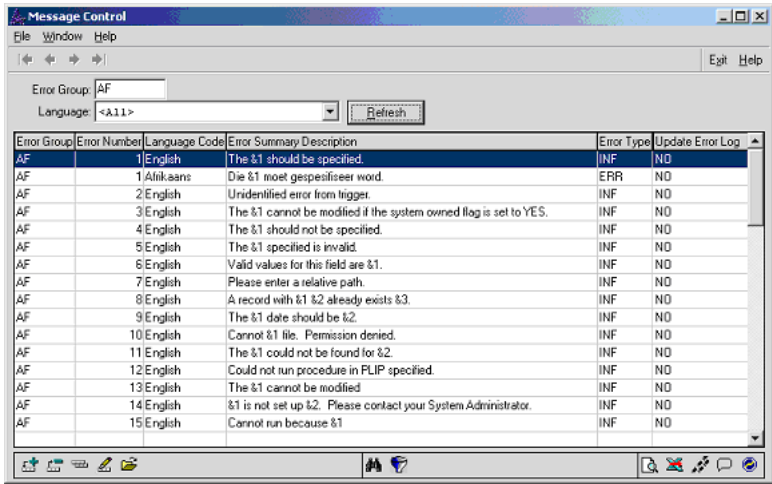
10.6 Message handling in Progress Dynamics

Some of the validation procedure examples above have introduced you to the Progress Dynamics convention for formatting messages to be returned to the client for display. This section describes the include files that you can use to facilitate this.

Messages in Progress Dynamics are stored in the Repository, so that the text and translations of all messages can be maintained independently of their use in applications. Follow these steps to see the messages that are shipped with the product, to add more messages for your application, or to translate existing ones:

- 1 ♦ From the Progress Dynamics Administration menu, select **System→Message Control**.

The Message Control browser lets you look at some of the characteristics of Progress Dynamics messages with which you need to be familiar to refer to them properly in your applications:



The screenshot shows the 'Message Control' application window. It has a menu bar with 'File', 'Window', and 'Help'. Below the menu bar are navigation buttons and 'Exit' and 'Help' buttons. There are two input fields: 'Error Group' with the value 'AF' and 'Language' with the value '<A11>'. A 'Refresh' button is to the right of the 'Language' field. Below these fields is a table with the following columns: 'Error Group', 'Error Number', 'Language Code', 'Error Summary Description', 'Error Type', and 'Update Error Log'. The table contains 15 rows of data, all with 'AF' as the Error Group and 'English' as the Language Code. The Error Types are either 'ERR' or 'INF', and the Update Error Log values are all 'NO'.

Error Group	Error Number	Language Code	Error Summary Description	Error Type	Update Error Log
AF	1	English	The %1 should be specified.	INF	NO
AF	1	Afrikaans	Die %1 moet gespesifiseer word.	ERR	NO
AF	2	English	Unidentified error from trigger.	INF	NO
AF	3	English	The %1 cannot be modified if the system owned flag is set to YES.	INF	NO
AF	4	English	The %1 should not be specified.	INF	NO
AF	5	English	The %1 specified is invalid.	INF	NO
AF	6	English	Valid values for this field are %1.	INF	NO
AF	7	English	Please enter a relative path.	INF	NO
AF	8	English	A record with %1 %2 already exists %3.	INF	NO
AF	9	English	The %1 date should be %2.	INF	NO
AF	10	English	Cannot %1 file. Permission denied.	INF	NO
AF	11	English	The %1 could not be found for %2.	INF	NO
AF	12	English	Could not run procedure in PLIP specified.	INF	NO
AF	13	English	The %1 cannot be modified	INF	NO
AF	14	English	%1 is not set up %2. Please contact your System Administrator.	INF	NO
AF	15	English	Cannot run because %1	INF	NO

Each message is assigned to a Message Group. (The Error Group label, as seen in the previous figure, is somewhat misleading. All messages, including informational messages that are not errors at all, are maintained together in the message table.) These groups are two-letter codes used simply to organize messages in a sensible way. The two predefined message groups in Progress Dynamics are AF, for messages that are a basic part of the application framework, and RY, for messages that are generated primarily by procedures responsible for maintaining the Repository itself. All of these messages are available for your use when you are building applications. You should use the existing messages wherever possible before defining additional messages of your own. The message numbers in the code generated by the Object Generator, for example, are references to standard messages in the AF group in the Repository.

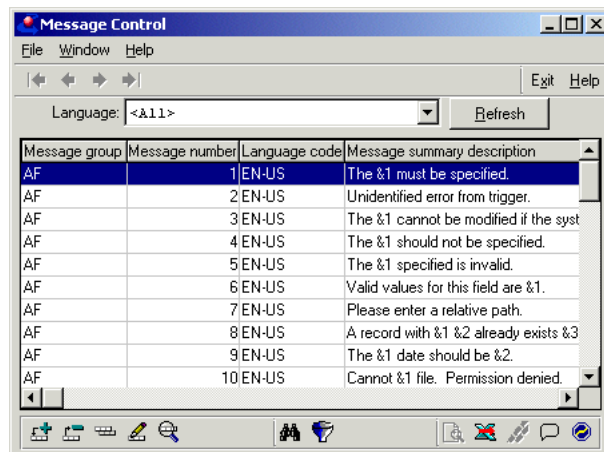
Within the Message Group, each message is given a unique number. The combination of Message Group and Message Number is unique throughout the application. It is these first two arguments to the `aferro.txt` include file that identify the message.

Each message also has a Language Code. As shown in the first lines of the above figure, you can translate each message into different languages. Each translation uses the same Message Group and Message Number, so that a reference to that key within an application identifies the message. The login language of the current user tells Progress Dynamics which translation to choose.

The third and fourth arguments to the include file are the table name and field name with which the message is associated. You should specify these arguments if they are meaningful.

Messages can contain additional arguments that are substituted into the base message text using the numbered replacement references (&1 etc.). Up to nine additional arguments can be passed to the include file, to be used as replacements in either the message summary or the longer message text that is defined in the Message Maintenance utility and displayed as part of the Progress Dynamics message dialog box.

- 2 ♦ To add a new message, choose **Add** in the Message Control.
- 3 ♦ To translate an existing message, choose **Copy**, since the Message Group, Number, and Type should be the same for the translation of the message. The Message Maintenance window appears:



You can select any language you have previously defined in the Language Control. (To run the Language Control, from the Progress Dynamics Administration window, select **Application→Language Control**.)

The Error Summary Description is a short form of the message. The editor at the bottom of the Maintenance screen lets you enter a longer description for the message, which will be displayed if the user expands the message display dialog box. These two descriptions share up to nine substitution parameters that can be passed along with the reference to the message in the application code.

10.6.1 Retrieving and formatting error messages with `aferrortxt.i`

The include file you use in a validation procedure of any kind to format a message to prepare it for return to the client, is `aferrortxt.i`. This include file has been designed to simplify and standardize the format of Progress Dynamics Messages. The include file supports the unnamed arguments listed in [Table 10–1](#).

Table 10–1: Supported arguments

Argument	Description	Example
{1}	The Progress Dynamics Message Group	AF, or ?, if using a hard-coded message
{2}	The Progress Dynamics Message Code	1 or a hard-coded message text
{3}	The table name with which the message is associated	Customer
{4}	The fieldname with which the message is associated	CustomerName
{5 - 13}	Extra insertion arguments for the message (a maximum of nine are supported).	–

Resulting message format

The whole purpose of `aferrortxt.i` is to shield you from being concerned about the exact format of the various components of a formatted message. For informational purposes, this section explains the resulting format.

The include file equates to a single string value of the formatted message with appropriate delimiters. The format conforms to that expected and used by ADM2, which is:

```
message + CHR(4) + field + CHR(4) + table
```

Additionally, the format adds on the following for extra information on where the error occurred:

```
+ CHR(4) + PROGRAM-NAME(1) + CHR(4) + PROGRAM-NAME(2)
```

The format of the message part conforms to that required by Progress Dynamics for reading of the error message from a message table in the database, which is:

```
"group^code^program-name(1):program-name(2)^insert1|insert2|insert3|etc."
```

This is a carat (^)-separated list starting with the message group, then the message code, then the program the message occurred in, then a pipe (|)-delimited list of optional insertion codes to replace in the found message text returned from the database table.

Where insertion codes exist in the message text, values must be passed into the appropriate include file argument as explained above. Note that arguments to the include file must be passed in as unquoted variable names or single-quoted literals (for example, 'text'). If a literal contains spaces, then it must be enclosed in double quotes, then single quotes (for example, "'text space'"). If double quotes are specified, these will be dropped automatically. If arguments need to be left blank, then a placeholder of '?' must be used if the argument is not the last item in the list. Only the first two arguments are mandatory.

An example of an include file for a standard Progress Dynamics message with no insertion codes is:

```
{aferrortxt.i 'AF' '1'}
```

If the message to be displayed is not in the message table but simply hard-coded in the application procedure, then as noted in the argument description, the first argument is replaced by a question mark to signal this, and the second argument is the message text. An example use of an include file for a hard-coded message with no insertion codes is:

```
{aferrortxt.i '?' "'message text'"}
```

This example shows a Progress Dynamics message with two insertion codes and the table/field names specified:

```
{aferrortxt.i 'AF' '1' 'gsm_user' 'user_login_name' 'insert1' 'insert2'}
```

This example uses a Progress Dynamics message with two insertion codes and no table/field specified:

```
{aferrortxt.i 'AF' '1' '?' '?' 'insert1' 'insert2'}
```

Since `aferrortxt.i` actually evaluates to the formatted message, you must place it into a context where that message will be returned appropriately. An example in an application is:

```
IF <condition> THEN RETURN ERROR {aferrortxt.i 'AF' '1'}.
```

Or, as used in the procedures that come out of the Object Generator (and as you should normally also code such procedures), potentially multiple messages are accumulated in a single string using this convention:

```
cMessageList = cMessageList +  
  (IF NUM-ENTRIES(cMessageList,CHR(3)) > 0 THEN CHR(3) ELSE '':U) +  
    {aferrortxt.i 'AF' '8' 'Employee' '  
      "'EmpNum, '" cValueList }.
```

You should take advantage of the message table wherever possible in your application. Using standard messages for errors and for any other message displays supports the translation and reuse of messages. Hard-coding a message in an application means that it cannot easily be translated. Any change to the message text requires a change to the source code of the application.

The insertion codes can be a valuable mechanism for enabling you to reuse essentially the same message in different circumstances. Do not overdo the substitutions, however. If what you are inserting is the name of a table, field, or value, or some other entity that does not require translation, then message reuse is maximized. However, if you put text that really requires translation into the insertion parameter, such as “Error - &1”, then you would be defeating the purpose of the message table, which is to put the meaningful text in one place where it can be maintained, customized where necessary, and translated.

10.6.2 Error message checking include file (checkerr.i)

The include file `aferrortxt.i` takes care of formatting messages into a standard format in preparation for returning them to the client to be presented to the user. As the examples above show, simply returning the message text string from any of the standard validation procedures results in the errors being processed correctly. The wrapper code that runs the validation procedures handles the messages for you.

In other cases, however, you might need to process errors yourself (for example, from stand-alone business logic procedures that you write, as described later). There is another standard include file called `checkerr.i` that you should always use when checking for errors within your own logic procedure, instead of doing an `IF ERROR-STATUS:ERROR` check manually.

The include file handles the checking of the error status. If an error occurred, it handles the displaying of the error message or passing on of the error message and return value back to the caller. It also contains code to build a return value from the Progress Dynamics get-message method if the `RETURN-VALUE` is empty. This is a very flexible include file and should cater for all scenarios. [Table 10–2](#) lists the named arguments.

Table 10–2: Named arguments for `checkerr.i`

(1 of 2)

Argument	Description
<code>&display-error = YES</code>	Display the error if not in a transaction. This invokes the standard Progress Dynamics message dialog box, and should therefore be done only if you are in client-side code that can present the message dialog box to the user.
<code>&return-only = YES</code>	Do not pass on the error but do a <code>RETURN</code> .
<code>&return-error = YES</code>	Pass on the error. The code defaults to this if there is currently an open transaction.
<code>&no-return = YES</code>	Do not return when finished; instead, this form lets you decide what to do in the code following the include file reference.

Table 10–2: Named arguments for checkerr.i*(2 of 2)*

Argument	Description
&ignore-errorlist = YES	Do not use get-message errors at all (use with caution). get-message is a standard Progress Dynamics method that retrieves error messages generated by Progress, in addition to any messages generated by the application.
&define-only = YES	Define the variables you need but take no other action. The checkerr.i include file defines several local variables that are needed in its processing, including cMessageList, cMessageButton, and cMessageAnswer, which are described below.

The &define-only argument is therefore useful if checkerr.i is included more than once in the same file but separate procedures. If you place this code at the top of your file, the necessary variables will be defined once, and no other action will take place:

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

When you are using checkerr.i in server-side business logic, normally some combination of the arguments in [Table 10–2](#) that handle the RETURN statement are the ones you will use in your code. If you are displaying an error message in client code from checkerr.i, then arguments listed in [Table 10–3](#) are also additionally available.

Table 10–3: Additional arguments for checkerr.i*(1 of 2)*

Argument	Description
&message-type	Type of message, one of 'MES' (ordinary message), 'INF' (informational message), 'WAR' (warning), 'ERR' (error), 'HAL' (halt), 'QUE' (question); the default = 'ERR'
&message-buttons	Comma list of buttons to be placed into the message dialog box, default is 'OK'.
&button-default	Default button, the default is 'OK'.
&button-cancel	Cancel button, default is 'OK' except for QUEstion messages.

Table 10–3: Additional arguments for checkerr.i

(2 of 2)

Argument	Description
&message-title	Title for the message dialog box; the default is the empty string.
&display-empty	YES—Display message dialog box even if empty, default = YES.

Table 10–4 lists the arguments available if the message type is 'QUE' for Question.

Table 10–4: Question arguments

Argument	Description
&message-data type	Data type, default is 'Character'.
&message-format	Format of question, default = 'CHR(35)'.
&default-answer	Default answer, default is empty.

NOTE: For a question message type, the defaults change to &message-buttons = 'OK,CANCEL', &button-default = 'OK', &button-cancel = 'CANCEL'.

The following rules apply when you use the checkerr.i include file:

1. Required logical arguments must be passed unquoted as YES or NO. Other text arguments must be single quoted literals (for example, 'text' or unquoted variables) and if the literals require spaces, they should be double-quoted then single-quoted (for example, "'text'").
2. All arguments can be left blank, in which case the error will not be displayed and will be passed on with a return error.
3. If everything is left blank except display-error = YES, then the error will be displayed using the Progress Dynamics message dialog box and a standard return statement done.
4. If in a transaction, the error will be passed on regardless, unless {&no-return} is set to YES.
5. If display error is set to YES, the default is to do a return only rather than return error.
6. If display error is set to NO, the default is to do a return error to pass the error on.
7. If return-only is set to YES, this will override any other return settings and let you regain control.

Following the include file, if {&no-return} is set to YES. [Table 10–5](#) lists the variable values available.

Table 10–5: NO-RETURN variables

Variable	Value
cMessageList	CHR(3)-delimited list of error message.
cMessageButton	Button Pressed.
cMessageAnswer	Answer if question dialog box used.

Here are a few examples of using the checkerr.i include file. First, another standard Progress Dynamics include reference is required in every procedure that uses other Progress Dynamics includes such as checkerr.i. This is afglobals.i, which defines global references to some of the Progress Dynamics Managers.

This example runs a business logic procedure using the launch.i mechanism (described in more detail in [Chapter 11, “Building Advanced Business Logic in a Progress Dynamics Application.”](#)) It then uses checkerr.i upon return to intercept any errors that occur. For the sake of simplicity, the first example simply displays the error, using the standard Progress Dynamics message dialog box, so that you can see how that works, even though you would of course not invoke this display directly from a server-side logic procedure.

Here is the code that launches the procedure and a reference to checkerr.i to display an error, if there is one. Pass in the name of the external procedure to run persistent on the server, the name of the internal procedure to run in its handle, whether to invoke it on the AppServer, and a list of parameters to it. The internal procedure validateDatasetQuery uses aferrortxt.i to return error number AF:114 if there is an error in the names of the input parameters. It is this error that you intercept here:

```
{afglobals.i}
{launch.i &PLIP = 'af/app/gscddxmlp.p'
      &IProc = 'validateDatasetQuery'
      &OnApp = 'NO'
      &PList = "('rycso','rycsf','smartobject_obj,smartobject_obj','',no)"
      &AutoKill = YES}

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

If you modify the parameter list to the `validateDatasetQuery` procedure so that it is no longer valid, an error message is generated and picked up by `checkerr.i`:

```
{afglobals.i}
{launch.i &PLIP = 'af/app/gscddxm1p.p'
      &IProc = 'validateDatasetQuery'
      &OnApp = 'NO'
      &PList = "('rycso','rycsf','smartobject_obj,smartxxxxx_obj','','no)'
      &AutoKill = YES}
{checkerr.i &display-error = YES}.
```

10.6.3 The Progress Dynamics message dialog box

The standard message dialog box used throughout Progress Dynamics is shown in [Figure 10–8](#).

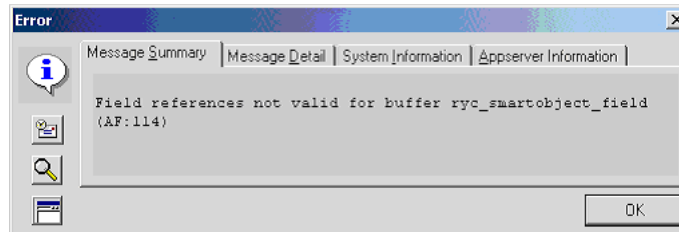


Figure 10–8: Standard Message dialog box: Message Summary tab

The first tab displays the Message Summary, or short form of the message. It shows the message text, in the appropriate language for the user, the error Message Group and Message Number (AF:114), and the message type (“INF” for informational, as indicated by the “i” symbol at the upper left). You can see that the default title is Error, and the default button is **OK**.

[Figure 10–9](#) shows the second tab, Message Detail, the longer form of the message that can be entered into the message definition in the message table in the Repository database. You can expand the dialog box to full-page size by choosing the window icon at the lower left.

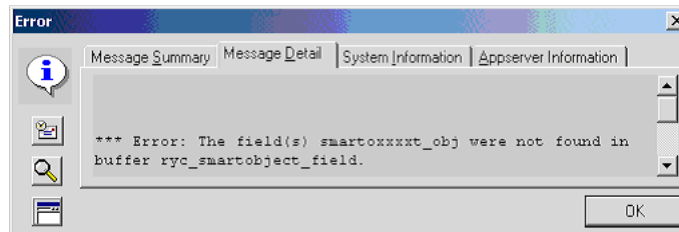


Figure 10–9: Standard Message dialog box: Message Detail tab

The third tab shows a host of system information about the user, the platform on which the application is running, what procedures are running, etc. [Figure 10–10](#) shows just a sample of this information.

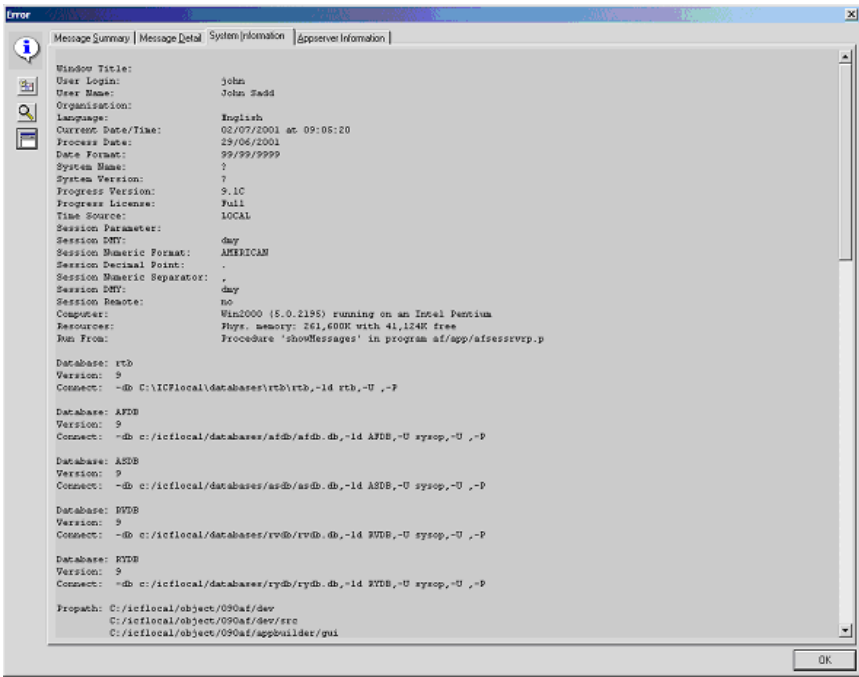


Figure 10–10: Standard Message dialog box: System Information tab

The fourth tab shows information about the AppServer connection if the application is running with an AppServer. If there is no AppServer connection, then this tab will be blank.

The magnifying glass icon brings up a Progress stack trace. The letter icon sends a message to a technical support organization or other responsible administrator if this is enabled.

This next code example shows a few of the formatting options when the message is being displayed (&display-error = YES). The &message-type has been set to WAR to signal a warning; the set of &message-buttons has been set to OK and CANCEL; and the &message-title is set to 'Sample Message'. Note the use of nested quotation marks in specifying the value Sample Message. This is a standard requirement of the Progress 4GL include file mechanism when an argument has nested white space:

```
{afglobals.i}

{!launch.i &PLIP = 'af/app/gscddxmlp.p'
  &IProc = 'validateDatasetQuery'
  &OnApp = 'NO'
  &PList = "('rycso','rycsf','smartobject_obj,smartobjectt_obj','',no)"
  &AutoKill = YES}

{checkerr.i &display-error = YES
  &message-type = 'WAR'
  &message-buttons = 'OK,CANCEL'
  &message-title = "'Sample Message'"
}.
```

When you run the procedure again with the same error condition, the message dialog box is changed, as shown in [Figure 10–11](#).

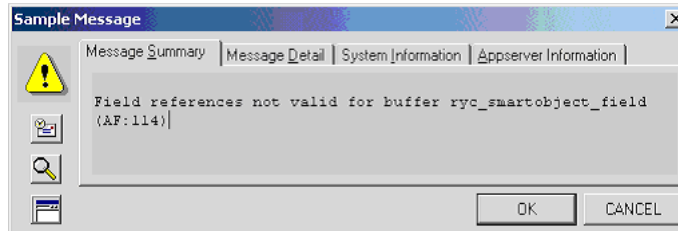


Figure 10–11: Sample Message dialog box: Message Summary tab

A server-side business logic procedure would not display the error directly or manipulate its format. It might, however, want to intercept the error in some other way using one of the RETURN-related include file arguments. For example, if you change the include file reference to specify &no-return, then the code block simply continues on following the error. In this case, the code can use one of the local variables defined as part of the error-checking mechanism.

The next example specifies `&no-return`, and then displays the message text stored in the variable `cMessageList`:

```
{af/sup2/afglobals.i}

{launch.i &PLIP = 'af/app/gscddxmlp.p'
  &IProc = 'validateDatasetQuery'
  &OnApp = 'NO'
  &PList = "('rycso','rycsf','smartobject_obj,smartxxxxt_obj','',no)"
  &AutoKill = YES}

{af/sup2/checkerr.i &no-return = YES }.

MESSAGE cMessageList.
```

The message display reflects the complex format of the various components of the message as explained previously.

Figure 10–12 shows the unadulterated message text that results. In very special cases, your logic might want to take this text apart to determine the nature of the error. In most cases, however, the text is simply returned to the standard display mechanism, which deals with the message format for you.

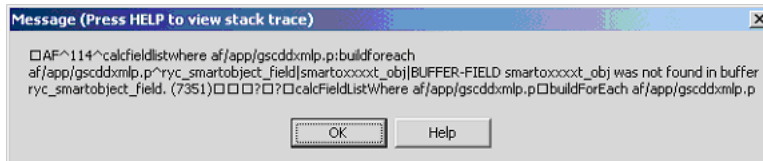


Figure 10–12: Unformatted message display

10.7 Summary

This chapter has described the structure of the SDO and how you can add business logic to the SDO to validate data in what will most commonly be a single-table update plus updates to related tables on the server. It has also shown you how to define messages in the Repository and use them in your application.

For a description of more advanced business logic, see [Chapter 11, “Building Advanced Business Logic in a Progress Dynamics Application.”](#) It covers the following topics:

- Information on using the SmartBusinessObject (SBO) to define an update spanning multiple, related tables that are modified on the client and then saved as part of the same transaction.
- Other techniques for handling parent-child record display and update in a single transaction.
- Guidelines for using database trigger procedures as part of your application logic.
- A description of how to build SDOs that operate against a temp-table created on the server rather than a database table.
- Writing your own custom business logic procedures to run on the server, and invoking those from client application code.

Building Advanced Business Logic in a Progress Dynamics Application

Chapter 10, “Building Basic Business Logic in a Progress Dynamics Application,” describes the basics of building business logic into your Progress Dynamics™ application. In particular, it describes building and using SmartDataObjects (SDOs) and using the Progress Dynamics’ message management tools.

This chapter continues the discussion of incorporating business logic with the following other ways to organize more complex business logic:

- [Building SmartDataObjects against temp-tables](#)
- [Database triggers and Progress Dynamics](#)
- [Running business logic procedures in Progress Dynamics](#)
- [SBO overview](#)
- [Building and using SmartBusinessObjects](#)
- [Example: creating an SBO](#)
- [Defining business logic for SBOs](#)
- [Conclusion](#)

11.1 Building SmartDataObjects against temp-tables

When your query is based on a derived dataset not directly represented in the application database, you can build SDOs against a temp-table. This section describes doing it with the following topics:

- [Temp-table basics](#)
- [Populating the SDO's temp-table](#)
- [Defining an SDO with a temp-table like a database table](#)

11.1.1 Temp-table basics

SmartDataObjects use a temp-table called RowObject as an intermediary for all data access. Data from a database is loaded into the RowObject table defined in an SDO. The data is then made available to client objects for viewing and updating.

Sometimes, the original source of the data might also be a temp-table. Perhaps the data is not coming in from a Progress database (or Progress DataServer), but is instead loaded into a temp-table by application code. The data might represent a set of calculations or summary records created by reading other database records. In any case, applications frequently use a temp-table as the original data source.

The Progress AppBuilder and SmartDataObject™ wizard support the definition of an SDO against a temp-table. A temp-table can be as complex as a database table. Its fields can have data types, formats, and other properties. The simplest way to define a temp-table that has a definition unlike any existing database table is to use the Progress Data Dictionary to define the temp-table as if it were a real database table.

The AppBuilder supports a special naming convention for this. If you define a database and connect it using the logical database name, `temp-db`, the AppBuilder treats table definitions from this database specially. The AppBuilder recognizes the tables in this database as temp-table definitions when you build your SmartDataObjects. You might also want to use a temp-table that is identical to a database table, possibly with a few added fields. You can simply choose this table as the basis for your SDO as you build it. The following examples illustrate both of these cases.

NOTE: You must have `temp-db` connected to edit dynamic SDOs and viewers that use temp-tables in AppBuilder. `temp-db` tables must be imported into entities with data fields.

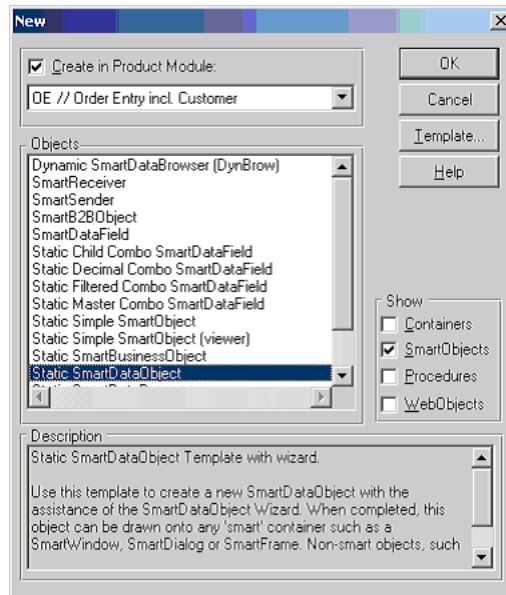
Because there is no database table to serve as the data source, you must load the temp-table records yourself. You can do this easily either in the Main Block of the SDO or in a local `initializeObject` procedure.

As an example, create a temp-db database with a table TTMTable in it, with two field definitions, Field1 and Field2. This is an example of a temp-table that is not derived from an actual database table. Therefore, you need to define it in this “dummy” database so that both the AppBuilder and the Progress compiler can access the table definition at design time. When you run your finished application, however, you will not need to connect this database.

The Object Generator introduced in [Chapter 5, “Using the Object Generator,”](#) generates SDOs and other objects for database tables. However, at present you cannot automatically generate SDOs for the temp-db in the Object Generator because it considers temp-db to be one of the special database names associated with the Repository, and excludes it from the drop-down list of databases to choose from.

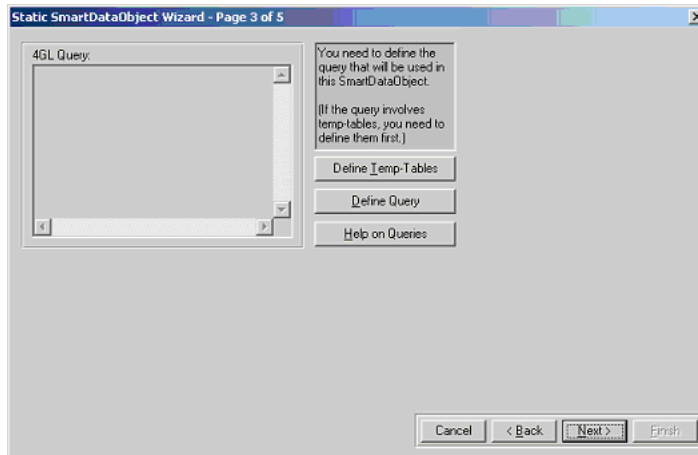
However, you can follow these steps to create an SDO yourself in the AppBuilder:

- 1 ♦ From the **New** dialog box, select **Static SmartDataObject**:

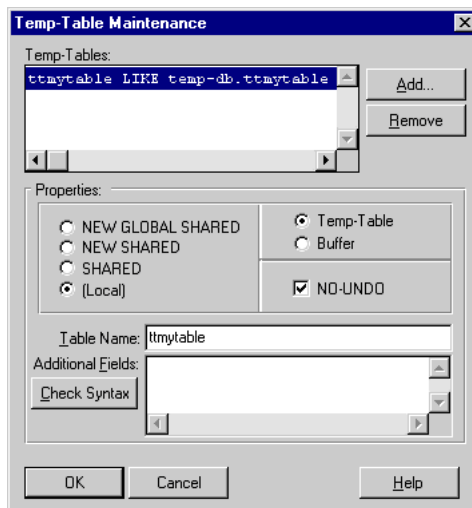


- 2 ♦ In the second page of the SDO wizard, fill in the SDO procedure name and other information that you want added to the comments section at the top of the source file.

- 3 ♦ On the third page of the wizard you must choose **Define Temp-tables** to choose your temp-db temp-table:



- 4 ♦ In the Temp-table Maintenance dialog box, choose **Add**.
- 5 ♦ Select the temp-db database and your table name. The AppBuilder generates the DEFINE TEMP-TABLE TTMyTable LIKE TTMyTable statement, allowing the definition of the table to be compiled into the SDO without requiring that the table definition be available at run time.
- 6 ♦ Choose **OK** to return to the Temp-table Maintenance dialog box:



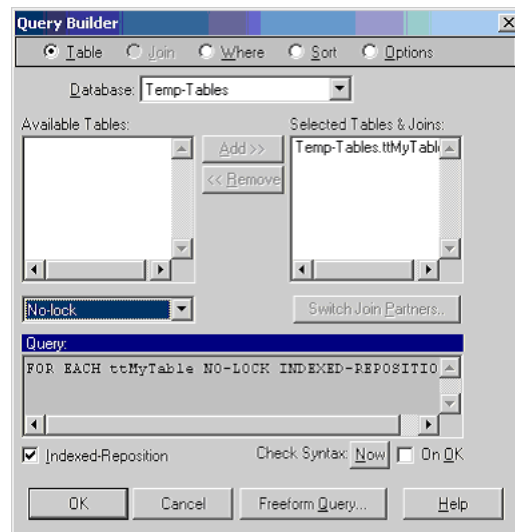
- 7 ♦ Choose **OK** to return to Page 3 of the wizard.

Now you must define the query for this table.

- 8 ♦ Choose **Define Query**.

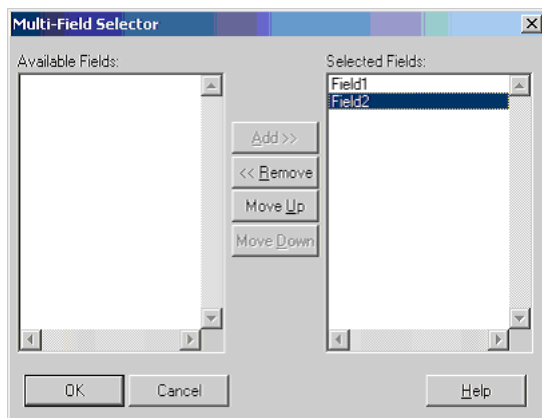
Because the AppBuilder recognizes the temp-db database as being special, it generates the alias Temp-tables as the database name for any temp-tables you have defined.

- 9 ♦ Select that database name, then choose **Add** to add TTMyTable to the SDO query:

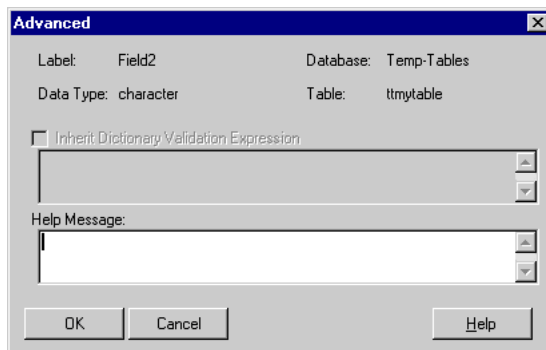


- 10 ♦ On the fourth page of the Wizard, choose **Add Fields**.

- 11 ♦ Select fields from the Temp-table as you would for any other table. Both fields are selected from our test temp-table:



- 12 ♦ If you select the Advanced button in the Column Editor, you will see the Inherit Dictionary Validation option. This option is normally turned on when you build an SDO in the AppBuilder, so that visual objects such as SmartDataViewers that use this SDO will inherit the Dictionary Field Validation that is executed when the user leaves each field. (It is automatically disabled for fields from temp-tables.) In this case it is off, as previously shown, because Progress temp-tables do not support Field Validation. You should turn off field validation so that it is not included in SDOs built against database tables:



11.1.2 Populating the SDO's temp-table

Remember that you are responsible for populating the temp-table your SDO uses. The following simple example shows how you can create 10 records in your TTMyTable table when the SDO is created:

```
/* ***** Main Block ***** */
DEFINE VARIABLE iRec AS INT NO-UNDO.
DO iRec = 1 TO 10:
  CREATE ttmytable.
  ASSIGN ttmytable.Field1 = iRec
         ttmytable.Field2 = 'Rec ' + STRING(iRec).
END.

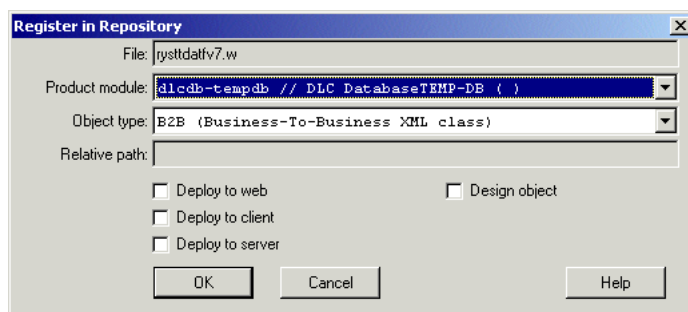
&IF DEFINED(UIB_IS_RUNNING) <> 0 &THEN
  RUN initializeObject.
&ENDIF
```

Because you created this object directly in the AppBuilder, you need to register it in the Progress Dynamics Repository. Follow these steps anytime you create or open an object in the AppBuilder and want to register it in the Progress Dynamics Repository:

- 1 ♦ Save the procedure file.
- 2 ♦ From the AppBuilder main window, select **File**→**Register in Repository**.

NOTE: In Progress Dynamics Version 1.1 it is necessary to close the object and reopen it as a file before the Register in Repository menu item will be enabled. This requirement will be corrected in a later release.

The Add to Repository dialog box appears:



- 3 ♦ Select the **Product Module** where the object should be registered.
- 4 ♦ Select **Static SmartDataObject (SDO)** as the Object Type, then choose **OK**.

Now you can build SmartDataViewers or SmartDataBrowsers for this SDO as you would for any other SDO. Keep in mind that the reason the SDO temp-table definition needs to match the name and definition of a table that can be located in a database at design time and compile time, is that visual objects, such as the SmartDataViewer, need to reference that same table definition when they are compiled. There is no need for the table definition itself at run time in these visual objects, but it is needed to compile the field definitions that make up the Viewer or Browser.

When you drop these objects into a SmartWindow or other Container, you can use them just like any other SmartObjects. The records loaded into the TTMyTable table are copied into the RowObject Temp-table just as database records would be. [Figure 11–1](#) shows a SmartWindow with a SmartDataViewer that displays the current row from the temp-table.

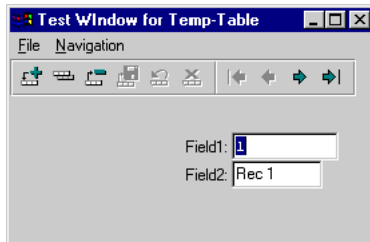


Figure 11–1: Test Window for Temp-Table

You can update rows in this table or add new rows to it the same as you could when using any other SDO, as shown in [Figure 11–2](#).

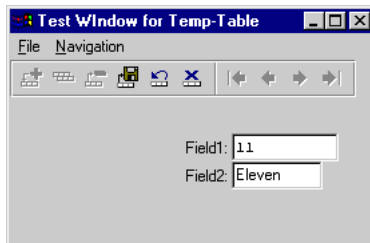


Figure 11–2: Updated Test Window for Temp-Table

These rows are copied back to the TTMyTable temp-table by the SDO code. Because this is just a temp-table, any changes are lost when the object is closed, unless you write code in your application to save the changes somehow. The following code example adds a local `destroyObject` procedure to the SDO `mytableo.w` so that changes are saved (or displayed in this case) on exit:

```

/*-----
Purpose:   Local version of procedure destroyObject.
           Saves changes to the TTMyTable Temp-table on exit.
Parameters: <none>
Notes:    This example simply displays the Temp-table contents
           to verify that changes have been saved back to the
           Temp-table.
-----*/
DEFINE VARIABLE cMessage AS CHAR NO-UNDO.
FOR EACH TTMyTable:
    cMessage = cMessage + string(TTMyTable.Field1)
                + " " + TTMyTable.Field2 + CHR(10).
END.
MESSAGE cMessage VIEW-AS ALERT-BOX.

RUN SUPER.
END PROCEDURE.

```

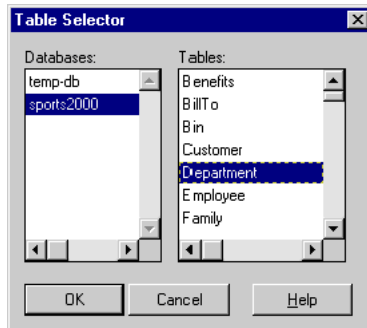
After making a change to the first record and adding an eleventh record to the table through the SmartDataViewer, and then exiting, you can see the confirmation that your changes made it back to the TTMyTable temp-table:



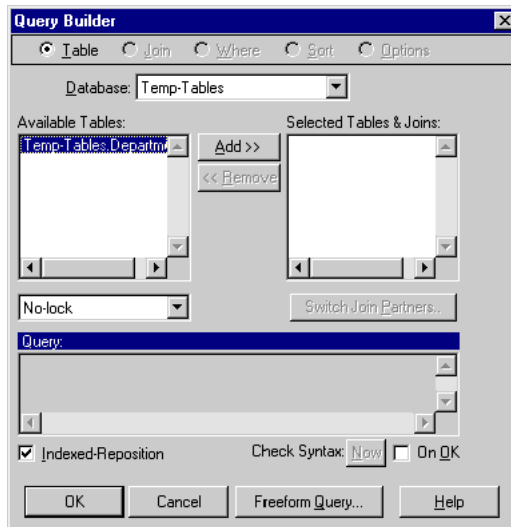
11.1.3 Defining an SDO with a temp-table like a database table

Follow these steps to define a temp-table that is identical to an actual database table:

- 1 ♦ From the SDO wizard, choose **Define Temp-tables**.
- 2 ♦ In the Temp-table Maintenance dialog box, choose the database table instead of a table from the temp-db database:



- 3 ♦ In the Query Builder, select the temp-table version of your table from the Temp-tables pseudo-database rather than the database table itself:



Progress can properly distinguish between a Temp-table and a database table of the same name, as in the example Department Temp-table SDO. References to the unqualified table name should be taken to mean the Temp-table. You can specify the database table using the database name qualifier. So in the Department SDO, you have an initialization block of code in the procedure's Main Block similar to the one for the first TTMyTable example. In the following code, you define some place-holder records of your own, and then copy the actual department names and codes from the database table into the Temp-table:

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

&IF DEFINED(UIB_IS_RUNNING) <> 0 &THEN
  RUN initializeObject.
&ENDIF

DEFINE VARIABLE cMessage AS CHAR NO-UNDO.
DEFINE VARIABLE iRec AS INT NO-UNDO.

DO iRec = 1 TO 9:
  CREATE department.
  ASSIGN department.DeptCode = '10' + STRING(iRec)
    department.DeptName = 'Dept ' + STRING(iRec).
END.
FOR EACH sports2000.Department:
  CREATE department.
  ASSIGN department.DeptCode = sports2000.department.DeptCode
    department.DeptName = sports2000.department.DeptName.
END.

FOR EACH department:
  cMessage = cMessage + department.deptcode
    + " " + department.deptname + CHR(10).
END.
MESSAGE cMessage VIEW-AS ALERT-BOX.
```

Displaying all of those values when the SDO starts up confirms that Progress has correctly interpreted your references to the sports2000.Department database table and the Department temp-table, as shown in [Figure 11-3](#).

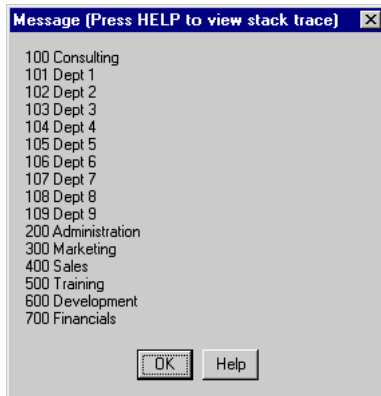


Figure 11-3: All values confirmation message box

Although you must define each Viewer or Browser you build using a particular SDO, you can connect these visual objects at run time to any SDO that supplies the right fields for them. In the case of the Viewer, this means that the fieldnames the Viewer asks for must be in the SDO that is its Data-Source. They do not have to be in the same order, they do not have to be all the fields defined in the SDO, and it does not matter whether the SDO uses a temp-table or an actual database table as its data source. So a Viewer built against a database table (meaning that a SmartDataObject defined against a database table was named in the Viewer's wizard) can be linked to an SDO built against a similar temp-table and used with it in an application. Because it is simply asking for field values, it will not know the difference.

11.2 Database triggers and Progress Dynamics

SmartDataObjects are intended to represent a single point-of-update for each database table. As such, it is theoretically possible to enforce all referential integrity within the SDO itself and avoid using Database Trigger procedures.

However, avoiding triggers is not necessarily wise. Sometimes you might have to write low-level AppServer routines to update the database directly without going through an SDO. It is imperative that you maintain Referential Integrity (RI) constraints in these circumstances.

The biggest argument against DB triggers is that they are not easy to maintain or document. However, with the use of relational design tools you can overcome both of these problems.

If you are using the ERwin modeling tool with Progress Dynamics, you should retain the Referential Integrity (RI) checks automatically generated in the DELETE and WRITE triggers.

This includes deletion cascade/restrict of records. In other cases, where the database schema is not generated using *ERwin* and all the Progress Dynamics naming conventions are not necessarily observed, then you can write trigger procedures that do basic referential integrity checks. If your application already includes such procedures, then you can keep them.

Where a delete restrict is conditional (for example, you cannot delete the Order if it has OrderLines, unless you are also deleting the Customer), the Customer SDO should delete OrderLines on delete of the Customer. Thus once the Customer DELETE event occurs in the database, the Orders remain but without any OrderLines. In this way the RI constraint (on parent delete restrict) is enforced in all other circumstances.

Where you need to make customizations to triggers, outside of referential integrity, you should almost always make such changes in the SDO itself. You should not need to edit *ERwin*-generated triggers.

In summary, where database trigger procedures are concerned, it is good practice to restrict their behavior to essential referential integrity checks. Any more complex validation constitutes application business logic, and you should move it up to the application logic procedures that are more visible, more maintainable, and more flexible.

In addition, in those cases where the logic is an RI check that will result in an error if the check fails (a CAN-FIND lookup, for example), it is good practice to duplicate even that minimal logic that goes into database triggers in the application logic procedures. In this way you can expect that an update to the database made through Progress Dynamics application components will never fail at the level of the database triggers, because the same check will have been made higher up. This prevents you from worrying about defining and capturing error conditions and error messages generated by trigger procedures. In particular, it is essential to keep in mind that trigger procedures execute on the AppServer when your application runs in a distributed environment. If a trigger procedure generates an error message using the Progress 4GL MESSAGE statement, that message will never be seen on the client.

Also, keep in mind that the use of controls such as Progress Dynamics Combos and Lookups virtually assures that basic referential integrity checks will not fail, at least when updates are done through the application components you build, because the user will be forced to select a valid value for a key field from a list generated dynamically from the database.

11.3 Running business logic procedures in Progress Dynamics

As Progress Dynamics supports a stateless environment, you must write business logic as concisely and efficiently as possible. When your environment uses one or more AppServers, which is considered the norm, a call to a business logic procedure will ordinarily result in a connect to an AppServer session (activation), the running of the code, and then the disconnection from the AppServer session (deactivation).

If you run a procedure persistently on the AppServer, then you will bind the connection to the AppServer until you delete the persistent procedure. You must take extreme care not to bind AppServer connections unnecessarily, as doing so negates the scalability of Progress Dynamics applications.

While you should keep persistent procedures to a minimum, when you use them, you should use the standard Progress Dynamics mechanism for invoking business logic procedures on an AppServer. This is supported by the Progress Dynamics Session Manager, in order that business logic procedures can be initiated on the AppServer and then accessed from client application sessions in the most efficient way possible.

These Progress Dynamics business logic procedures are called Persistent Libraries of Internal Procedures (PLIPs). They are external procedures (.p's) that contain many internal procedures, run persistently (by a call to `launchProcedure`, normally handled by the Session Manager). The internal procedures usually relate to API calls relevant to a particular object. For example, a PLIP could be defined for various supporting logic for client updates. The internal procedures inside that PLIP could be called to validate particular parts of the client information or execute additional business logic, for example, when address information is changed.

If business logic needs to identify whether it is running locally or remotely, then you can check the `session:remote` flag and `SESSION:PARAM = "REMOTE"`. If either of these is true, then the code runs remotely (on an AppServer), otherwise it runs locally on the client. You must also check the session parameter, since when running in a WebSpeed environment, the `session:remote` flag returns as false.

All business logic in Progress Dynamics that needs to be run persistently must be run by a standard include file, called `launch.i`, that encapsulates calls to `launchProcedure`. This include file takes one or more of the named arguments listed in [Table 11-1](#).

Table 11-1: `launch.i` named arguments (1 of 2)

Argument	Description
{&PLIP}	Physical procedure (PLIP) name, for example, <code>ac\app\actaxplipp.p</code> .
{&Iproc}	Internal procedure, for example, <code>calculateTax</code> or left undefined to just start the PLIP.
{&PList}	Parameter List, for example, <code>(OUTPUT cTaxAmount)</code> or left undefined for no parameters.
{&OnApp}	Run on AppServer flag YES, NO, or APPSERVER default = YES.

Table 11–1: launch.i named arguments*(2 of 2)*

Argument	Description
{&Partition}	AppServer partition name.
{&Perm}	Run permanently YES/NO, default = NO (do not let agent kill it).
{&AutoKill}	Auto kill PLIP after run; YES/NO, default = NO.
{&NewInstance}	New instance YES/NO, default = NO (use running instance if any).
{&Define-only}	YES = define the variables you need but take no other action.

The following rules apply to the use of `launch.i`:

1. Required logical arguments must be passed in unquoted as YES or NO. Other text arguments must be single quoted literals (for example, 'text') or unquoted variables. If the literals require spaces, they should be double quoted then single quoted (for example, "'text'").
2. The only exception to rule #1 is for the parameter list, which does not support a variable, so only double quotes must be used.
3. You can omit the internal procedure so you can manually run procedures in the PLIP. The PLIP handle is made available in `hPlip`, a standard Progress 4GL variable.
4. If the `&OnApp` parameter is not specified, it defaults to YES, which means that the procedure runs on the AppServer, if one is connected, otherwise it runs locally. Passing in `APPSERVER` causes the procedure to fail if the AppServer is not connected.
5. If the `&OnApp` parameter is not set to NO, then you can specify a partition that will be connected, if not already connected. If you do not specify a partition, it defaults to the standard AppServer partition and runs on the `gshProgress DynamicsAppserver` handle.
6. If the `&Perm` parameter is passed in as YES, then the procedure is not killed when the agent disconnects, leaving it running for other connections. This is not the norm.
7. If `&AutoKill` is set to YES, then following the run of the internal procedure in the persistent procedure, the PLIP is automatically closed. This is useful when you are running routines on an AppServer to ensure the connection does not remain bound.
8. If `&NewInstance` is set to YES, then it does not check for an already running instance of the PLIP to use. Instead, it runs a new version.

Following the use of the include file, the variable values noted in [Table 11–2](#) will be defined and available.

Table 11–2: Variable values available

Variable	Definition
hPlip	Handle of the PLIP that was run
cPlipErrorButton	Error button chosen, if any

If the run failed for any reason, the hPlip handle will be invalid and any errors will have been displayed, if possible.

EXAMPLES

```
/* All parameters passed */
{launch.i      &PLIP = 'af/app/aftstplipp.p'
               &IProc = 'objectDescription'
               &PList = "(output cHello)"
               &OnApp = 'yes'
               &Partition = 'Progress Dynamics'
               &Perm = NO
               &Autokill = NO
               &NewInstance = NO
               &Define-only = NO}
```

```
/* Just required parameters passed accepting defaults for rest */
{launch.i      &PLIP = 'af/app/aftstplipp.p'
               &IProc = 'objectDescription'
               &PList = "(output cDescription)"}
}
```

dynlaunch.i

This include file is only recommended for Appserver calls where the procedure handle is not need after the call. It encapsulates the Dynamic Call Wrapper and therefore all Dynamic Call Wrapper comment s apply.

`dynlaunch.i` accepts a temp-table of call information. The call is then constructed from this information, and invoked. This process does result in a performance overhead, it is thus recommended that users only use this call under the following conditions:

- When making calls across the Appserver, where the benefits of reduced Appserver calls and an unbound connection outweigh the disadvantage of the overhead associated with the dynamic call.
- When call parameters are only known runtime.

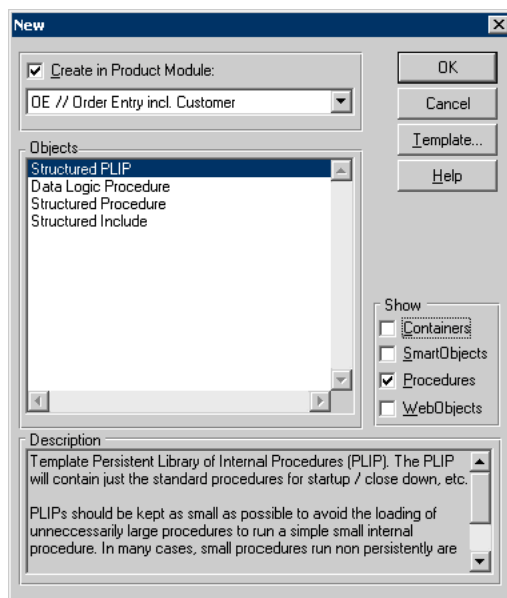
When making calls across the Appserver, the call is constructed and invoked from a non-persistent procedure on the Appserver. The call results are stored in the temp-table, and returned to the user. This results in one Appserver call, with the agent not bound. Note that persistent procedures handles are not available to the client after the call has completed. If the user wishes to run a procedure persistently, and use the procedure handle after the call, it is recommended `launch.i` is used.

11.3.1 Creating PLIPs

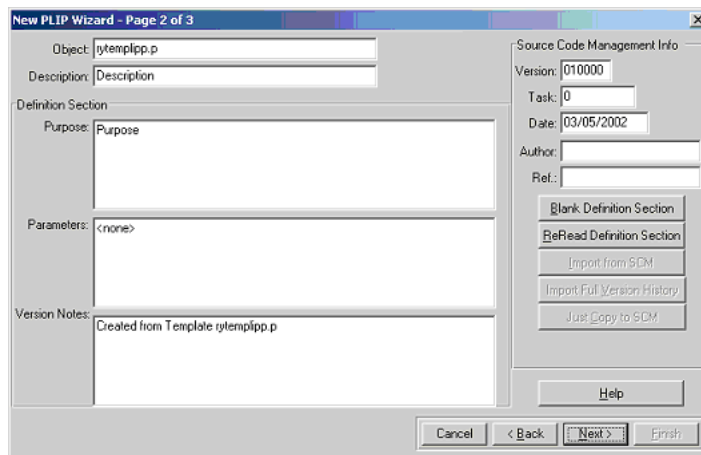
There is a Progress Dynamics structured procedure template for PLIPs. If you create a new procedure using this template, you can then place your own internal procedures inside it and invoke them through the Session Manager using the `aflaunch` include file.

To create a new PLIP, follow these steps:

- 1 ♦ From the AppBuilder, choose **New** or select **File→New**. The New dialog box appears:



- 2 ♦ Select **Structured PLIP** as the object type to create in the AppBuilder. The wizard prompts you to identify the name, description, and purpose of the procedure for documentation purposes:



Your PLIP will contain a few standard include file references, which provide it with the structure it needs, in the Main Block, and in the procedures `killPlip`, `plipSetup`, and `plipShutdown`.

There is also a standard `objectDescription` procedure, shown in [Figure 11–4](#), where you can specify a meaningful description string for your procedure that can be returned to callers.

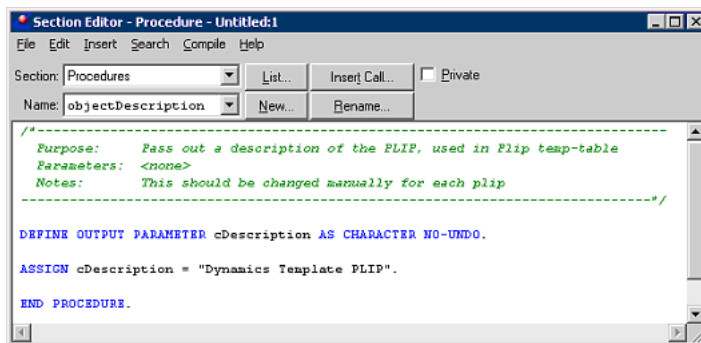


Figure 11–4: Standard `objectDescription` procedure

Now you can add your own internal procedures and invoke them through `launch.i`. The Session Manager and the `launchProcedure` method that is invoked by `launch.i` handle the following tasks in running your business logic:

- Identifying the AppServer partition where the external procedure is running or is to be run.
- Establishing a connection to the AppServer.
- Making the call to the specified internal procedure.
- Returning any OUTPUT parameters.

Your calling procedure can specify the logical name of the AppServer partition where the PLIP will run at run time, which maximizes the flexibility of your distributed application.

Keep in mind the following issues:

- You should never place any statements into your business logic procedures that would attempt any interaction with the user interface. Always expect that such procedures will be run in a separate Progress session on a separate machine from where the calling client application procedure is located.

- Because your procedure should run on an AppServer, you should not pass Progress buffer references to it, as the AppServer connection does not support this. Always pass data in the form of temp-tables, even if there is only one record involved.
- Make sure that your PLIP will not expect to have access to any other information that would only be available in the client session, such as SHARED or GLOBAL variable values, other buffer values, object properties, etc. Expect that it will run in a completely separate session from the client, and always be sure to test it that way.

PLIPs can be used for any kind of business logic that needs to be isolated from the client applications that call it. Using the `launch.i` include file, you can pass any set of parameters into the procedure. This can make it straightforward to incorporate existing logic procedures you might have written into Progress Dynamics, as long as they do not have any dependencies on the user interface or on client data.

11.3.2 Transaction Scoping within procedures

Progress provides sophisticated rules within the 4GL to manage database transactions for a variety of situations controlled by the definition of table buffers, blocks of executable code, and other factors. These rules have been effective in supporting complex procedures for character-mode applications running in a host-based or client/server environment, in which the user interface definition and the application logic are often heavily intertwined. However, such complex blocks of database-related logic can be problematic when used in a distributed environment such as that managed by Progress Dynamics. In general the best rule for managing transactions in a distributed environment is to keep the transaction as small in scope as possible. Also, to keep the definition of the transaction block as simple and unambiguous as possible, avoid unanticipated transaction scoping effects.

If you write logic to be executed within the context of one of the standard SDO or SBO validation procedures, then the database transaction is managed for you. The transaction block is started before the `beginTransactionValidate` procedures execute and ended after the `endTransactionValidate` procedures. You do not need to define a transaction block within those procedures.

If you are writing procedures for PLIPs to be executed outside these boundaries, however, you need to manage the transaction scope yourself. Again, keep the transaction definition simple and unambiguous to avoid errors. Here are some basic guidelines:

- Define the transaction block explicitly within a single internal procedure, and not in the Main Block of an external procedure with internal procedures where the updates are done.
- Define a buffer with a distinct name (such as b_ <tablename> or b_ <tablename> as the SDO logic procedure uses) for the database table to be updated, within the internal procedure where the transaction block is defined.
- Ensure that records are strong scoped using a DO FOR block for the table buffer.
- Scope transactions explicitly using the TRANSACTION keyword.
- Always use an explicit record -locking keyword on any statement that references the database, so that the behavior of the code is clear and does not rely on defaults.
- Avoid SHARE-LOCKS wherever possible, as these do not translate well to other databases supported by Progress DataServers. Read the documentation and white papers on DataServers if you need to familiarize yourself with other guidelines for writing code that ports effectively to DataServers.
- Define the UNDO and LEAVE qualifiers of the transaction block explicitly.
- Use the NO-ERROR qualifier on database statements that might fail. Check the ERROR-STATUS or the record AVAILABLE state and use the Progress Dynamics messaging support to generate a meaningful application message. Write code to react to the error appropriately, rather than relying on default behavior.

The following abbreviated example adapted from the framework code illustrates some of these points:

```
PROCEDURE updateProc:
DEFINE INPUT PARAMETER pcObjectName AS CHARACTER NO-UNDO.
DEFINE OUTPUT PARAMETER pcErrorText AS CHARACTER NO-UNDO.
DEFINE BUFFER bWizardTbl FOR WizardTbl.

tran-block:
DO FOR bWizardTbl TRANSACTION ON ERROR UNDO tran-block, LEAVE tran-block:
  FIND FIRST bWizardTbl EXCLUSIVE-LOCK WHERE NO-ERROR.
  IF NOT AVAILABLE bWizardTbl THEN
    DO:
      ASSIGN pcErrorText = {aferrortxt.i 'RY' '2' '?' '?' pcObjectName
        "'Browser'"};.
      UNDO tran-block, LEAVE tran-block.
    END. /* END NOT AVAILABLE bWizardTbl */
  . . .
END. /* END tran-block TRANSACTION */
RETURN.
END PROCEDURE.
```

11.3.3 Using PLIPs versus SDO logic procedures

A basic question in the mind of any developer will be when to use which mechanism for application logic. [Chapter 10, “Building Basic Business Logic in a Progress Dynamics Application,”](#) introduces you to the logic procedure for SDOs, which is itself a PLIP. The [“Transaction Scoping within procedures”](#) section discusses the use of the PLIP for other kinds of logic. Absolute guidelines are difficult to arrive at, since they depend on the style and structure of a particular application and on the extent to which existing application logic is being repackaged for use within a Progress Dynamics application. Nonetheless, it is important to make some general recommendations.

First, either put logic directly related to the update of a table into the logic procedure for its SDO or access it from there. Wherever possible, use the standard validation procedure hooks for this purpose. The phrase “access it from there” is to remind you that the principle of “putting logic in the SDO” does not at all mean that the code needs to be literally part of the SDO source procedure, or for that matter, its new logic procedure. It means that those entry points are a useful way to organize access to logic. If you have existing procedures that do largely the same work in the context of an older application, then by all means reuse them. Call them from appropriate entry points in the SDO, and the logic will happen at the right time.

The SDO logic procedure itself is a PLIP. This means that it can be run and its logic accessed independently of the SDO with which it is associated; this is part of the reason for creating it and maintaining it separately. If you need to access some of that logic from the client or from elsewhere on the server, you can do so using the `launch.i` mechanism. Earlier this chapter recommended against using the technique of obtaining the AppServer handle through the `getASHandle` function and running SDO procedures directly in that way. Instead, in the same situation, the client code can simply use `launch.i` to invoke the logic through the Session Manager. In this way, the Session Manager will take care of locating the server-side procedure (which might already be running), run the right internal procedure, return the results, and remove the connection.

A logic procedure PLIP can certainly contain procedures other than those that are already recognized as standard validation hooks, if they pertain to the SDO the procedure supports, just as an SDO today can contain additional logic procedures. If you have business logic that is broader in its scope than a single SDO, that can and should go into a PLIP of its own, which can be located on an AppServer and run as needed from any client. There is no absolute rule for the right granularity for such procedures, any more than for Progress procedures in general. Sensible guidelines apply. Internal procedures that are directly related should be grouped into a single PLIP as long as the size does not get out of hand. It will normally be necessary to start a PLIP for each access from a client. Therefore, you should avoid very large procedures when you can easily break them up. Also, if the contents of a large procedure are not directly related to one another, you might find one day that they really belong on different AppServers, which would force you to break up the procedure. In general, make your PLIPs as small as is reasonable to keep essential logic together, but no larger.

In summary:

- Use the SDO and SBO entry points (defined for you) for logic that will be executed at some point in the update process for a record or a set of related records. Put validation logic for an SDO into its logic procedure or run it from the appropriate SDO logic procedure entry point.
- Avoid making extra calls from an SDO client to an SDO on the server. If you must access logic in the server from some point in the client application, use `launch.i` to let the Session Manager control the connection for you. You can use `launch.i` to access the logic procedure of an SDO as well as any other PLIP.
- Package logic that does not fit neatly into the validation hooks of the SDO or SBO into separate external procedures (.p's) as PLIPs, using the Structured PLIP template as the starting point. This logic can be run from SDOs and SBOs, from the client application, or from other procedures on the server.

- Put related entry points into the same PLIP, but keep the size of the overall procedures modest so that there is minimal overhead in starting the procedures.
- If you have done any work to create business logic procedures that are independent of the user interface of your existing application, you might be able to repackage that logic into PLIPs with minimal effort. If you recast existing procedures using the PLIP template, and create internal procedures that correspond to existing internal or external procedure names in your application logic, you will be setting your new application up to manage such logic effectively through the Session Manager.

11.4 SBO overview

As a container of related SmartDataObjects (SDO), the SmartBusinessObject (SBO) manages a set of SDOs as a single database transaction and provides a place to express business logic that has access to the data of the entire transaction. In addition, the SBO provides optimized *fetch* and *commit* methods for the entire dataset across an AppServer connection using a single request. The SBO is the primary object in ADM and Progress Dynamics applications providing transaction control over multiple table updates. It also supports custom programming via data logic procedures.

Progress Dynamics also provides a dynamic SBO, which you can build in the Container Builder using both static and dynamic SDOs as data sources. Dynamic SBOs are only supported for GUI clients.

NOTE: While a dynamic SDO can act as a data source for a static SBO, the dynamic SDO cannot be contained in a static SBO.

The class **DynSBO** (a subclass of SBO) implements the dynamic SBO. The dynamic SBO moves the procedures and functions of a static SBO into the class super procedure. Like the dynamic SDO, the dynamic SBO supports a data logic procedure. The data logic procedure provides a place for your custom business logic, including hooks for validation code.

11.4.1 Container Builder support

The dynamic SBO is a dynamic container, and is therefore created in the Container Builder. The Container Builder also supports:

- Specifying data links between contained SDOs.
- Specifying the SBO as a data source for a visual object.

- Generating an optional data logic procedure.
- Editing a data logic procedure.

NOTE: If a visual object is an update source of an SBO, the `UpdateTargetNames` property of the visual object must be set to a comma-delimited list of all the SDO names contained in the SBO that will be affected by an Update operation. Note that you must carefully manage `UpdateTargetNames` and `DataSourceNames` if you make any changes in the way data flows between SDOs, SBOs, and viewers. After making any changes like adding or deleting an SDO/SBO or converting between SDOs and SBOs, check these properties in all related data objects to make sure they match your intent.

11.4.2 Viewers

The AppBuilder allows use of both static and dynamic SBOs when you select fields for dynamic viewers. The AppBuilder will create `DataField` instances for the viewer when it is saved. `DataField` names for SBOs include the name of the SDO that is the data source for the field and use this format:

`<SdoInstanceName>.<FieldName>`

As an exception, and to promote reuse, if a viewer only contains SBO `DataFields` from a single SDO, then the SDO name does not need to prefix the field names.

NOTE: To generate a viewer for a SBO, use AppBuilder not the Object Generator.

11.4.3 Data logic procedure

Dynamics uses a template for data logic procedures that support SBOs. Use the AppBuilder's New dialog box to access the template. The only architectural difference between static SBO and dynamic SBO data logic procedures is that static SBOs support custom logic hooks either directly in the object (`<SboName>.w`) or in a data logic procedure, but not both. Dynamic SBOs can only support custom logic in their data logic procedure. Only the major four custom hooks are supported in the SBO data logic procedure: Pre/Begin/End/Post `TransactionValidate`.

Like the SDO, the dynamic SBO data logic procedure contains static definitions of all SDO temp tables. This feature allows you to express custom business logic in static 4GL code.

The `DataLogicProcedure` attribute is part of the SBO and `DynSBO` classes.

NOTE: If custom 4GL code exists in the static SBO, you must move it manually to the new data logic procedure for the SBO.

11.4.4 API

The SBO supports the standard hooks for custom code structured around a commit operation. The SBO also helps coordinate the execution of custom code hooks for the contained SDOs.

An ADM2 object, `dynsbo.w`, represents run-time instances of DynSBOs.

The dynamic SBO supports the ADM2 API used for static SBOs. In addition, the SBO supports the following SBO Properties:

- **LastCommitErrorType** — This SDO property is included in both static and dynamic SBOs. It contains a semicolon-delimited list of the corresponding SDO property values.
- **LastCommitErrorKeys** — This SDO property is included in both static and dynamic SBOs. For SDOs, it is a comma-delimited list of the key values of the records that failed to commit. The `KeyFields` property of the SDO holds the key field names. For SBOs, the property will hold a semicolon-delimited list of corresponding SDO property value.
- **UpdateData** — This is the SBO version of an existing SDO function. The parameters are identical. The SBO version adds support for column names that are qualified by the appropriate SDO name.
- **DeleteData** — This is the SBO version of an existing SDO function. The parameters are identical. The SBO version adds support for column names that are qualified by the appropriate SDO name.
- **CreateData** — This is the SBO version of an existing SDO function. The parameters are identical. The SBO version adds support for column names that are qualified by the appropriate SDO name.

11.5 Building and using SmartBusinessObjects

The Progress SmartBusinessObject (SBO) is a container object for multiple SDOs. It can express complex business logic and data definitions that cannot be expressed in a single SDO, since an SDO is limited to a single database query (either a single database table or a straightforward join of tables). Its fundamental purpose is to allow a single update transaction on the server to span multiple SDOs.

The SBO is a nonvisual container for SDOs. Each SDO has its own RowObject definition and matching database query exactly as it would when used on its own. In addition, each SDO can still have its own update logic and separate logic procedure, as described in [Chapter 10, “Building Basic Business Logic in a Progress Dynamics Application.”](#) The value of the SBO is that it allows more complex data to be managed and more complex logic to be expressed. Data from multiple SDOs can be coordinated, which could never be expressed in a single join. (A single join might place it only in an awkward nested one-to-many join that would be unusable for data management.) Business rules can be written for an SBO on top of the business logic in each SDO. The SBO logic can also look at updated rows in each of the contained SDOs.

Because they are also nonvisual SmartObjects, you can add SBOs to a static or dynamic window just as you can add SDOs. You can then link User Interface objects to the SBO as you would with an SDO. That is, the SBO can be a Data-Source and an Update-Target for Viewers and Browsers, and a Commit-Target and a Navigation-Target for a SmartToolbar or Progress SmartPanels™. The SBO brokers these data connections between outside objects and the SDOs inside itself. In some cases this is accomplished automatically, based on the signatures of the objects (by matching fields displayed in a visual object with fields supplied by each SDO). In other cases, the other client objects have new properties to allow them to specify the contained SDO with which they should be associated. This section describes how to add SBOs to Progress Dynamics windows and how to set these additional properties.

In general, the SBO is intended to be used as a kind of super SDO, reproducing most of the behavior, API, and properties of the SDO but providing the additional ability to deal with complex datasets.

11.5.1 Guidelines for using SBOs

The SBO has three fundamental purposes:

- To let you define a single database transaction for a complex data set, including data from multiple tables all updated on the client in a single operation, either in a one-to-one or master-detail relationship.
- To provide a place where you can define business logic that must examine data from all of those data sets at the same time.
- Dynamics supports dynamic SBOs. Dynamic SBOs support data logic procedures like SDOs.
- When using the SBO as a data source, the ForeignFields property must be qualified with the name of the parent SDO.

It is important to note that in most cases you should only use an SBO if records from all the tables are added or updated on the client and returned to the server together. If a user update to a single table on the client results in executing business logic that updates other related tables on the server, this does not require using an SBO.

For example, if your application is browsing data in a master-detail relationship such as Orders and their OrderLines, but the two tables do not need to be updated in the same transaction, then you can and should continue to use multiple SDOs without the SBO container just as you would in earlier Version 9 releases. You can make the master SDO the Data-Source for the detail SDO and define the foreign key values that relate them. The SDOs will automatically be kept in sync as the master is repositioned, and users can update one SDO or the other. An update to either table from the client can execute whatever business logic it needs to on the server, including updates to any number of other related tables.

However, if you really need to allow users to make changes to both master and detail on the client (even potentially multiple levels of detail) in the same transaction and to commit all of those changes at once, then you can use the SBO to represent the data relationships. Likewise, if you need to define business logic that looks at changes to both master and detail at the same time, then you can also use the SBO.

It is entirely likely that your application will have places where you should use an SDO directly, and places where you should use that same SDO inside an SBO. For example, if at one point in your application the user can browser Orders, you should just use the Order SDO. If the application window just allows the Order to be updated, then you should also use the SDO by itself. If you want to show OrderLines of the Order but not update them in the same transaction, then the Order SDO should be the Data-Source for an OrderLine SDO. If the application window allows one or more OrderLines to be updated but does not allow updates to the Order at the same time, then you should use the OrderLine SDO. If you need to allow multiple OrderLines to be changed together, then you can use the OrderLine SDO together with the Commit/Undo band of a SmartToolbar.

But if your application logic requires that an Order and its OrderLines be added or updated in the same database transaction, and that business logic must control and validate those compound updates, then you can use a SmartBusinessObject that contains the Order SDO (as Data-Source) and the OrderLine SDO (as Data-Target). Each SDO can continue to have its own business logic, which affects updates to that single object. Any additional business logic written in the SBO will be executed in addition to the logic in each SDO.

11.5.2 SBO files and properties

The SBO is a nonvisual Progress SmartContainer™ object, with a ContainerType property of VIRTUAL. A static SBO is based on a template file named `ry\obj\rysttasboo.w`, a variation of the standard Version 9 SBO template `src\adm2\template\sbo.w`, and it has its own New item in the AppBuilder New menu list and its own button in the AppBuilder.

NOTE: They differ primarily in details of the wizard pages.

It has defined ADM-SUPPORTED-LINKS of Commit-Target, Data-Source, Data-Target, Update-Target, and Navigation-Target. Its Procedure-Type (which becomes the ObjectType ADM property) is SmartBusinessObject.

The SBO has these properties also found in the SDO:

- **AppService** — (CHARACTER) Application Service (Partition) name. If this property is blank, then the property will be written to the Repository with the default value. The default value for AppService value is the same as for all data objects: “ASTRA”. If you need a blank value here, edit the Repository with the ROM tool.
- **ASDivision** — (CHARACTER) Client, Server, or blank.
- **ASHandle** — (HANDLE) Handle of the SBO procedure running on the AppServer.
- **AutoCommit** — (LOGICAL) True if each Save should be immediately committed (which is the default if there is no Commit-Source) or False if changes should be accumulated in the Update temp-tables until an explicit Commit is done (which is the default if there is a Commit-Source). If the SBO has multiple SDOs, unsaved changes in all SDOs are saved.
- **DataColumns** — (CHARACTER) Comma-separated list of all columns in contained SDOs; this differs from the SDO format in that it is a single list where each column name is qualified by its SDO’s ObjectName.
- **ForeignFields** — (CHARACTER) Mapping of Data-Source fields to MasterDataObject fields if there is a Data-Source for this SBO.
- **ForeignValues** — (CHARACTER) List of the current values of ForeignFields (stored as a CHR(1)-delimited list of field values).
- **UpdateableColumns** — (CHARACTER) Comma-separated list of all updateable columns from all contained SDOs; this differs from the SDO format in that each column name is qualified by its SDO’s ObjectName.

- **RowObjectState** — (CHARACTER) NoUpdates or RowUpdated. Determines whether Commit-Source buttons are enabled.
- **CommitSource, CommitSourceEvents, NavigationSource, NavigationSourceEvents, DataTargetEvents** — To support those links.

The SBO has these additional ADM properties not found in the SDO:

- **MasterDataObject** — (HANDLE) Identifies the Master SDO or top-level SDO contained in the SBO.
- **ObjectMapping** — (CHARACTER) Maps the handles of client objects, including Browsers, Viewers, and Toolbars, to the procedure handles of the SDOs they are associated with. This can include mapping Browsers to the SDO whose query they are browsing, Navigation-Sources to the SDOs they are navigating, and Viewers to the SDOs whose fields they are displaying. The format of this property is a comma-separated list of pairs, with the first entry in each pair being the client object handle and the second entry being the SDO handle. Because an SDO can be associated with more than one other object, there might be duplications in the list. This property is used, for example, to determine which object to pass a queryPosition event on to. This property is intended for internal use, though it is possible that application code might, under some special circumstances, want to make use of it.
- **ContainedDataColumns** — (CHARACTER) A list of all of the DataColumnns from all contained SDOs. Column names are not qualified and are in a set of comma-separated lists, one for each contained SDO, with the lists being further delimited by semicolons. This property is intended for internal use.
- **ContainedDataObjects** — (CHARACTER) A comma-separated list of the handles of the SDOs contained in the SBO. This list, and others that list contained objects or their columns, are always kept in matching order (so that, for example, the nth entry in ContainedDataObjects is the handle of the nth entry in DataObjectNames; also, the nth sublist of column names in the ContainedDataColumns property is for the nth entry in ContainedDataObjects, and the nth list in ContainedDataColumns).
- **DataObjectNames** — (CHARACTER) The ObjectNames of its contained SDOs as a comma-separated list.
- **DataObjectOrdering** — (CHARACTER) This property, intended for internal use only, maps the order of the contained SDOs as defined by the AppBuilder-generated code to the order assigned either by organizing the objects in top-down Data link order (which happens during initialization if no other order has been assigned) or by using the user-defined update order as defined in the SBO property dialog box.

- **UpdateOrder** (CHARACTER) — List of the contained SDOs in the order that an update operation will process them. It is user-settable at the SBO instance level. It can be set programatically or it can be set manually using the Instance Properties dialog box.

In addition, there is a `getDataHandle` function that returns the handle of the SDO query matching a Browser that makes the request. There is also a `getQueryPosition` function, which returns the current `QueryPosition` setting for the contained SDO mapped to the requesting object.

Also, the following new `SmartObject` property is defined for all `SmartObjects`, and is used by SBOs in particular:

- **ObjectName** — (CHARACTER) The logical name of the `SmartObject`. By default it is the simple filename of the `SmartObject` procedure, with no leading path and no file type extension. This property can be used to give distinctive and meaningful names to the SDOs contained in an SBO, and is set in the instance property dialog box of the SDO when placed in an SBO. This name becomes the name of the update temp-table (defined within the SDO as `RowObjUpd`) when referenced in the SBO. Giving a distinctive name to each SDO makes it easier to write business logic in the SBO that refers to all of the contained SDO's updates.

The SBO has as its basic include file the file `src\adm2\sbo.i`. This includes the property include file `src\adm2\sboPROP.i` and prototype file `src\adm2\sboPRTO.i`, along with the standard custom include files as well (using `sbo` as the base include filename).

The SBO starts the super procedure `src\adm2\sbo.p`. Procedures and functions described below are implemented in `sbo.p` unless otherwise noted.

11.5.3 Data relationships in the SBO

In a `SmartContainer`, you can create a new SBO in the `AppBuilder` and drop one or more SDOs selected from the Palette. The SBO is only a static object in Progress Dynamics Version 1, so you always create it and add SDOs to it in the `AppBuilder`. The data links you make between SDOs in the SBO at design time determine the default order of initialization and update of the objects, but none of the objects are visualized when you run the application. Management of the SDOs by the SBO is done through the Container link, created automatically the same as for `SmartWindows` and `Progress SmartFrames™`. So all of the SDOs inside an SBO are its Container-Targets.

If there are multiple SDOs in the SBO, you must connect them to each other by Data links. You must define one of the SDOs as the Master SDO. This SDO has no Data-Source. You must link all the other SDOs to it, directly or indirectly, in a Data link hierarchy, as shown in the example in [Figure 11–5](#).

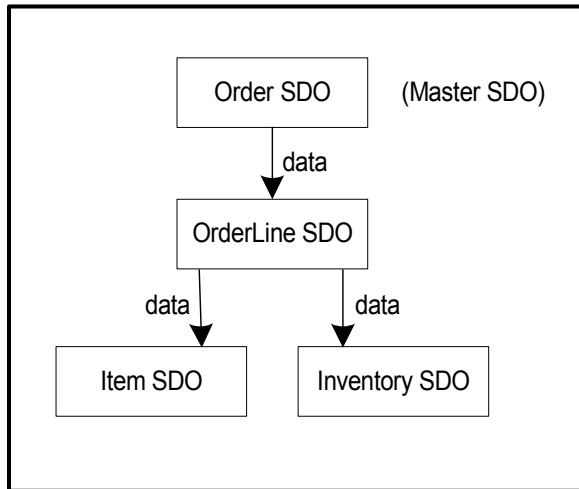


Figure 11–5: Example data link hierarchy

The SBO property `ContainedDataObjects` holds a list of all of the SDO procedure handles. The order of these handles is top-down. In the case where multiple SDOs are Data-Targets of a single parent SDO, the order will default arbitrarily. The order can be adjusted in the SBO Instance Property dialog box, so that updates to one SDO, which are dependent on updates to another (using a newly generated key value to assign to a field in a dependent SDO, for example), can be assured to happen in the correct sequence. The user adjusted order will be saved in the `UpdateOrder` property.

11.5.4 Brokering SDO data in an SBO

The SBO acts as an intermediary for all data in its SDOs. It serves as a Data-Source for client-side SmartObjects requesting a variety of data that it can provide. When you use it in another container such as a SmartWindow, you can link visual objects such as Browsers and Viewers to it. If you make a Data link from the SBO to such a Data-Target, then at run time the SBO examines the Signature property of the Data-Target and compares it to the Signature of the SDOs. The `ContainedDataColumns` SBO property, a delimited list of all DataColumns in the SDOs, is initialized at startup time by the SBO to assist in this. At run time the client-side SBO is responsible for brokering data to and from the visual objects that are linked to it.

In the case of Browsers, the SBO provides the Browser with the handle of the SDO query it needs to browser (using the `getDataHandle` function). The `ObjectMapping` SBO property is built up as Browsers are connected to the SBO at run time, to keep track of the SDO queries the Browsers are attached to.

During the initialization of the SBO, other client objects linked to the SBO are added to the `ObjectMapping` property. A `Viewer` is mapped to the SDOs that provide the fields it displays and updates. Likewise, the SBO matches any Browsers that are `Data-Targets` of the SBO with the appropriate SDO's temp-table based on matching field/column names. A `Navigation-Source` (Toolbar band) can specify in its property dialog box which SDO should be the `Navigation-Target` for its event messages. This SDO is also added to the `ObjectMapping` property.

`Navigation-Sources` must also specify which SDO should respond to the `Navigation` events. The default is the `MasterDataObject`, but if this is not the SDO you want, then you must choose the SDO `ObjectName` in the `Instance Property` dialog box of the `SmartToolbar`, as shown in [Figure 11–6](#). Access this dialog box from the `Container Builder` by choosing the `Properties` button, or from a static `SmartWindow` as you would any property sheet.

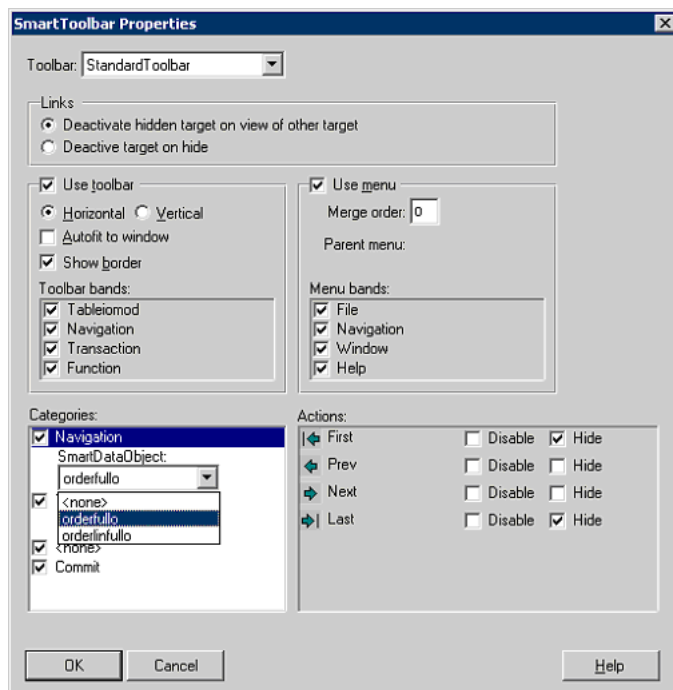


Figure 11–6: SmartToolbar properties dialog box

In the case of Viewers and other objects that request and display individual fields from the current row, the SBO provides those values by requesting them from one or more contained SDOs and passing them on. The API for these calls is the same as in SDO connections (including the published `dataAvailable` event and the `colValues` function). For example, an SBO containing a Customer SDO and an Order SDO can feed data to a Customer SmartDataViewer showing the selected Customer, and also to an Order Browser showing all retrieved Orders of the currently selected Customer. You can also build a SmartDataViewer against the SBO itself rather than against one of the SDOs. It can request fields from more than one contained SDO. If you build a Viewer against an SBO, the Viewer wizard displays columns from all contained SDOs, qualified by the SDO `ObjectName`, to allow the user to choose fields from multiple SDOs. When the Viewer is built against an SBO in this way, each frame field in the Viewer itself will have its SDO's `ObjectName` as a table-name qualifier (rather than `RowObject`), to allow the SBO to determine which SDO should supply the requested column values. Otherwise, if a Viewer is used that you built against one of the contained SDOs, its fieldnames have the standard `RowObject` qualifier, and in that case the SBO matches the requested column list against its `ContainedDataColumns` to identify the matching field (the first match is taken).

11.5.5 Data management in the SBO

The SBO can run either self-contained on a client with a direct database connection or divided between client and AppServer. Most of this section describes the divided case, because it is the more complex. In this case, the SBO has a client proxy just as an SDO does, with the same `_c1` name extension and the same use of the DB-REQUIRED preprocessors. The `_c1` file is generated on Save from the AppBuilder automatically, the same as for an SDO.

The client-side SBO proxy contains the same logic for choosing the appropriate SDO file to run as is used by other Container objects. That is, if the necessary databases used by the SDO are not connected, the SmartContainer (in the procedure `constructObject`) will search for and run the `_c1` proxy file for the SDO. If the required databases are not connected and the SDO proxy file is not found, an error will result.

When the client application is run, the client SBO runs as a client proxy and runs a server-side copy of itself. It supports the same AppServer properties and supporting code as the SDO to accomplish this. The server-side object in turn starts the SDOs, which in the two-layer model now have their own database connection. The client SBO proxy runs each contained SDO. All data is passed from server to client and back again through the SBO connection, not directly from the SDOs. When the client application is run with a state-aware AppServer, the client SBO initializes its temp-tables from communication with the server-side object through the SBO Container. When run with a stateless AppServer, the server-side SBO and SDO are not involved in the initialization of the client.

Figure 11–7 illustrates an example.

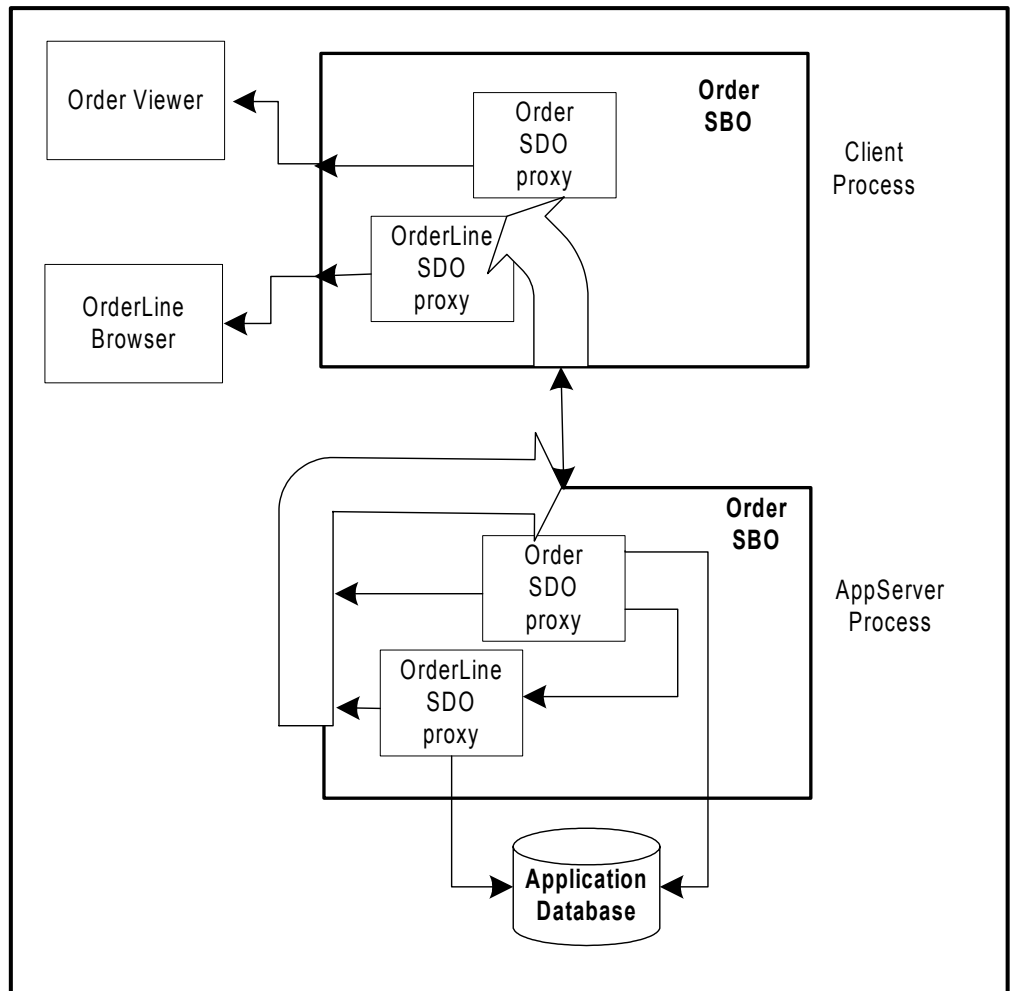


Figure 11–7: Data management example

When you are retrieving data from the server through an SBO, you can specify a WHERE clause for one or more of its SDOs. The functions `assignQuerySelection`, and `removeQuerySelection` are supported for SBOs just as for SDOs, and they let you modify the WHERE clause of any of the contained SDOs. The one restriction in the implementation of these functions for the SBO is that you must qualify column names with the `ObjectName` of the SDO, to let the SBO know which object to pass the where clause assignment.

NOTE: The `setQueryWhere` function is not supported; in any case, `addQueryWhere` and `assignQuerySelection` are more flexible ways to manipulate the Where clause.

The SBO retrieves data from the server by running the `fetchContainedData` procedure. (This is run internally and automatically; it is not expected that application code will run this procedure itself. The `openQuery` function is provided to do this from application code, to provide an interface similar to that of the SDO.) The `fetchContainedData` procedure collects the `QueryString` properties of its `ContainedDataObjects` and passes them to the server by running `serverFetchContainedData`. This server-side procedure resets the `QueryString` properties on the server, and the query for the top-level SDO specified is opened.

The `fetchContainedData` procedure gets back as OUTPUT parameters the TABLE-HANDLES of the `RowObject` tables and passes these on to the individual SDOs.

When the user changes the current row in an SDO with a Data-Target, all rows in dependent queries are flushed from the client and replaced by the rows matching the `ForeignField` values for the new master row (with the same checking for modified rows as for independent SDOs).

When the user changes individual rows, the rows are saved on the client (in their respective SDOs) until committed. When the user chooses the Commit button or the `CommitTransaction` event occurs in some other way, all updated rows from all updated SDOs are sent back to the server and validated there.

If scrolling or other repositioning moves to the end of the current data set for any SDO, and the entire result set has not been retrieved from the server, the next batch of rows is retrieved from the server for that SDO.

There are two basic procedures to handle this data transfer from server to client:

- **`fetchContainedData`** — This makes a call to its server-side counterpart, `serverFetchContainedData`, passing a delimited list of the `QueryString` properties of all the SDOs, and getting back a set of TABLE-HANDLE OUTPUT parameters for each SDO. These are the handles of the temp-tables of the SDOs contained in the SBO on the client side. In this way, all of the data from multiple SDOs can be requested and returned in a single call. This is different (and more efficient) than what happens with individual SDOs that are nested in a parent-child relationship. In that case, each SDO has its own connection to the server, and each makes its own independent request for data, with the `dataAvailable` events that cascade the key values from parent to child occurring on the client. With an SBO, all the data is transmitted through the single SBO connection, and the `dataAvailable` events occur on the server.

- **serverContainedSendRows** — Takes care of retrieving additional batches of rows in the case where more than one batch is required. In general, it is expected that this will occur only when the user is browsing the top-level SDO in the SBO. However, the mechanism will also work for dependent SDOs with enough rows that they cannot all be retrieved in a single batch. The `serverContainedSendRow` procedure is run from `clientSendRows` (the existing SDO procedure) when another batch of rows is needed. The `clientSendRows` procedure determines whether it is inside an SBO by checking the new SDO property `QueryContainer`, which is true if the SDO's Container is itself a `QueryObject`. In this case, the SDO will get the `ASHandle` property of the Container SBO and run `serverContainedSendRows` directly itself, using the SBO's `AppServer` connection.

11.5.6 Updates through the SBO

The business logic of the SBO is additive. That is, you can keep basic logic affecting only one table in each individual SDO as now. If you need to add validation or other logic at the level of the SBO, it will have access to the update temp-tables for all of its contained SDOs. This enables the SBO `pre/post/begin/endTransactionValidate` procedures to use the `RowObjUpd` temp-tables from the different SDOs, map them to meaningful local temp-table names, and write normal (static) Progress 4GL logic against those tables. As described earlier, you can give a logical name to each SDO you drop onto it. This is the `ObjectName` property, which is initialized to the simple filename of the SDO, without the extension. You can change this Instance Property as needed to give the SDO a distinctive name within this SBO. (This renaming might be necessary, for example, if the same SDO appears more than once in the tree, which occurs frequently in real-world applications. An Item SDO, for example, might need to be named `Item` in one instance and `AlternateItem` in another instance, if that is how it is used within the SBO.) This property is then attached to a reference to the SDO temp-table include file. For example, the `Order/OrderLine` SBO can use the default SDO `ObjectNames` that will result in AppBuilder code generation to name its update tables `orderfull0` and `orderlinfull0`.

Transaction logic in the SBO

The transaction logic is also parallel to and augments the logic of the SDO. The SBO defines an encompassing transaction within which the data from individual SDOs is committed to the database. As with SDOs, if the `AutoCommit` property is true (which will be the default if there is no `Commit-Source`), then each individual Save operation results in one set of changes being committed to the database.

NOTE: If the Save is against a `SmartDataViewer` that updates fields from multiple SDOs, then it is possible that a single Save operation could update more than one database record.

If the `AutoCommit` property is false (which is the default if there is a `Commit-Source`), then the SBO can accumulate multiple updates on the client to be sent back to the server together. This would be considered the norm for SBO usage, since SBOs are primarily intended for use in situations where rows from multiple different SDOs need to be committed as part of the same transaction. The exception to this is where two or more records related on a one-to-one basis are updated through an SBO. In this case, you do not need the `Commit` operation, and you can set `AutoCommit` to `True`. Each `Save` request invokes `submitRow` in the appropriate SDO.

These changes are stored on the client until a `Commit` occurs. When `Commit` is chosen, or if `AutoCommit` is `True`, the following sequence of events takes place:

1. The `CommitTransaction` procedure is run in the client SBO. This procedure collects all the `RowObjUpd` tables from any modified SDOs on the client and passes them to the server-side SBO. It runs `serverCommitTransaction` in the `AppServer` SBO, which in turn copies the tables to their respective SDOs.
2. The server-side SBO executes code similar to the following block of pseudo-code. An error detected at any point results in the `Commit` being rolled back and the error message being returned to the client:

```
RUN preTransactionValidate IN THIS-PROCEDURE NO-ERROR.
FOR EACH SDO:
  IF preTransactionValidate is defined for it THEN
    RUN pushTableAndValidate in the SDO
      (INPUT Pre, INPUT-OUTPUT hROUTable).
  END.
SBO-TRANS:
DO TRANSACTION:
  RUN beginTransactionValidate IN THIS-PROCEDURE NO-ERROR.
  /* ServerCommit will run beginTransactionValidate
    in each SDO, commit the changes, and run
    endTransactionValidate in each SDO.
    Any errors will undo the whole SBO-TRANS. */
  FOR EACH SDO:
    ServerCommit().
  END.
  RUN endTransactionValidate IN THIS-PROCEDURE NO-ERROR.
END. /* END TRANSACTION */

FOR EACH SDO:
  IF postTransactionValidate is defined for it THEN
    RUN pushTableAndValidate in the SDO
      (INPUT Post, INPUT-OUTPUT hROUTable).
  END.
RUN postTransactionValidate IN THIS-PROCEDURE NO-ERROR.
```


The `pushTableAndValidate` procedure referenced in this code tells the SDO being called whether this is pre- or post-transaction validation, and passes the SDO's `RowObjUpd` table to it. This runs `pre/postTransactionValidate` in the SDO, and then returns the `TEMP-TABLE` back to the SBO in case it has changed. When `Commit` is executed in the SDO, SDO code suppresses running `pre/postTransactionValidate` in the SDO if it is inside an SBO. It is run at the proper time from the SBO (to allow all `preTransactionValidates` to happen together, followed by the transaction, followed by all `postTransactionValidates`).

The `pre/begin/end/postTransactionValidate` procedure names are the same as the standard validation procedure hooks supported for SDOs. The SBO logic extends the logic of the SDOs without introducing new naming conventions. The additional validation logic entry points that are available for use with the Progress Dynamics SDO's logic procedure (`createPreTransValidate`, etc.) are not defined for SBOs.

This nested form allows maximum flexibility in the location of SBO logic and the coordination of that logic with SDO logic. The `pushTableAndValidate` procedure (implemented in `data.i` to allow it to use the static SDO temp-table as a parameter) assures that changes are automatically shared between the SBO and its SDOs. The cost of organizing the code in this way is not as great as it might seem at first. `pushTableAndValidate` will not be executed except where validation procedures exist in the individual SDOs. It is important to let you control the placement of logic at each stage (`pre/begin/end/post`), and also at the appropriate place within each stage. The SBO can add its own logic at the very beginning of all logic, at the beginning of the transaction, at the very end of the transaction, and at the very end of all the logic (after all the SDOs' `postTransactionValidate` procedures). Or a given SBO could completely replace the `serverCommitTransaction` procedure if necessary to control the logic in a special way. (Note that this procedure is defined in `sbo.i`, to receive the `RowObjUpd` temp-tables into the SBO's definition of them, so you cannot override it, only replace it.)

The most common form of business logic that spans SDOs, namely assigning key values to new records, is handled automatically by the SBO support code in the `serverCommitTransaction` procedure. If you are adding rows to both a parent and child SDO in the same transaction, then `ForeignField` values are retrieved from the parent record after it has been written to the database and assigned to each of the child records. This is similar to the way in which key values are assigned to new records added through an SDO with a Data-Source. The difference is that the supporting code in the SBO allows both parent and child records to be added in the same transaction. All newly assigned values will be passed back to the client for display, as with any other changes made on the server-side.

11.5.7 Other SBO issues

This section describes the following SBO issues:

- The SBO as Navigation-Target
- Locating a Contained Data Object
- SDO Programming Interface in the SBO
- The SBO as Data-Target

The SBO as navigation-target

An SBO can be a Navigation-Target. Since you might want to navigate more than one of the contained SDOs, it is necessary to allow the Navigation-Source (SmartToolbar) to specify which SDO is to be associated with this Navigation-Source without changing the existing interface between these objects. (See the [“Brokering SDO data in an SBO”](#) section for more information.) To accomplish this, there is a new property for the Panel class used by Toolbars called `NavigationTargetObject`, which can optionally be set to the `ObjectName` of an SDO. `NavigationTargetObject` is an Instance Property. When the Instance Property dialog box is run, this property is displayed as a drop-down list and enabled if the Navigation-Target is an SBO (that is, if it is an object that has a `ContainedDataObjects` property). Otherwise it is disabled. If it is not explicitly set, the default is to pass the Navigation link on to the `MasterDataObject`. Use this value to create an entry pair in the SBO `ObjectMapping` property associating the two objects, so that the SBO knows which SDO to pass Navigation events.

Locating a contained data object

Generally, you should consider the SBO to be reasonably opaque, so that it accepts requests from other objects and coordinates those with the contained SDOs. However, it is not possible to do this in every case, as that would unnecessarily complicate the API of the SBO. The SBO supports a function called `dataObjectHandle`, which takes an `ObjectName` as an input parameter and returns the procedure handle of that `SmartDataObject`. You can use this handle for a client object that needs to communicate directly with an SDO. The majority of the SDO programming interface, however, is duplicated in the SBO so that generally a client object can run the same routines in an SBO that it would in an SDO, and have the SBO pass the call on to the appropriate contained objects.

SDO programming interface in the SBO

Most of the common routines in the SDO are also supported with the same calling sequence in the SBO support code (in `sbo.p` or `sbo.i`). This allows application code to treat an SBO like a compound SDO in many ways. For example, a general restriction in some of the calls where column names are an input parameter, as described for `addQueryWhere` and `assignQuerySelection`, is that you must qualify column names by the SDO `ObjectName`. Generally, the SBO identifies the SDO associated with the caller by using the `ObjectMapping` property. It passes the call on to that object, or else uses property data defined within the SBO to satisfy the request. Here is a list of SDO internal procedures and functions duplicated in the SBO's API (in all cases, the calling sequence is the same):

The internal procedures that are duplicated are:

- `commitTransaction`, `undoTransaction`
- `dataAvailable`
- `fetchBatch`, `fetchFirst`, `fetchNext`, `fetchPrev`, `fetchLast`
- `initializeObject`

The functions that are duplicated are:

- `addQueryWhere`, `assignQuerySelection`
- `addRow`, `copyRow`
- `columnDataType`, `columnLabel`, `columnColumnLabel`, `columnExtent`, `columnFormat`, `columnHelp`, `columnInitial`, `columnColumnLabel`, `columnMandatory`, `columnModified`, `columnPrivateData`, `columnQuerySelection`, `columnReadOnly`, `columnStringValue`, `columnTable`, `columnValExp`, `columnValMsg`, `columnValue`, `columnWidth`, `columnDbColumn`
- `colValues` (can take either qualified or unqualified column names)
- `deleteRow`, `cancelRow`
- `openQuery`
- `submitRow`

The SBO as data-target

An SBO can be a Data-Target for an SDO just as another SDO can. It has the `ForeignFields` property, which lets you specify fields to use as foreign keys, and the `ForeignValues` property, which stores the current values for those fields. The `ForeignFields` will always be applied to the SBO's `MasterDataObject`. You cannot currently use the SBO as a Data-Source for another SBO.

11.6 Example: creating an SBO

This chapter has discussed two kinds of situations where using an SBO is appropriate:

- Where updates from the client span multiple tables in a parent-child relationship
- Where they are in a one-to-one relationship.

You should build an SBO to handle such updates only if both of the following are true:

- Fields from both (or all) the tables involved can be changed on the client
- These changes must be made to the database in a single transaction.

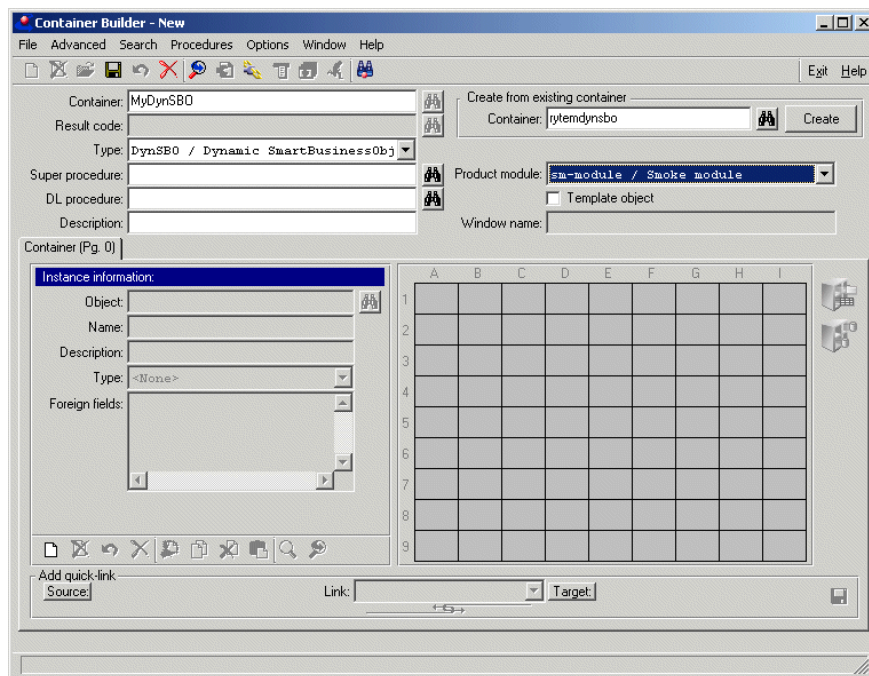
In the example you will build an SBO with two SDOs, for Orders and OrderLines of an Order.

11.6.1 Creating a dynamic SBO

You can create new dynamic SBOs in the Container Builder. To create a dynamic SBO:

- 1 ♦ Open the Container Builder.
- 2 ♦ Choose **File**→**New**.
- 3 ♦ Enter a name in the **Container** field.

- 4 ♦ From the **Type** combo, choose **DynSBO**:



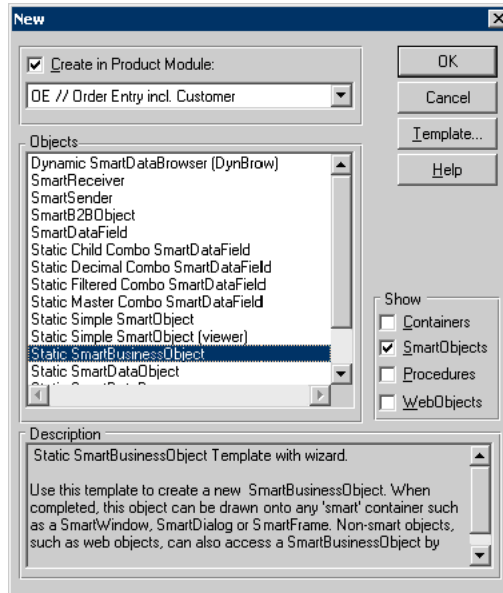
- 5 ♦ To create this DynSBo from the template that comes with the framework, click the **Lookup** button next to the **Create from existing container... Container** field and select **rytemdysbo**.
- 6 ♦ Select a **Product module**.
- 7 ♦ Click **Create**.
- 8 ♦ Click **Save**.

You can now access the property and dynamic property sheets of the dynamic SBO to finish your definition.

11.6.2 Creating a static SBO

Follow these steps to create an SBO:

- 1 ♦ From the AppBuilder main window, select **New→Static SmartBusinessObject**. The New dialog box appears:

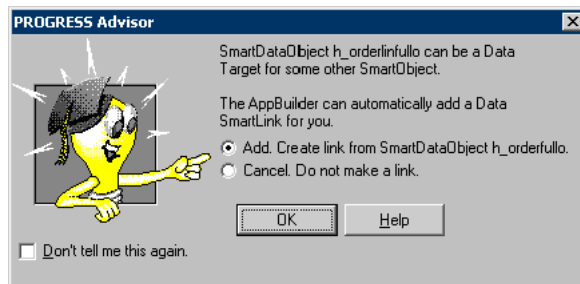


The SBO wizard prompts you for the same sort of documentation information as for other objects, and then leaves you in a nonvisual design window.

- 2 ♦ Select the SDOs that you need to group together from the AppBuilder palette. Drop them onto the SBO design window in their top-down or header-detail order. For this example, use the SDOs for the Order and OrderLine tables created by the Object Generator, `orderfullo.w`, and `orderlinfullo.w`:

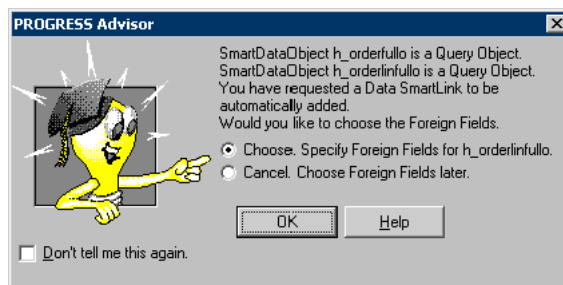


When you drop the second (and any succeeding) SDOs onto the SBO, the AppBuilder's Link Advisor prompts you to let it create a Data link between SDOs. Confirm that the link is appropriate, then accept the link as presented by the Progress Advisor dialog box:

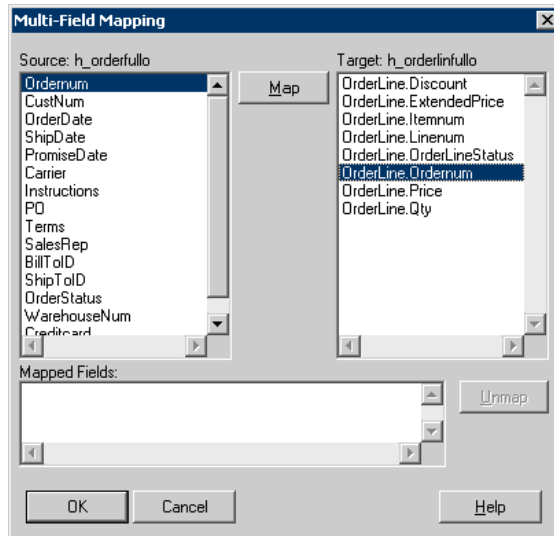


Alternatively, add links of your own after adding the SDOs. There must be a data link from the Order SDO to the OrderLine SDO in the example.

- 3 ♦ For each data link you are prompted for the foreign key fields to use to filter the query of each SDO based on data from the Data-Source:



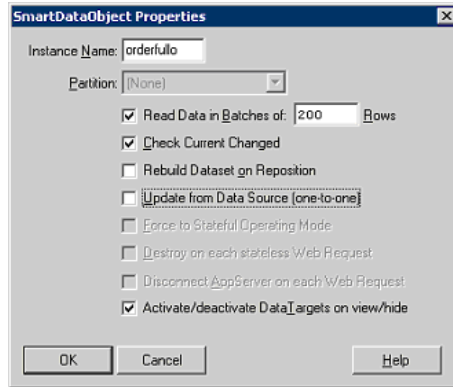
- 4 ♦ Choose OK. The Multi-Field Mapping dialog box appears:



- 5 ♦ Choose the appropriate fields from each SDO. In the example, the OrderNum field links the two SDOs.
- 6 ♦ Save the SBO and give it the name you used in the documentation page of the wizard. Name the example SBO `orderlinesbo.w`.

Because the Order SDO has no Data-Source, it is automatically made the MasterDataObject of the SBO in the example.

In the Instance Property dialog box for each SDO, you can change the ObjectName property if you wish, along with other properties such as RowsToBatch. (In this example, the ObjectName property is called *Instance Name* to distinguish it from the object name of the variable holding the handle of the SmartObject—used elsewhere in the AppBuilder windows.) It is not meaningful to set the Partition for the SDO. The Partition property of the SBO determines the AppServer connection, if any, so this field is disabled:



The Update from Data Sources check box is a property used for SDOs in SBOs. If you use the SDOs in the SBO in a one-to-one relationship where they all represent different parts of a single logic record (such as Customer Header, Address, and Detail Information tables for example), then this should be checked for all SDOs except the Master. This will allow them to be updated and all changes committed to the database together just by choosing the **Save** button in a Toolbar, rather than requiring a Save and a separate Commit. In other words, this toggle box tells the SBO to initiate an update from this SDO's Data-Source (the Master SDO), rather than saving its changes separately.

In the case of a master-detail transaction where there might be two or more detail records added or modified together with a single master record, the changes all need to be saved separately on the client and then committed together to the server. In this case, leave the Update from Data Source toggle box unchecked and include Commit and Undo buttons (the Transaction band available on the Standard Toolbar and other Toolbars) in your window.

Page 0 (the background page) and Page 1 are displayed when the window comes up, as shown in [Figure 11–8](#).

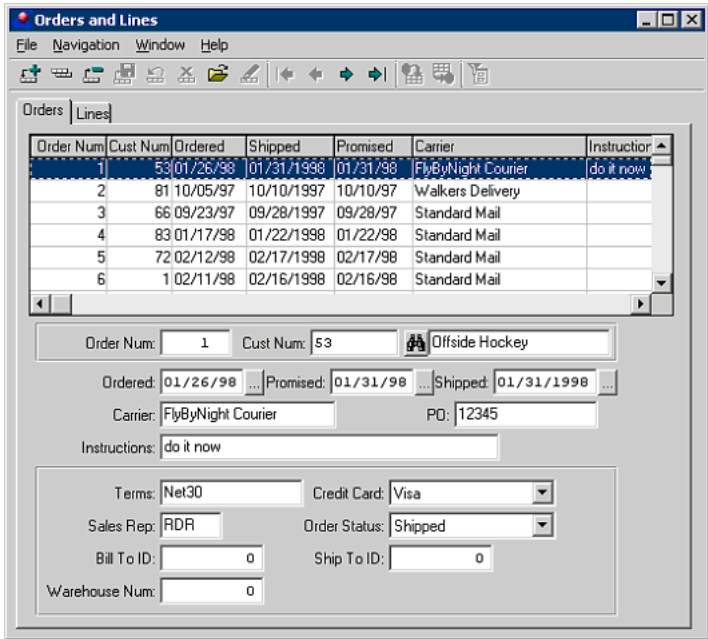
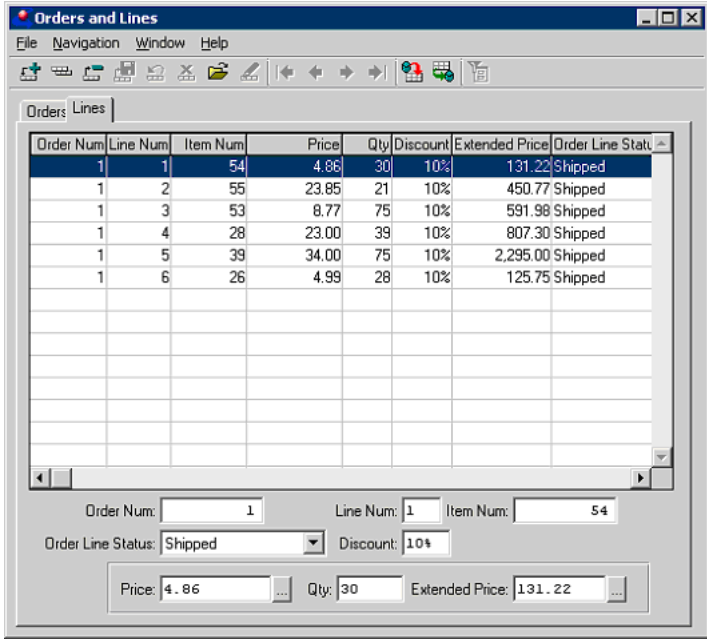


Figure 11–8: Example window – Pages 0 and 1

The SBO has retrieved the first batch of Orders through the Order SDO, and in the same operation, the OrderLines for the first Order. If you modify a field in the Order Viewer and Save that change, the Commit and Undo buttons are then enabled to let you send that change back to the server. Remember that when the Commit band is in your Toolbar and there is a Commit link from the Toolbar to a data object such as your SBO, all changes will be held in local temp-tables on the client until you choose **Commit**. If you choose **Undo**, all these changes are deleted.



11.7 Defining business logic for SBOs

The standard validation hooks for the SBO are basically the same as for the SDO: they have the names `preTransactionValidate`, `beginTransactionValidate`, `endTransactionValidate`, and `postTransactionValidate`. The `preTransactionValidate` procedure executes before the corresponding procedure in any contained SDO, and can modify values in SDO records that will be seen by the validation logic in the SDOs. The `beginTransactionValidate` procedure executes at the very beginning of the single database transaction that encompasses all the updates in the SBO and its SDOs. Likewise, the `endTransactionValidate` procedure executes at the very end.

The dynamics SBO also supports a custom logic procedure that works just like the dynamic SDO's custom logic procedure.

This first code example verifies that the client-side logic defined for an SDO will execute even when it is contained in an SBO. The Object Generator creates code for you in each SDO that verifies that all indexed fields have been entered. Because this code can be executed without returning to the server where the database is, it is placed into the client-side `rowObjectValidate` procedure:

```
/*-----
Purpose:  Procedure used to validate RowObject record client-side
Parameters: <none>
Notes:    Other mandatory checks left out of this example.
-----*/
DEFINE VARIABLE cMessageList AS CHARACTER NO-UNDO.
DEFINE VARIABLE cValueList AS CHARACTER NO-UNDO.

IF isFieldBlank(b_Order.SalesRep) THEN
  ASSIGN
    cMessageList = cMessageList +
      (IF NUM-ENTRIES(cMessageList,CHR(3)) > 0 THEN CHR(3) ELSE '':U) +
      {aferrotxt.i 'AF' '1' 'Order' 'SalesRep' "'Sales Rep'"}.
  ERROR-STATUS:ERROR = NO.
  RETURN cMessageList.
END PROCEDURE.
```

If you run the ordersbowin window, delete the SalesRep code from an Order, then choose **Save**, you immediately see the Progress Dynamics message shown in [Figure 11–10](#).

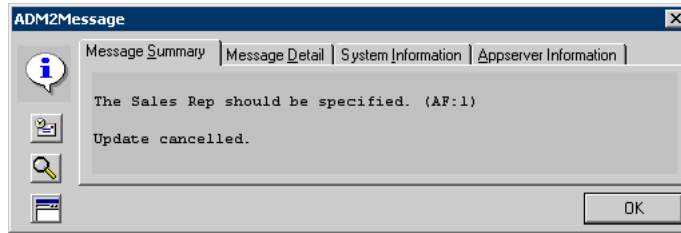


Figure 11–10: ADM2 message

Because the logic executes on the client, you see the error as soon as you choose **Save**, without having to do a Commit.

The next code example shows that the SDO's `preTransactionValidate` or `writePreTransValidate` logic will be executed from inside an SBO. Here is an excerpt from the `writePreTransValidate` procedure generated for the Order SDO by the Object Builder. It verifies that an updated `OrderNum` field does not match an existing `OrderNum` in the database:

```

/*-----
Purpose:  Procedure used to validate records server-side before the
          transaction scope upon write
Parameters: <none>
Notes:
-----*/
DEFINE VARIABLE cMessageList AS CHARACTER NO-UNDO.
DEFINE VARIABLE cValueList AS CHARACTER NO-UNDO.
IF NOT isCreate() AND CAN-FIND(FIRST Order
    WHERE Order.CustNum = b_Order.CustNum
    AND Order.Ordernum = b_Order.Ordernum
    AND ROWID(Order) <> TO-ROWID(ENTRY(1,b_Order.RowIDent))) THEN
DO:
    ASSIGN
        cValueList = STRING(b_Order.CustNum) + ', ' + STRING(b_Order.Ordernum)
        cMessageList = cMessageList +
            (IF NUM-ENTRIES(cMessageList,CHR(3)) > 0 THEN CHR(3) ELSE '':U) +
            {aferrotxt.i 'AF' '8' 'Order' ' ' 'CustNum, Ordernum, ' cValueList }.
END.

ERROR-STATUS:ERROR = NO.
RETURN cMessageList.
END PROCEDURE.

```

For example, if you run the SBO window and change OrderNum from 1 to 2, then choose **Save**, and then **Commit**, the error message shown in [Figure 11–11](#) appears.

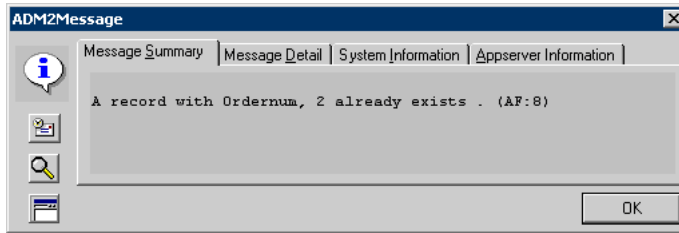


Figure 11–11: ADM2 error message

Because this logic must execute on the server side, you do not see the error when you first choose **Save**. At that time, the change is saved only locally on the client. Only when you choose **Commit** to send the change back to the server does the `writePreTransValidate` execute. The code formats the error message and returns it up the execution stack, where it is intercepted by the framework code and returned to the client for display.

The next code example shows the interaction between the SBO `preTransactionValidate` and its counterpart in the SDO. In this rather contrived bit of code, the procedure first checks that if the PO field is not blank, then it must begin with the letters 'PO.' If not, the code returns an error. Otherwise, if it is blank, it is replaced by the string `<none>`. In the SBO, each contained SDO's temp-table has, by default, the unqualified name of the SDO itself. You can change this value in the Instance Property dialog box for the SDOs within the SBO:

```
/*-----
Purpose:  Verify that a PO begins with the letters 'PO'.
         If it's blank then set it to <none>.
Parameters: <none>
Notes:
-----*/
FOR EACH orderfullo WHERE orderfullo.RowMod = "U":U:
  IF orderfullo.PO NE "" :U AND NOT orderfullo.PO BEGINS "PO":U THEN
    RETURN {aferrortxt.i '?' "'PO must begin with PO'" 'Order' '' 'PO'
            orderfullo.PO }.
  ELSE IF orderfullo.PO = "" :U THEN
    orderfullo.PO = "<none>":U.
END.
END PROCEDURE.
```

If you run the application again with this code, then save and commit any change to an Order record with a blank PO, the value is changed to `<none>`. This value is returned to the client and displayed after the change is committed.

The next code sample adds another check to the Order SDO's `writePreTransValidate` procedure. As described in [Chapter 10, "Building Basic Business Logic in a Progress Dynamics Application,"](#) if you use the create/write/delete validation procedures in the SDO logic procedure, the buffer name is the simple table name preceded by `b_`:

```
IF b_order.PO = "<none>":U THEN
  cMessageList = cMessageList + (IF NUM-ENTRIES(cMessageList,CHR(3)) > 0
    THEN CHR(3) ELSE '':U) +
    {aferrortxt.i '?' "'PO cannot equal <none>' " 'Order' ' ' 'PO' cValueList }.

```

This code verifies that the PO number field has not been set to the string `<none>`. If it has, then an error is reported. This example does not put this error into the Progress Dynamics message table as recommended you always do. For this reason, the message text is just inserted into the message formatting include file `aferrortxt.i`.

Because the SBO's `preTransactionValidate` procedure executes first, it changes a blank PO number to `'<none>'`, and passes this along to the SDO for further validation. Thus a blank **PO** is rejected with the error message that it cannot equal `<none>!`

The next block of code shows how the SBO can look at updates in more than one SDO, referring to each by its logical name. This logic totals the Quantity of any modified OrderLines of each Order record, which was itself modified. It puts that total into the Order record's **PO** field, so that the calculation can be seen back on the client when the transaction completes:

```
/* Additional code for SBO preTransactionValidate: */
DEFINE VARIABLE iQty AS INTEGER NO-UNDO.
/* If both an Order and OrderLines were modified, total the quantities
  of the changed OrderLines and stick that (in brackets) into the
  PO fld. */
FOR EACH orderfullo WHERE orderfullo.rowmod = 'u':
  FOR EACH orderlinfullo WHERE orderlinfullo.rowmod = 'u' AND
    orderlinfullo.ordernum = orderfullo.ordernum:
    iQty = iQty + orderlinfullo.Qty.
  END.
IF iQty NE 0 THEN
  orderfullo.PO = orderfullo.PO + " <" + STRING(iQty) + ">".
END.

```

The next example is a `beginTransactionValidate` procedure for the SBO. This is the place to put logic that needs to lock or modify other database records before the SDOs' updates are written back to the database. This simple example identifies each `OrderLine` where the `Price` field has been modified, finds the corresponding `Item` record in the database, and modifies that record's `Price` field to match. (This almost certainly is not what real-world business logic would want to do, but it serves the purpose of a demonstration.) The `Item` record is updated as part of the same transaction as the `OrderLine` changes:

```
PROCEDURE beginTransactionValidate:
/*-----
Purpose:   SBO logic to execute at the beginning of the transaction.
Parameters: <none>
-----*/
DEFINE BUFFER doline2 FOR orderlinfullo.
/* Set the price for the item record in the database equal to the
price in the OrderLine record in the SDO if it has been changed.
The Doline2 buffer allows us to compare the Before and After
versions of each modified OrderLine. */
FOR EACH orderlinfullo WHERE orderlinfullo.rowmod = 'u':
FIND doline2 WHERE doline2.rownum = orderlinfullo.rownum AND
doline2.rowmod = ''.
IF orderlinfullo.price NE doline2.price THEN
DO:
FIND ITEM OF orderlinfullo NO-ERROR.
IF NOT AVAILABLE(ITEM) THEN
RETURN "Item Record for OrderLine " +
STRING(orderlinfullo.ordernum)
+ " " + STRING(orderlinfullo.linenum) + " not found.".
ELSE ITEM.price = orderlinfullo.price.
END.
END.
END PROCEDURE.
```

NOTES:

- As an alternative to defining a second buffer for the Before version of a changed record, as the previous example did, if you only need to know which fields were modified, you can look at the `ChangedFields` field of the update table record. If the condition is true (the `Order` and the `Price` or `Qty` of at least one of its `OrderLines` have been changed), then all the `OrderLines` for the `Order` are re-read from the database, totaled, and the total placed into the `PO` field of the `Order`.
- This kind of logic has to take place at the end of the transaction, because both modified `OrderLines` and unmodified `OrderLines` need to be read. You want the total to reflect both updates that were just made within the transaction, and any other `OrderLines` that were also in the database but not modified in this transaction.

Finally, you define an `endTransactionValidate` procedure for the SBO. This is the place to put logic that needs to execute within the transaction block, but after all of the SDO updates have taken place. The procedure determines if an Order record has been changed, and if the Qty or Price field for at least one of its OrderLines has also changed.

In addition, because it is actually the `ExtendedPrice` that is being totaled, and the `ExtendedPrice` is calculated by a database trigger, this value will be available only at the end of the transaction, after the individual OrderLines have been written to the database.

To demonstrate returning an error from within the transaction, the following code example then checks the `CreditLimit` of the Customer record associated with the updated Order. It returns an error message if the `CreditLimit` has been exceeded by this new Order total. If this happens, the whole transaction is backed out, the error message will be returned to the client, and the user will be able to make whatever changes are necessary to be able to resubmit the updates successfully. Or the user can back out of all the changes to all SDOs by choosing the **Undo** button:

```
PROCEDURE endTransactionValidate:
/*-----
Purpose:   SBO logic to be executed at the end of the transaction.
Parameters: <none>
-----*/
DEFINE VARIABLE dPrice AS DECIMAL NO-UNDO.

FOR EACH orderfullo WHERE orderfullo.rowmod = 'u':
/* Stick the order total in the order record, if the order
and at least one of its OrderLines' price/qty was updated. */
FIND FIRST orderlinfullo WHERE orderlinfullo.rowmod = 'u' AND
(LOOKUP('Qty', orderlinfullo.ChangedFields) NE 0 OR
LOOKUP('Price', orderlinfullo.ChangedFields) NE 0) NO-ERROR.
IF AVAILABLE(orderlinfullo) THEN
DO:
FOR EACH OrderLine WHERE OrderLine.OrderNum = orderfullo.orderNum:
dPrice = dPrice + OrderLine.ExtendedPrice.
END.
FIND Order WHERE Order.Ordernum = orderfullo.orderNum.
FIND Customer WHERE Customer.CustNum = Order.CustNum.
IF dprice > Customer.CreditLimit THEN
RETURN "This one order total of " + STRING(dPrice) +
" exceeds the Customer Credit Limit!".
ELSE ASSIGN Order.PO = Order.PO + " [" + STRING(dPrice) + "]"
orderfullo.PO = Order.PO.
END.
END.
END PROCEDURE.
```

After making some changes to the database with this logic, the Order records will start to look rather strange. The values in angle brackets (<>) are the totals of OrderLine quantities modified within a transaction. The values in square brackets ([]) are the Order totals, that is, the total of all ExtendedPrice values for all OrderLines for an Order, whenever both the Order record and the Qty or Price fields of one or more of the OrderLines have been modified within a single transaction.

11.8 Conclusion

These examples show most of the important principles of defining business logic for SBOs:

- The SBO supports the same four server-side hooks for logic as the SDO:
 - `preTransactionValidate`, for validation checks or changes to be made to the updated records before the transaction starts
 - `beginTransactionValidate`, for changes to be made to other related database records, or to lock one or more related records for the duration of the transaction
 - `endTransactionValidate`, for logic that needs to look at the state of the database after all the updates have been applied
 - `postTransactionValidate`, for any logic that needs to execute after the execution has completed
- The SBO can refer to all of the updates in all of the SDOs, using the logical `ObjectName` that has been given to each SDO.
- All of the logic in the individual SDOs continues to be executed when they are used within an SBO.
- The SBO can return error messages just as the SDOs can.
- Any changes made by the business logic of the SBO are seen by the SDOs when they execute the validation procedure of the same name.
- Any changes made to the updated records by the SBO are seen back on the client at the end of the transaction, just as changes made by the SDOs, or by database trigger procedures, are seen.

Using the Toolbar and Menu Designer

In the Progress Dynamics framework, a SmartToolbar is a standard dynamic object you use in your applications. The SmartToolbar is entirely data-driven. It supports both menu functions and toolbar buttons that drive an application or application window.

The Toolbar and Menu Designer is the tool that you use to create menu and toolbar items. You can also define the actions the items initiate and group items into bands and complete toolbars. With the Container Builder, you can then place a custom Toolbar object onto any window in your application. Progress Dynamics provides a large number of prebuilt Toolbars and bands to use as a starting point when appropriate.

This chapter describes the Toolbar and Menu Designer. It guides you in creating Toolbars and customizing existing ones. This chapter also describes the steps to build a new Toolbar and place it into an application window. There is also a completed sample that illustrates how you can use a Toolbar to organize your application and provide much of the application's behavior. This chapter discusses the following topics:

- [Overview of the Toolbar and Menu Designer](#)
- [Defining menu and toolbar items](#)
- [Defining toolbar bands](#)
- [Defining SmartToolbar objects](#)
- [Menu translations](#)
- [Adding a toolbar to an application window](#)

12.1 Overview of the Toolbar and Menu Designer

You can access the Toolbar and Menu Designer (Toolbar Designer for short) as follows:

- From the AppBuilder main window, select **Build**→**Toolbar and Menu Designer**.
- From the Progress Dynamics Administration window, select **Application**→**Toolbar and Menu Designer**.

The Toolbar Designer, shown in [Figure 12–1](#), is a TreeView-based window. All the categories, items, bands, and toolbars you work with are represented as nodes in the left-hand TreeView control. The tab folders where you define these objects are displayed on the right when you select a particular node.

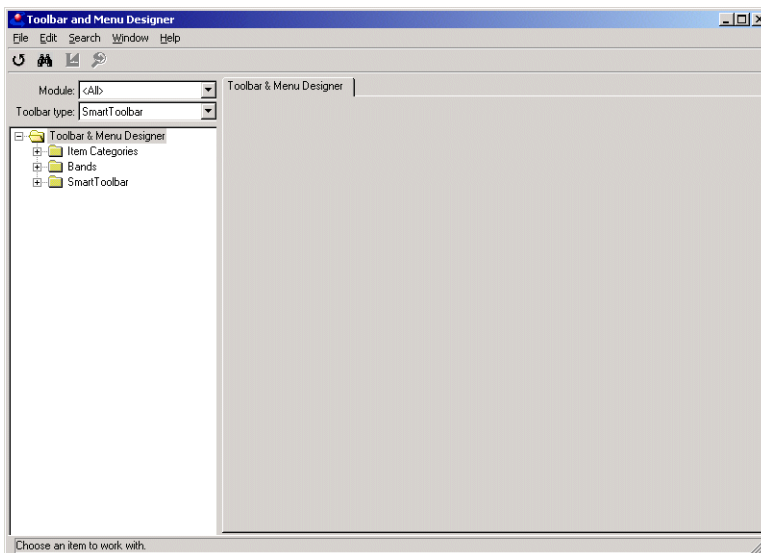


Figure 12–1: Progress Dynamics Toolbar and Menu Designer

Categories, Items, Bands, and Toolbars are all displayed in the TreeView control. Expanding a parent node in the tree shows the objects under that parent. The root node is always Toolbar & Menu Designer. The primary child nodes are Item Categories (which contains both Categories and Items), Bands, and SmartToolbars.

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.

By default, the Toolbar Designer shows objects from all Product Modules. To organize Toolbar objects that you create, you should generally assign them to a particular Module. To do this, select a Module from the drop-down list before you start defining Items, Bands, and SmartToolbars. If you do this, it will be easier to locate and maintain your objects later. Since the framework itself defines a great many Items, Bands, and Toolbars for the framework tools themselves, you will see all of these along with your own if you do not filter the objects by Module. In many cases, however, you will want to integrate some of these built-in objects into your own Toolbars. In this case, you must reset the Module drop-down list to <All> because these objects are not part of any particular Module.

NOTE: Template objects are not shown in the drop-down lists.

Selecting the Refresh button in the Designer's toolbar causes all nodes in the tree to collapse and refreshes the contents of the tree.

The Search button brings up the Search Nodes dialog box where you can search for bands that contain a specific item. For example, in [Figure 12-2](#) you can see that the First menu item in the Navigation category is used in two different bands, called Navigation and NavRight (shown in the Bands browser).

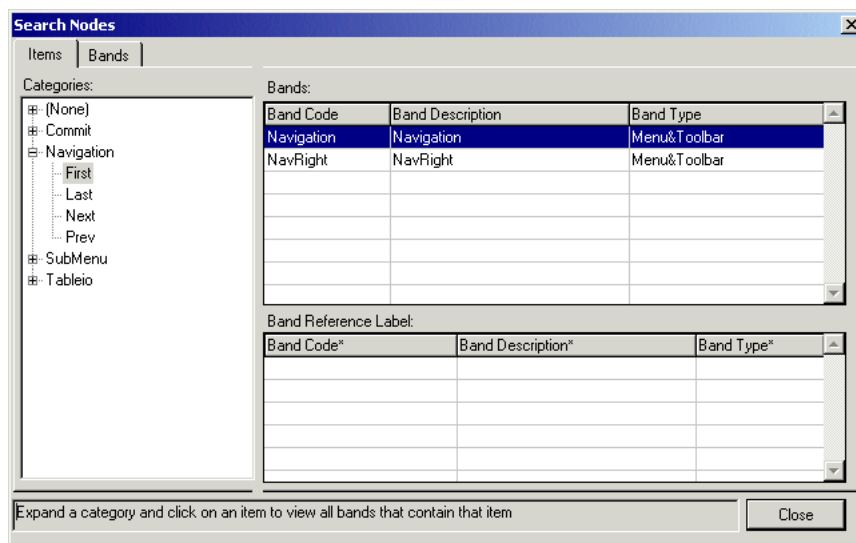


Figure 12-2: Search Nodes dialog box

If you select an item from the SubMenu category, you can use its label in other bands as the band label or submenu. In this case, a list of bands that use the item's label is displayed in the Band Reference Label browser in the bottom part of the dialog box. In [Figure 12–3](#), the File item, which is itself a SubMenu used to group other items, is used in a number of different bands, including the StandardMenuBar, NavMenuBar, etc. Its label is also used in a number of other bands, including many variants of the File band.

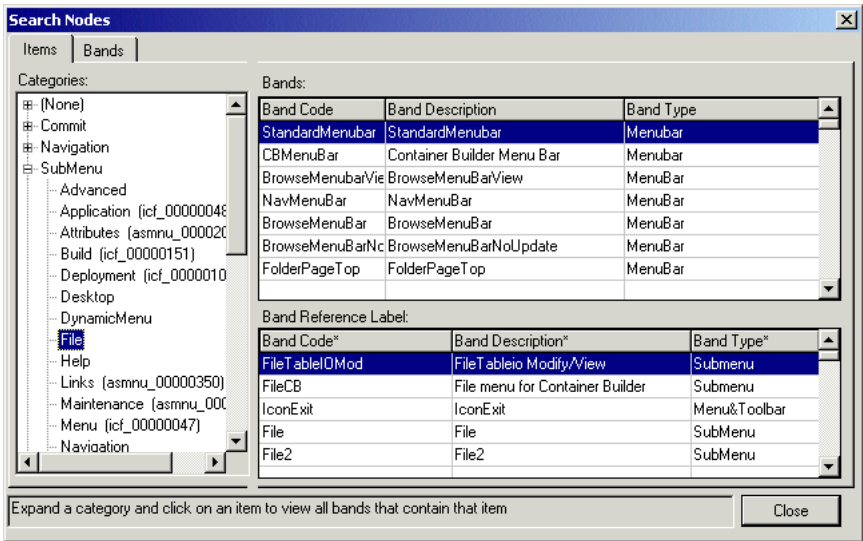


Figure 12–3: Search Nodes dialog box: Items tab folder

If you select the Bands folder tab at the top of the Search Nodes dialog box, you can then expand the Bands tree and select a particular band. The browser at the top right shows all the Toolbar objects that use that band. For example, [Figure 12–4](#) shows that the Help band is used in a number of different Toolbars.

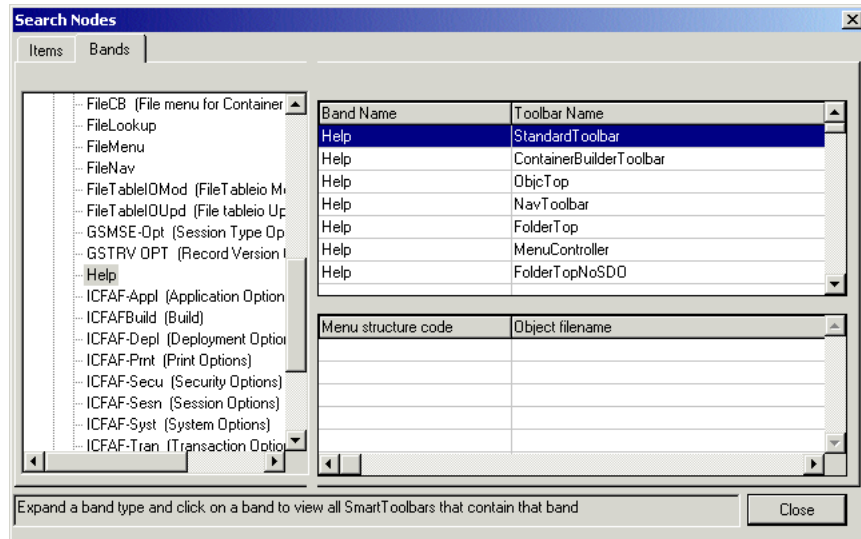


Figure 12–4: Search Nodes dialog box – Bands tab folder

If a band has been merged into a window’s Toolbar using the Band/Object association, then those window objects are displayed in the browser at the lower right.

The split bar is a selectable object between the TreeView control and the display panel on the right. By positioning the cursor over the split bar and dragging it to the right or left, you can change the relative size of the TreeView panel and the right-hand display panel, in order to see all the sublevels of nodes in the TreeView. You can also resize the entire window as necessary to avoid having scroll bars in the tab folders on the right.

12.1.1 Defining menu and toolbar items

An Item is an object that represents a single element of a Toolbar or Menu (or both). In a menu, it can be visualized as a menu item with a label. It can also simply be a label or a separator between other items. In a Toolbar it is represented as a button with either a text label or a bitmap Icon. Unless it is just a label or separator, an item will cause some action to occur when it is selected. Items are the basic building blocks from which you will build up all your Toolbars.

12.1.2 Defining categories

Categories are used simply to group Items in any meaningful way. To define Items, first expand the Item Categories node in the TreeView. A list of all existing categories appears. The categories Commit, Navigation, SubMenu, and TableIO are already defined as part of the framework. A great many Items also have no specific category, and these are displayed under the (None) category heading. You are free to use this category, but usually you should organize your Items into meaningful categories just to keep track of them more easily. In addition, if you want a group of Items to have a specific SmartLink associated with them by default, you must define this at the level of the Category explained below.

Figure 12–5 shows a list of Categories. If you expand a Category, you see all the Items under that Category.

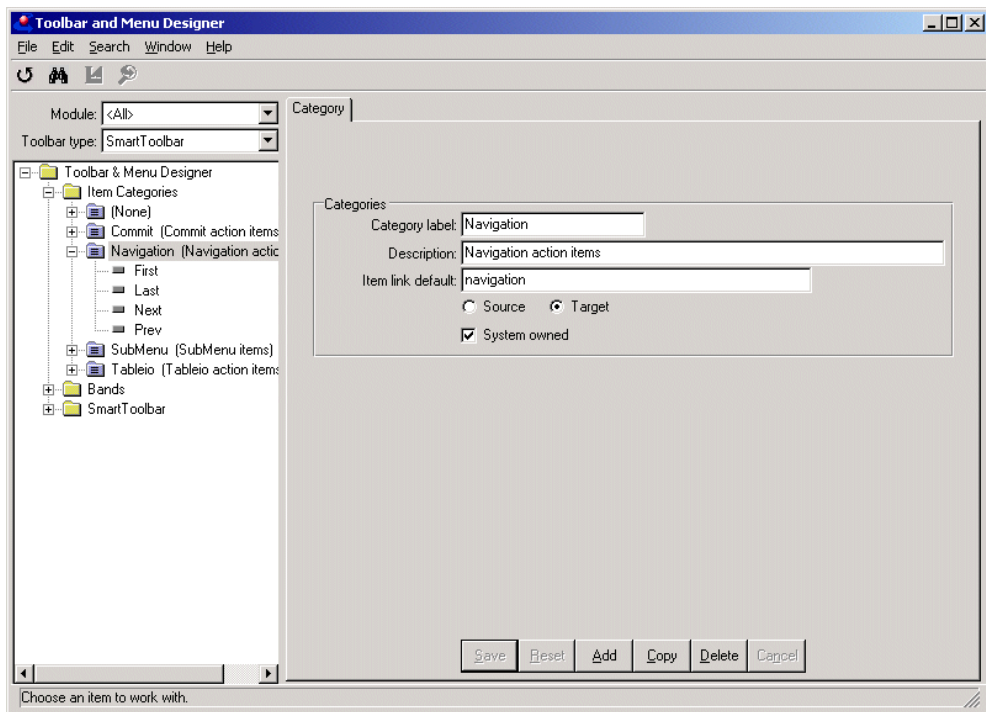


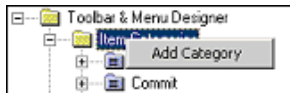
Figure 12–5: Toolbar and Menu Designer - Category tab folder

NOTE: This screen capture shows a few more categories than you will see when you open your Toolbar Designer, because of various sample objects, some of which you will be defining yourself.

Adding a new category

To add a Category, do one of the following:

- Right-click on the **Item Categories** node and select **Add Category**, as shown below:



- Or, select any existing Category, then choose the **Add** button in the Update panel of the Category tab folder on the right.

If an Item publishes an event when selected, you should specify the link name used to categorize this event when you define the Item in the Item Link Default field. If no item-link is specified, the event is published from the container.

A Toolbar can be a Source or Target for any number of different SmartLinks (or just Links for short), which are used to communicate events and messages between the different objects in an application. For example, a Navigation band in a Toolbar publishes the fetchFirst event when the user chooses the First button or selects the equivalent menu item. Another object, such as an SDO, is subscribed to this event in the Toolbar automatically when there is a Navigation link between the two objects. When the event occurs, the procedure named fetchFirst in the subscriber is executed and the subscriber can in this way respond to the event. The publisher of the event is called the Source, and the subscriber is the Target.

The First Item publishes the fetchFirst event. Likewise, the Next, Prev, and Last Items publish fetchNext, fetchPrev, and fetchLast respectively. Collectively these are referred to as Navigation events. The Navigation link defines the association between objects for all of them.

In a case where the events are grouped in this way, you can organize the corresponding Items into a Category where the link name is defined. This will then become the default link name for all Items in the Category. Specify the name of the link, and whether it is the Source or the Target. Any toolbar containing Items of this Category will then be a candidate to establish the proper link. If there is no default link for the Category, leave this field blank.

12.1.3 Creating a category

In these sections, you will go through the steps to build an example. These steps will help you to understand how Toolbars are constructed and how they operate.

Follow these steps to create a new Category:

- 1 ♦ Right-click on the **Item Categories** node, then select **Add Category**.
- 2 ♦ Enter **Tutorial** as the Category Label, and some meaningful description.

- 3 ♦ There is no link associated with the new Category, so leave the Item Link Default blank. The setting of Source/Target does not matter in this case.
- 4 ♦ Set the System Owned toggle box to **False** in this and all other new Items and bands that you define as part of this example.

12.1.4 Defining items

An *Item* can be a menu item in a menu or a tool in a toolbar or it can be both. It is a visual representation used to perform an action within the application. The item can publish an event, run a procedure, or launch another application window. When in a toolbar, the Item can be visualized as a button, label, or separator. Items can be reused in multiple bands.

To create a new Item, do one of the following:

- Right-click on the Category where the Item is to go, and select **Add Item**. [Figure 12–6](#) shows you all the data that can be entered for an Item. The First button for the Navigation category is used as an example.
- Select any existing Item and then choose the **Add** button in the Update panel in the Item Maintenance tab on the right.

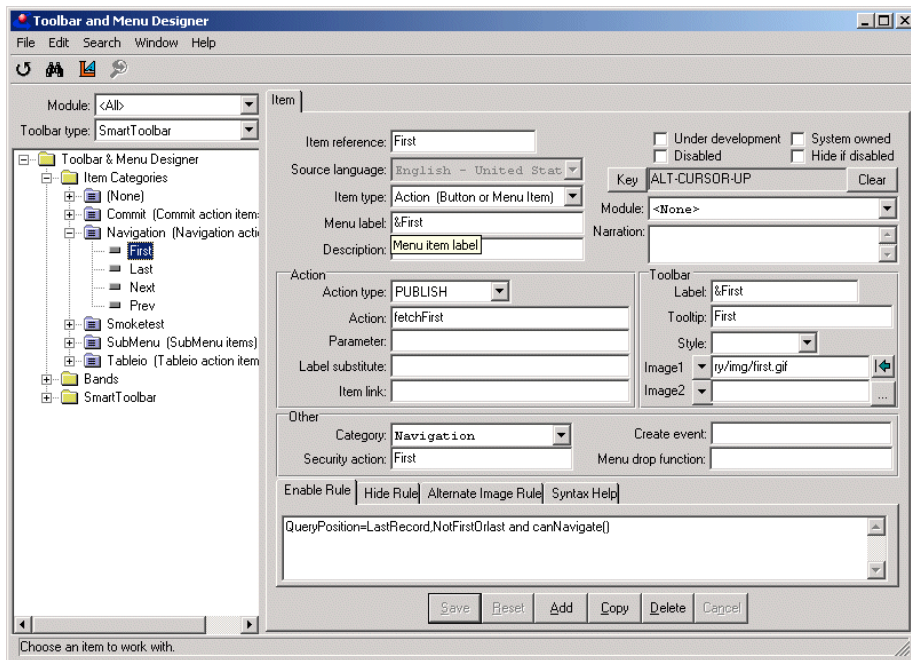


Figure 12–6: Toolbar and Menu Designer - Item tab folder

The Item Type can be one of the following:

- **Action** — Specifies that this is a user action that appears on a menu or toolbar. When on a toolbar, the item is visualized as a button. When on a menu, the item is visualized as a menu item.
- **Label** — (for menus only) Indicates this item will be used strictly as a submenu item. That is, it defines a label to be used to identify a submenu defined separately.
- **Placeholder** — Used as a placeholder for dynamic menus or toolbar bands. Each individual container where the Toolbar object is used can define its own band of items to be inserted in the spot reserved by the placeholder. (See the [“Band object associations”](#) section for more information on how this is defined.)
- **Separator** — Can be specified when the action is on either a toolbar or menu. When on a menu, a rule (a separating line between items) is created. When in a toolbar band, a line appears separating the buttons. This type of item cannot be added to a top-level (MenuBar) menu structure.

Rules

Rules are used to enable/disable items, hide/view items, or swap images in toolbar buttons. A rule contains a delimited list of either function references or properties that specify a logical result (either Yes or No):

Syntax: [property | function] = list [AND | OR] ...

property

The name of a property whose value is retrieved across the specified Item link. This can be any SmartObject property defined in the object across the Item link.

function

A function that is executed across the specified Item link

list

A comma-delimited list of values that is compared to the property or function result. An OR comparison is performed.

You can define rules to change the state of Items in the following ways:

- **Enable Rule** — Defines whether the item is enabled.
- **Hide Rule** — Defines whether the item will be hidden or viewed. A logical YES result hides the image. A NO result displays the image.
- **Alternate Image Rule** — If there is an alternate image defined (Image 2), this defines the criteria to show either Image1 or Image 2. A True result swaps the image with Image 2. A False result swaps the image with Image 1.

Here is an example from the Enable Rule for the First navigation button:

```
QueryPosition=LastRecord,NotFirstOrLast and canNavigate()
```

QueryPosition is an SDO property that keeps track of whether the SDO's query is positioned at the beginning, the end, or somewhere else in the data set. The expression "QueryPosition=LastRecord,NotFirstOrLast" means "IF QueryPosition = LastRecord or QueryPosition = NotFirstOrLast". In other words, if the SDO query is positioned on the LastRecord or positioned somewhere in the middle, then it is appropriate to enable the First button to allow the query to be repositioned to the beginning.

The canNavigate function is defined for SDOs also (in the super procedure `data.p`, among other places). It returns true if there are no dependent uncommitted updates pending that need to be committed before the query can be repositioned. If both the QueryPosition and canNavigate conditions are met, then the First Item (menu item or toolbar button) is enabled. Similar rules are defined for the other navigation Items (Next, Prev, and Last).

Here is another example for the Save Item in the TableIO Category, which groups buttons and menu items used in the update bands of toolbars:

```
NewRecord=add,copy or DataModified
```

The NewRecord SDO property is set to Add if a new record is being added. It is set to Copy if a new record is being created based on an existing record. Otherwise it is blank. The DataModified SDO property is True if an existing record has been modified on the screen but not yet saved. If either of these conditions is met, then the Save Item is enabled.

For more information on SmartObject properties and functions you can use to form rules for changing the state of your Items, refer to the [Progress Dynamics ADM2 API Reference](#) manual.

NOTE: If you create a rule using custom properties, remember to define the properties in the class. The class must also contain a get function to retrieve each new property.

12.1.5 Creating items

In this section you will define the following different Items for your example Toolbar:

- A set of Cut/Copy/Paste Items, which will be viewed both as menu items and as toolbar buttons
- A Label menu item to group Cut/Copy/Paste when they appear on the menu
- Another Label menu item to provide access to the built-in navigation band
- A Label menu item that will be used as a placeholder for another band to be merged into the toolbar at run time
- An Action menu item to launch the Progress Dynamics Administration menu

The Cut/Copy/Paste actions will run a procedure that you will write in a custom super procedure supporting the Toolbar.

Figure 12–7 shows a sneak preview of what the final Toolbar will look like, placed into a Customer maintenance window.

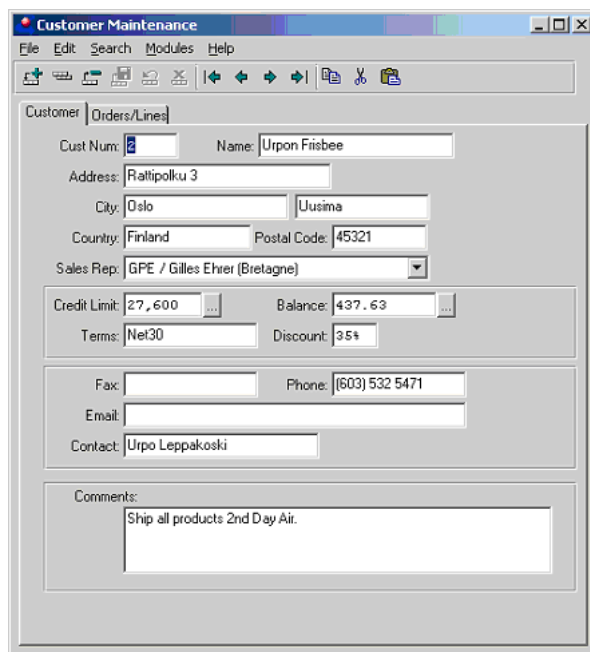


Figure 12–7: Example toolbar

File menu

The File menu, shown in [Figure 12–8](#), is a standard, predefined Progress Dynamics File menu that you will add to your Toolbar.

File	Edit	Search	Module
Add record	Alt+A		
Copy record	Alt+C		
Delete record	Alt+D		
Modify record	Alt+M		
View record	Alt+V		
Save record	Alt+S		
Reset record	Alt+R		
Cancel record	Alt+L		
Filter...	Alt+F		
Translate...			
Print Setup...			
Exit	Alt+X		

Figure 12–8: File menu

The update-related items in this menu (Add, Copy, Delete, Modify, View, Save, Reset, Cancel) will also be buttons. Menu items and toolbar buttons are always enabled and disabled together, as appropriate for the current state of the maintenance operation, and as defined by the Rules for each item. In this case, for example, no changes have been made to the current record (so the DataModified property is False). As a result, the Save, Reset, and Cancel items, both as menu items and as toolbar buttons, are disabled.

Edit menu

The Edit menu item, shown in [Figure 12–9](#), contains three other items that define Cut, Copy, and Paste operations.

Edit	Search	Mod
Copy	Ctrl+C	
Cut	Ctrl+X	
Paste	Ctrl+V	

Figure 12–9: Edit menu

Search menu

These same three operations are also available as buttons on the toolbar. The Search menu item, shown in [Figure 12–10](#), will be a new label to which you will attach a standard Progress Dynamics Navigation band.

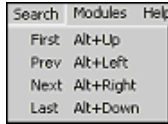


Figure 12–10: Search menu

These items will also be seen as Navigation buttons on the toolbar.

Modules menu

The Modules item, shown in [Figure 12–11](#), is a label under which you will place an Action Item to launch the Progress Dynamics Administration window, as shown in [Figure 12–12](#).



Figure 12–11: Modules menu

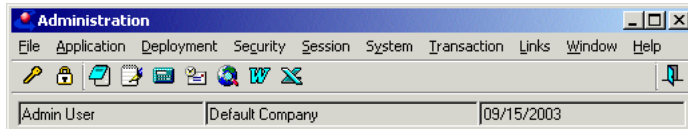


Figure 12–12: Progress Dynamics Administration window

Help menu

Finally, the Help menu item brings in the standard Progress Dynamics Help menu.

Creating items

Follow these steps to create this set of Items:

- 1 ♦ Right-click on your new **Tutorial Category** and select **Add Item**:

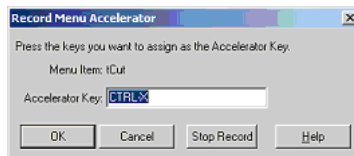


- a) Create the Label item for the Edit function, which will group together the cut, copy, and paste operations that you define next.
- b) Enter **tEdit** for the Item Reference, **Edit Label** for the Description, **&Edit** for the Menu Label.
- c) Select **Label** as the Item Type.
- d) Enter a Narration such as “**Edit label for use with cut/copy/paste operations.**” Because this Item is just a label, the Action list is disabled, because no action is associated with a label.
- e) Choose **Save** to save this first Item.

- 2 ♦ Choose **Add** in the Item Maintenance tab to create the next new Item. Follow these steps:

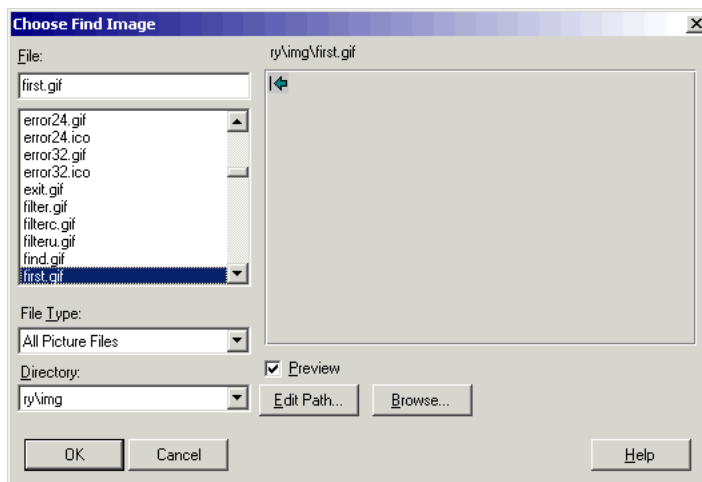
This next Item is the first of three Action items that will run a procedure to perform cut, copy, and paste operations in the window where the Toolbar is placed.

- a) Enter **tCut** as the Item Reference, **Cu&t** as the Menu Label, **Cut** as the Description, and **Action** as the Item type.
- b) To define a short-cut key for the Item, choose the **Key** button to display the Record Menu Accelerator dialog box:



- c) Press the control sequence **CTRL-X**, then choose **OK** to save this sequence.
- d) Enter the Narration “**Cuts the current select and puts it on the clipboard.**” Because you chose an Item Type of Action, the Action Type list is enabled.

- e) Select **RUN** from this list.
 - f) Enter **EditAction** as the Action. This is the name of an internal procedure that you will write a little later to perform the Cut operation.
 - g) Enter **Cut** as the parameter. This value will be passed as a CHARACTER literal to the EditAction procedure, so that it will know what kind of action to perform.
- 3 ♦ To make the cut, copy, and paste operations become toolbar buttons, as well as menu items, follow these steps to define those attributes of the Item next:
- a) In the Toolbar section of the Item tab, the value Cut should be set for you as the default for the Label. Since you will be displaying the Item in the toolbar using an image, the label does not matter, so leave this as the default value.
 - b) Choose the **Image 1** button to display the following dialog box:

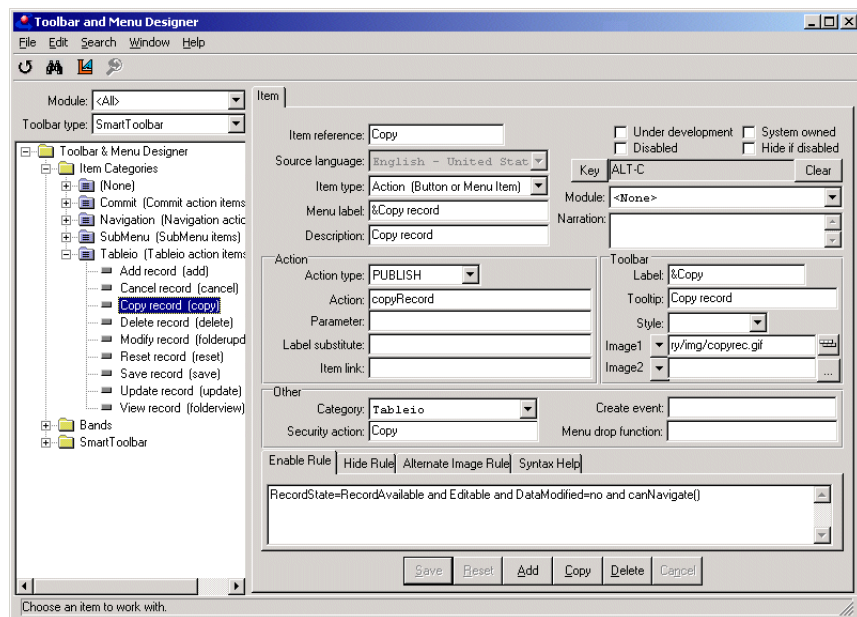


- c) Select **ry/img** as the Directory, and then locate the objectcut.bmp image file in that directory.
- d) Choose **OK** to select it.

- e) (Optional) Alternatively, you could click the drop-down arrow next to the Image1 label and select PicClip1. If an image is the importation of a graphic file, a PicClip is the importation of a cropped piece of a graphic file. You could keep all of your icons in one compact file and use this feature to clip the appropriate icon out of the file. When you select a PicClip image, you must also enter the offsets in the format:

image, x-offset, y-offset, width, height

- f) Choose **Save** to record your definition of the Cut Item. All of your data should look like this:



In this case, only the Cut and Edit Items will be displayed as a node in the Tutorial Category in the TreeView. The rest of the Items you will create next.

- 4 ♦ To create two more Items just like the Cut item, follow these steps:

- a) Choose **Add** to create the first Item.
- b) Give it the Item Reference, **tCopy**.
- c) Define its shortcut key to be **CTRL-C**.
- d) Pass Copy as the parameter for the EditAction procedure.

- e) Choose the `afcopy.bmp` image file to represent it in the toolbar. All the other choices should be the same as for the Cut Item.
 - f) Choose **Add** to create the second of the two remaining edit items, the Paste Item.
 - g) Give this an Item Reference name **tPaste**.
 - h) Define its shortcut key to be **CTRL-V**.
 - i) Pass Paste as the parameter for the EditAction procedure.
 - j) Choose the `afpaste.bmp` image file to represent it in the toolbar.
- 5 ♦ Follow these steps to create a label for a menu item to which you can add other related application modules, so that the user can invoke these from the window that contains your new Toolbar:
- a) Add another new Item.
 - b) Give it the Item Reference, **tModule**, a menu label of **&Modules**, and a Description of **Module Label**.
 - c) Select the Item Type, **Label**, and give it an appropriate narration as a description.
 - d) Choose **Save** to complete the Modules Item.
- 6 ♦ The next Item you create (using the following steps) will launch a dynamic container object when selected, the Progress Dynamics Administration menu controller window:
- a) Choose **Add** to create another Item.
 - b) Give it an Item Reference of **tAdmin**, a menu label of **&Admin**, and a Description of **Progress Dynamics Administration**.
 - c) Select an Item Type of **Action**.
 - d) Select **Launch** as the Action Type. This changes the field below the Action Type to be Object Filename.
 - e) Enter the name of the Administration control window, **afallmencw**.

NOTE: Note the difference between choosing an **Action Type** of RUN, which will run an actual 4GL procedure, and LAUNCH, which will make the framework read the Repository records for a logical object that has no procedure, and instantiate it as a dynamic object.

- f) Save the Admin Item.
- g) Choose **Add** to create one final Item. This will be another label item used to provide access to a Navigation band of items already defined for you in the framework.
- h) Give the new Item an Item Reference of **tSearch**, a menu label of **&Search**, a Description of **Search Label**, and an Item Type of **Label**.
- i) Choose **Save** to complete the definition of this final item.

After the discussion of Bands in the next section, you will define bands where you will group these items together.

12.1.6 Copying nodes

To create a new node by copying an existing one:

- 1 ♦ Select the source node in the treeview.
- 2 ♦ Click the **Add** button at the bottom of the tab. The framework creates a copy of the selected node and displays a set of identical values in the tab.
- 3 ♦ Give the new node a new unique **Band code** and a new unique **Band description**.
- 4 ♦ Change other values as necessary.
- 5 ♦ Click the **Save** button.

12.1.7 Editing nodes

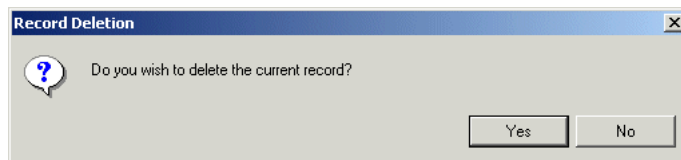
To edit an existing node:

- 1 ♦ Select the node in the treeview.
- 2 ♦ Click the **Edit** button at the bottom of the tab. The framework enables all the appropriate fields for input.
- 3 ♦ Change values as necessary.
- 4 ♦ Click the **Save** button.

12.1.8 Deleting nodes

To delete an existing node:

- 1 ♦ Select the node in the treeview.
- 2 ♦ Click the **Delete** button at the bottom of the tab. The framework displays a confirmation message:



- 3 ♦ Click Yes.

12.2 Defining toolbar bands

A *band* (also referred to in Progress Dynamics as a menu structure) is a logical container for a series of Items. It serves as a grouping mechanism for a set of Items that should be used together. You can visualize it as a submenu in a menu, or as a band (a visual group of buttons) in a toolbar. To help organize bands in the Toolbar Designer's TreeView, bands are grouped into the following subgroups:

- **MenuBar Bands** — Top-level menu bars that appear on a container.
- **Submenu Bands** — Menu bands that are parented to the menubar or to another submenu.
- **Toolbar Bands** — Those that only appear in a toolbar.
- **Menu & Toolbar** — Bands that can appear in both a toolbar and a submenu.

Bands are also hierarchical in nature and can contain child bands. You can identify an Item within the band as having a child band. In a menu, this creates a submenu for the Item. You can reuse bands in multiple SmartToolbars.

To define a new Band, follow these steps:

- 1 ♦ Expand the **Bands** node in the TreeView, then expand the appropriate sub node.
- 2 ♦ Right-click on the node and select **Add Band**:



Figure 12–13 shows the fields that you can enter to define a new Band.

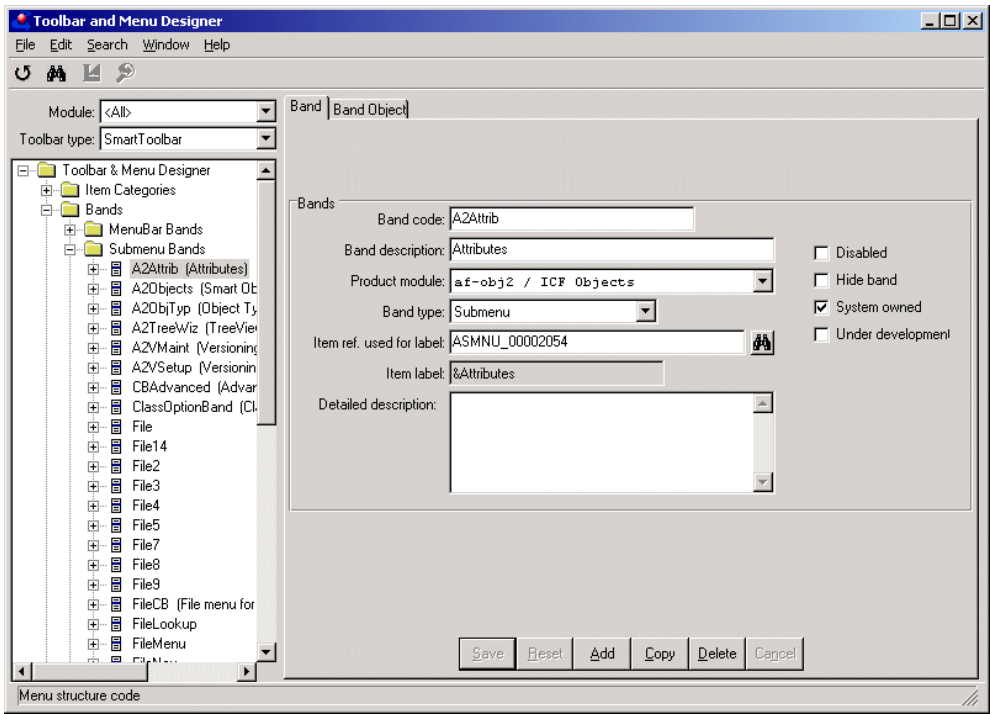


Figure 12–13: Defining toolbar bands

12.2.1 Adding items to bands

Once you have created a new Band, you must add Items to it. You need to group these Items in a particular sequence that identifies the order in which you want them to appear in the Band.

To add an existing Item to a Band, follow these steps:

- 1 ♦ Right-click on the TreeView node for the Band.
- 2 ♦ Select **Add Item to Band**. The Band Item Maintenance folder appears.

12.2.2 Displaying band items and adding items on the fly

The Band Item Maintenance folder has a second tab labeled Item (add to Band). Select this tab to see the detailed information for any Item you have selected. If you want to add an Item that does not yet exist:

- 1 ♦ Choose the **Add** button.
- 2 ♦ Enter the Item information.
- 3 ♦ Choose the **Save** button. The Item is created and added to the current Band in one step (in sequence after the previously selected Item).

12.2.3 Band object associations

You can associate an object with one or more menu structures. When the container of a window is initialized, it searches for all associated menus for all objects contained within the window. It then inserts or merges each menu structure into the menu or toolbar. You can also specify where to merge the menu bands by specifying a placeholder item within the menu hierarchy.

Figure 12–14 shows the Band Object tab using the Progress Dynamics Session Options band as an example. As you can see, the Progress Dynamics Session Options band is associated with a large number of different container objects, as shown in the Object Filename browser field.

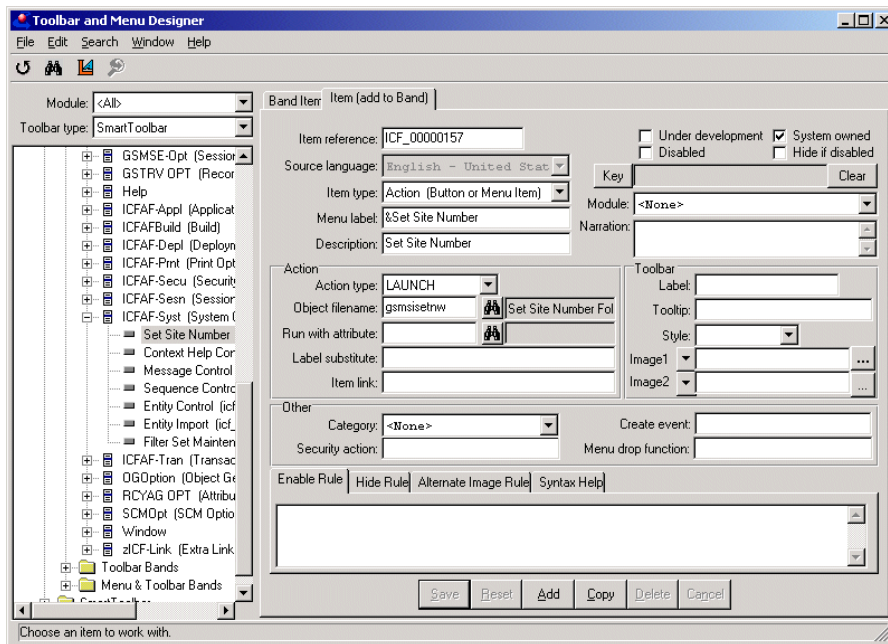


Figure 12–14: Toolbar and Menu Designer – Band Object tab

To add an object association, follow these steps:

- 1 ♦ Choose the desired band node.
- 2 ♦ Select the **Band Object** tab.
- 3 ♦ Choose the **Add** button in the Update panel.

To illustrate, consider the File band that contains the items shown in [Figure 12–15](#).

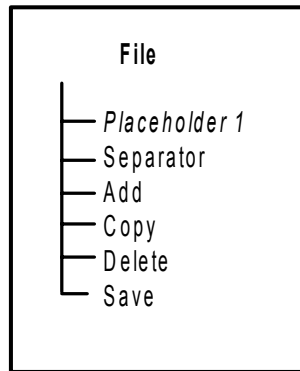


Figure 12–15: File band

Assume that a Navigation band is to be merged into the File band at the Placeholder 1 position. [Figure 12–16](#) shows the results.

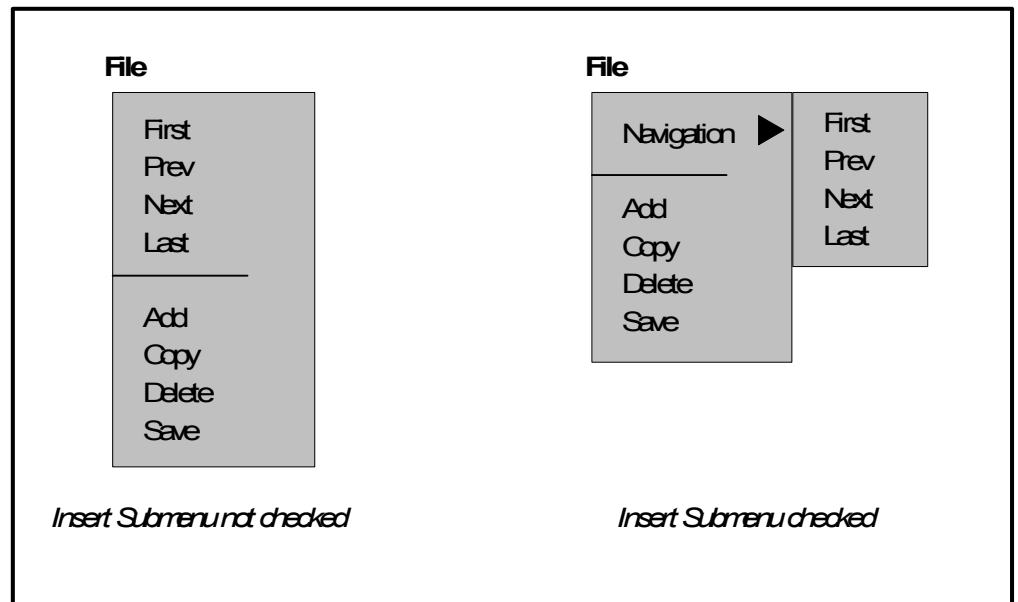


Figure 12–16: Merging bands: results

Figure 12–17 shows an example to make it a bit clearer how bands are defined and merged.

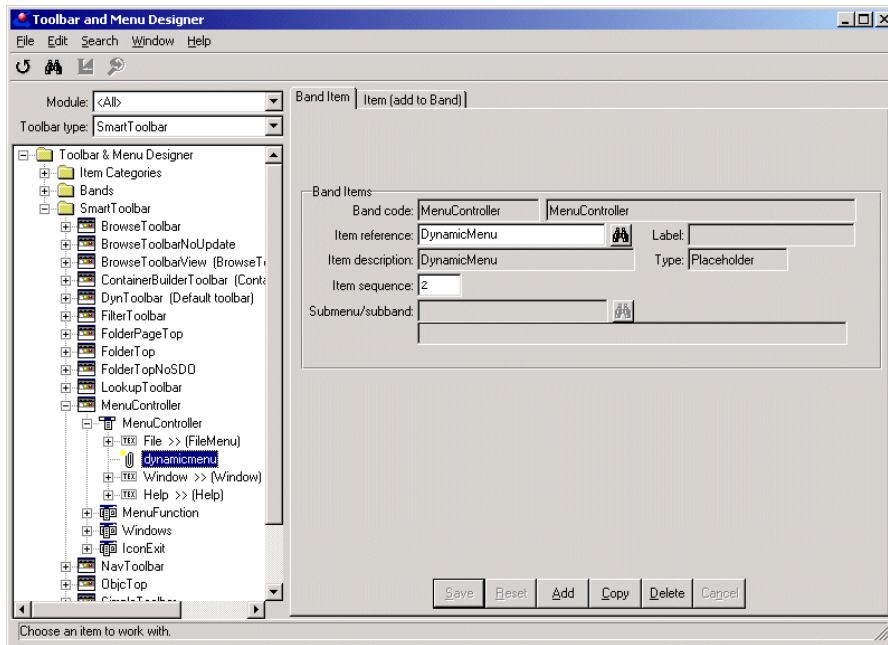


Figure 12–17: Toolbar and Menu Designer – Band Item tab

Figure 12–17 shows expanded parts of the MenuController Toolbar. This is the standard Toolbar that appears in the type of dynamic window called a Menu Controller, which you have already seen in the Progress Dynamics Administration window, and perhaps also as the main window of the sample application in the *Getting Started with Progress Dynamics* manual. As you can see, the MenuController Toolbar contains four Bands: a MenuController band, a MenuFunction band, a Windows band, and an IconExit band.

The MenuController band in turn contains four sub-bands: File, DynamicMenu, Window, and Help.

Any bands that you merge into a container with the MenuController Toolbar appear in the location identified by the DynamicMenu placeholder band. Since this band appears after the File menu, and before the Window and Help menus, the merged band or bands also appear in that sequence.

Figure 12–17 shows that the Progress Dynamics Session Options band is associated with, or merged into, a number of different container objects. The Progress Dynamics Administration window, named `afallmenw`, is one of these. In that window, it has a Band Sequence of 6, meaning that it is the sixth sub-band to be inserted into that one placeholder following the File menu. (In fact, because some of the sequence values are not used, it is the fourth sub-band merged in. In other words, sequence values do not need to be contiguous, only in the proper relative order.) The Progress Dynamics Administration window in Figure 12–18 shows the effects of this.

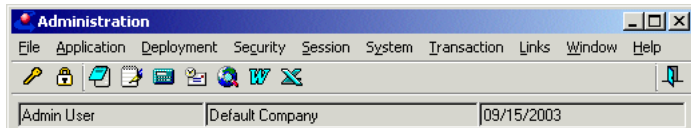


Figure 12–18: Progress Dynamics Administration window example

The Application, Deployment, Security, Session, System, Transaction, and Links subbands have all been merged into the placeholder for the MenuController band, appearing between the File band and the Window band.

If you drop down the File menu, you will see all the items that belong to the FileMenu band, as shown in the TreeView structure. This included Re-Logon, Suspend, etc.

The Window band contains a single item that lets the user alternate between having multiple windows for different data items and having them displayed in a single window. For example, if one Customer is selected in a Customer Object Controller or browse window, it brings up the Customer Maintenance folder. Then if another Customer is selected, this toggle box determines whether a second instance of the Customer Maintenance window comes up for the second selected Customer.

NOTE: If you follow this thread by walking through this set of related elements in the Toolbar Designer, you can find out a little about how this works. Selecting the Window band (not to be confused with the Windows band), you will see that it contains an Item called `multiwindow`. If you then locate this Item under the `<None>` category, you will see that the property that is alternated on and off here is called `MultiInstanceActivated`. This property is examined by the framework code at run time to determine the correct behavior.

The Help band contains the usual Help options.

The MenuController band has two more sub bands in it: a Windows band (different from the Window band), and an IconExit band. They are both defined as Menu&Toolbar Bands. The toolbar buttons that appear at the left side of the Menu Controller window are the toolbar visualization of the Items in the Windows Band. These Items provide access to MS Windows applications such as Excel and Word. In the MenuController hierarchy, the same Windows band is also a sub band of the File menu. These same functions will appear as menu items in a submenu under the File menu. The IconExit band defines the Exit button that appears on the right side of the Menu Controller window.

So this is how you can use Bands to form parts of a larger whole, represented by the Toolbar object itself, which you will examine in more detail in the next section.

Before you do that, however, you have some bands of your own to define as part of the example exercise.

12.2.4 Creating bands

This section describes how to put together the Items you created in the previous section to form Bands.

You will create the following three bands:

- An Edit Band, where you can group the Cut/Copy/Paste Items.
- A Module Band, where you can merge in an appropriate other band or bands for a specific container window.
- A top-level MenuBar band, where the other new bands and a couple of standard Progress Dynamics bands such as those you have looked at will be grouped together.

Follow these steps to create these bands:

- 1 ♦ Expand the **Bands** node in the TreeView.
- 2 ♦ Right-click on the **Menu&Toolbar** node, then select **Add Band** from the pop-up menu.
- 3 ♦ Enter a Band Code of **tEditBand**, a Description of **Edit Band**, and an appropriate Detailed Description, such as “**This band is used to contain the cut/copy/paste items.**” Because you added this band under the Menu&Toolbar node, the Band Type is properly set to that value.
- 4 ♦ Choose **Save** to complete this Band definition.
- 5 ♦ Choose **Add** in the Band Maintenance tab to add a second new Band.

- 6 ♦ Enter a Band Code of **tModuleBand**, a Description of **Module band**, and a Detailed Description: “**This band is to be merged for the customer maintenance window.**” This is to be a SubMenu Band.
- 7 ♦ Because you just chose Add rather than first selecting the SubMenu node in the TreeView, change the Band Type to **SubMenu**.

Because this is a SubMenu band, it needs a Label, which when selected brings up the submenu. The Label needs to be an existing Item of that type.
- 8 ♦ Enter **tModule** as the Item Reference used for Label.
- 9 ♦ Choose **Save** to complete this Band.
- 10 ♦ Choose **Add** once more to create a third new Band. This Band will represent the top-level menu for your Toolbar. Under it you will place the Edit Band, the Search Item that you created earlier as a Label for the built-in Navigation Band, a standard File menu band, a standard Help menu band, and a DynamicMenu placeholder where the Module band will go.
- 11 ♦ Enter a Band Code of **tStandard**, a Band Description of **Standard MenuBar**, a Band Type of **Menubar**, and the Detailed Description: “**This is the standard top-level menu bar for maintenance toolbars.**”
- 12 ♦ Choose **Save** to save this final new Band.

Since there are a lot of elements to a complex Toolbar, and you need to create and organize objects at several different levels, it is important to design your Toolbars and their contents before you start creating them. In particular, explore the Items, Bands, and Toolbars that come with the framework to save yourself the trouble of creating something similar from scratch. And organize your own new objects in such a way as to permit maximum reuse. This way you will not only save yourself development and maintenance chores, but also provide a more consistent interface for your application.

Associating items with bands and subbands with other bands

Follow these steps to associate Items with Bands and SubBands with other Bands.

- 1 ♦ Select the Edit Band node that you created in the previous steps. This node is under the Menu&Toolbar node. Note that it is the Band Description, not the Band Code, that you see in the TreeView.
- 2 ♦ Right-click on the **Edit Band**, then select **Add Item to Band** from the pop-up menu.

- 3 ♦ Add the following three Band Items. For each of these you need only specify the Item Reference and the Item Sequence:
 - Copy — 1
 - Cut — 2
 - Paste — 3
- 4 ♦ Save each of these in turn to associate them with the Edit Band.
- 5 ♦ Select the **Module Band** from under the SubMenu node in the TreeView.
- 6 ♦ Right-click on the node, then select **Add Item to Band**.
- 7 ♦ Add a single Item to the Band: the **tAdmin Item** with a sequence of **1**. This places the menu item that brings up the Progress Dynamics Administration window under the Modules menu item.
- 8 ♦ Save this Band Item.
- 9 ♦ Select the **Standard Band** from under the MenuBar node, and right-click on it.
- 10 ♦ Add these five Items to the Standard Band:
 - File, with an Item Sequence of **1** and a Submenu/SubBand of **FileTableIOMod**.
 - Edit, with an Item Sequence of **2** and a Submenu/SubBand of **tEditBand**.
 - Search, with an Item Sequence of **3** and a Submenu/SubBand of **Navigation**.
 - DynamicMenu, with an Item Sequence of **4** and no Submenu/SubBand.
 - Help, with an Item Sequence of **5** and a Submenu/SubBand of **Help**.

Now you have created all the Bands for your application Toolbar. As a final step in working with bands, you will associate the Module Band with a particular application window. (In the next section of this chapter, you will create a Toolbar object for this application window.). This example uses the Customer maintenance window that you might have already created as part of the *Getting Started with Progress Dynamics* tutorial, called `custfoldwin`. If you do not have this container window, you can associate the Module Band and later the new Tutorial Toolbar with any other maintenance window you create:

- 1 ♦ Select the **Module Band**, then select the **Band Objects** tab in the folder.
- 2 ♦ Choose the **Add** button in the Update panel in the Band Objects tab.

- 3 ♦ Enter an Object Filename of **custfoldwin** and a Band Sequence of **1**. Leave the Insert Submenu toggle box checked (True).
- 4 ♦ Choose **Save**.

12.3 Defining SmartToolbar objects

Toolbars and menu structures are created as a single object. The example you will create in the next section is a standard lookup window, which would typically contain a standard menu structure along with a toolbar containing one or more bands. The term *SmartToolbar* (or just *Toolbar* for short) refers to a combination of both a menu structure and toolbar, although it can contain any combination. See [Figure 12–19](#).

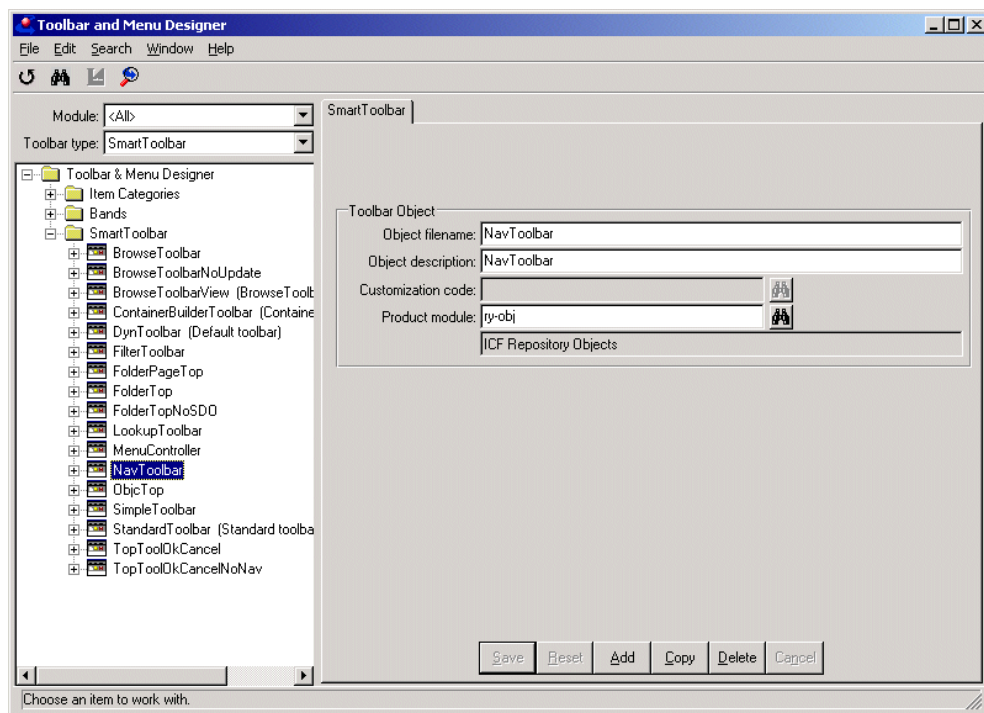


Figure 12–19: Toolbar and Menu Designer – SmartToolbar Tab

To add a SmartToolbar object, use either of the following techniques:

- Right-click on the **SmartToolbars** node, then choose **Add Toolbar as Object**.
- Choose any other displayed toolbar object, then choose the **Add** button in the Update panel.

12.3.1 SmartToolbar bands

Once you create a SmartToolbar object, you can then associate a MenuBar and/or one or more toolbar bands with it to complete your Toolbar definition.

NOTE: A SmartObject can have only one menu Band associated with it. This Band must be of Band Type MenuBar. This is a top-level Band that defines the entire menu structure. You can also assign Multiple Bands of type Toolbar/Menu or Toolbar to the Toolbar.

To add a band to a toolbar, use either of the following techniques:

- Right-click on any toolbar node, then select **Add Toolbar Band**.
- Choose any other displayed toolbar bands, then choose **Add** in the Update panel.

After you add a band to a Toolbar object, you can expand the Band in the Toolbar Designer to view all the Items in that Band. If an Item contains a child Band, you can expand that Item to display the SubBand showing all the Items in that Band.

12.3.2 Menu bar options

The Toolbar Designer menu bar contains the options listed in [Table 12–1](#).

Table 12–1: Toolbar Designer menu bar (1 of 2)

Menu	Menu options	Submenu options
File	<u>N</u> ew	<u>C</u> ategory
		<u>I</u> tem
		<u>B</u> and
		<u>T</u> oolbar/Menubar
	<u>E</u> xit	—

Table 12–1: **Toolbar Designer menu bar**

(2 of 2)

Menu	Menu options	Submenu options
Edit	Cu t (CTRL-X)	—
	C o py (CTRL-C)	—
	P a ste (CTRL-P)	—
Window	1. AppBuilder	—
Search	L o okup Items & Bands	—
Help	Help Topics	—
	Help Contents	—
	How to use Help	—
	Help About	—

12.3.3 **Defining the example SmartToolbar object**

The next step in building your sample Toolbar is to define the Toolbar object itself and associate Bands with it. Refer to [Figure 12–19](#) to see what the Toolbar Designer looks like when you create a Toolbar and add Bands to it.

To define the Toolbar, follow these steps:

- 1 ♦ Right-click on the **Toolbar/MenuBars** node in the TreeView, then select **Add Toolbar Object** from the pop-up menu.
- 2 ♦ Enter **TutorialToolbar** as the Object Name and **Sample Toolbar** as the Object Description.
- 3 ♦ Select an existing Product Module for the Toolbar. This example uses the General Module that you might have defined in the [Getting Started with Progress Dynamics](#) tutorial. Note that the Module for a Toolbar object cannot be defaulted.
- 4 ♦ Leave the Physical Object at its default value.
- 5 ♦ Right-click on the **Sample Toolbar** node in the TreeView, then select **Add Toolbar Band**.

- 6 ♦ Enter **tStandard** as the Band Code. Because this is a MenuBar band, none of the other fields apply, and will be disabled.
- 7 ♦ Choose **Save** to save this first Band.
- 8 ♦ Choose **Add** to add a second band. This is a Menu&Toolbar band, so its Items appear in both the menu and in the toolbar.
- 9 ♦ Enter **TableIO** as the Band Code, **1** for the Band Sequence, and **LEFT** for the Band Alignment.
- 10 ♦ Save this Band.
- 11 ♦ Choose **Add** to add a third band.
- 12 ♦ Enter **Navigation** as the Band Code, **2** for the Band Sequence, and **LEFT** for the Band Alignment.
- 13 ♦ Save this Band.
- 14 ♦ Choose **Add** to add a fourth and final band.
- 15 ♦ Enter **tEditBand** as the Band Code, **3** for the Band Sequence, and **LEFT** for the Band Alignment.
- 16 ♦ Save this Band.

Your new SmartToolbar is now complete. You can exit the Toolbar Designer and add your Toolbar to the window, as described in the next section, so that you can test it.

12.4 Menu translations

An integral part of the globalization functionality of Dynamics up to now is the ability to translate any kind of widget found on an object.

Previously, the only way to display a menu item in your preferred language (if it is not EN-US) was to create a menu structure. You would have to duplicate the English language structure, substituting translated labels and descriptions.

With the new menu translation tool, exclusively available to the Toolbar and Menu Designer, you are now able to translate menu items easily without having to deal with standard translation methods.

Source language

The introduction of a source language enables you to determine the default language of a menu item. Due to the global nature of Dynamics, the language in which an application is ‘developed’ is the source language, so no translation is necessary.

Translating a menu item

The Toolbar and Menu Designer has been modified to accommodate the translation of menu items. The newly added source_language_obj field has been updated to get its value from the source language set for the current user’s profile.

Translation will not be allowed if there is not at least one record in the source language, thus adding a new menu item will force you to enter the menu item in the source language.

A new Translate button has been added to allow the translation of existing menu items. This button invokes the window in [Figure 12–20](#).

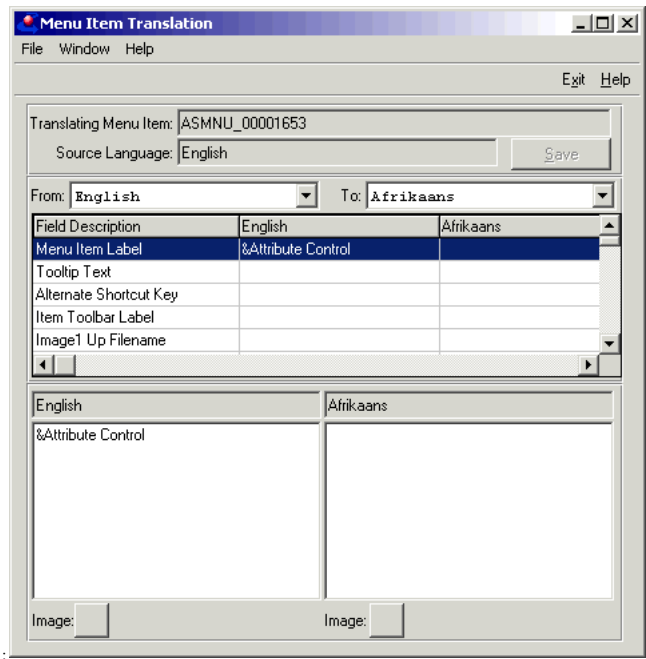


Figure 12–20: Menu Item Translation main window

Follow these steps:

- 1 ♦ Select the **From** language and the **To** language.
- 2 ♦ Select a row in the browser for the text element you want to translate.

- 3 ♦ Complete the text editors for the fields required. Alternatively, fill in the text in the updateable browser. Both options have been included to give users a choice, as text editors are usually more suitable for long descriptions.
- 4 ♦ Click **Save**.

12.4.1 Adding a toolbar to an application window

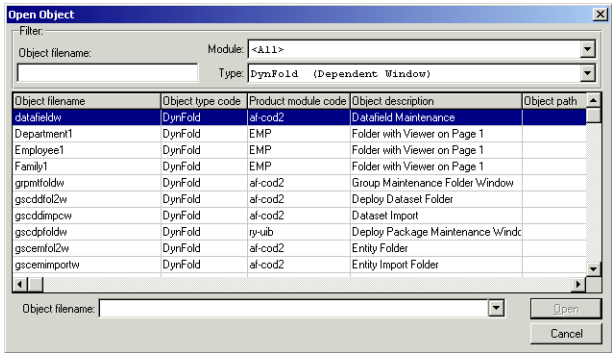
There are several different ways to add your Toolbar to a Progress Dynamics application window:

- Define a new Layout that uses your Toolbar as a standard component of the layout. In this way the Toolbar will be a part of any window built from that layout. For more information on creating layouts, see [Chapter 8, “Using the Progress Dynamics Container Builder.”](#)
- Modify an existing dynamic window to use your Toolbar or specify your toolbar when you create a window based on a standard Layout.
- Use the SmartToolbar Instance Attribute dialog box to assign the Toolbar. The following section describes how to add the Toolbar to a dynamic window.

12.4.2 Adding your toolbar to a dynamic window

Follow these steps to add your toolbar to a dynamic window:

- 1 ♦ In the AppBuilder main window, choose the **Open Object** button. The Open Object dialog box appears:

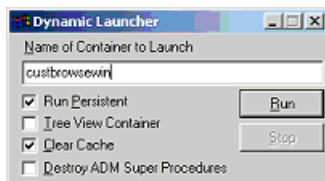


- 2 ♦ Select your dynamic window from the Open Object browser. This example uses the Customer Maintenance window built as part of the [Getting Started with Progress Dynamics](#) tutorial. You can use that folder window or any other window with a Toolbar and a Viewer for updating a record.

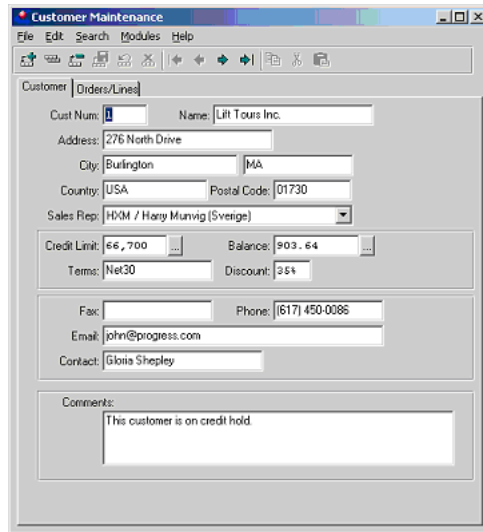
- 3 ♦ To use the *Getting Started with Progress Dynamics* example, select **custfoldwin** by double-clicking on the selection or by choosing the **OK** button. This brings up the design window:



- 4 ♦ Double-click on the design window to display the Container Builder.
- 5 ♦ Select the toolbar from the browser, then choose the **Lookup** button next to the Object field.
- 6 ♦ Select the **Tutorial Toolbar** from the Lookup browser window.
- 7 ♦ Choose **Save** to register your selection.
- 8 ♦ Choose **Exit** to exit the property sheet.
- 9 ♦ In the AppBuilder, choose **Save** to save your window.
- 10 ♦ Enter the name of the window to run. In this example, the Customer Maintenance window cannot be run directly, because it expects to get a Customer key as input. To test it, launch the browse window custbrowsewin from the Dynamic Launcher:



- 11 ♦ Select any record in the browser by double-clicking on it, to bring up the Maintenance window:



The screenshot shows a 'Customer Maintenance' window with a menu bar (File, Edit, Search, Modules, Help) and a toolbar. The 'Customer' tab is selected, showing the following fields:

- Cust Num: 1
- Name: Lift Tours Inc.
- Address: 276 North Drive
- City: Burlington, MA
- Country: USA, Postal Code: 01730
- Sales Rep: HDM / Harry Munvig (Sverige)
- Credit Limit: 66,700, Balance: 903.64
- Terms: Net30, Discount: 3.5%
- Fax: , Phone: (617) 450-0086
- Email: john@progress.com
- Contact: Gloria Shepley
- Comments: This customer is on credit hold

The Cut/Copy/Paste buttons on the Toolbar are not enabled. The same will be true if you drop down the Edit menu.

The reason for this is that the procedure that runs from these Items has not been created. Remember that when you created the Cut, Copy, and Paste Items, you specified an Action Type of RUN, and a procedure to run named EditAction. You have to create this procedure and associate it with the Toolbar in your window before the actions will work.

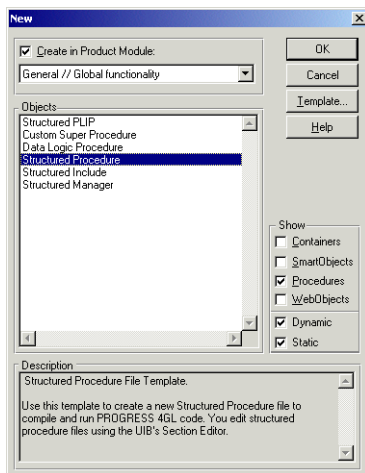
The SmartToolbar procedure recognizes that the supporting procedure cannot be found and disables the buttons automatically.

12.4.3 Defining a custom super procedure

If you define dynamic, logical objects in your Progress Dynamics application, they exist only as records in the Repository database. Therefore, you have no source procedure where you can write custom code that is needed by the dynamic object. Progress Dynamics handles this situation by letting you define a custom procedure containing code needed by a dynamic object. This file is referred to as a custom super procedure because at run time it runs as a persistent procedure and is made a super procedure of the object it supports. In this way, any RUN statement done in the dynamic object will find the supporting code in the super procedure and execute it there. Because this Toolbar example uses custom code to support some of its actions, one specific example of the technique is provided here.

To create the procedure where you can write the EditAction procedure, follow these steps:

- 1 ♦ Create a new structured procedure in the AppBuilder. So that the procedure is registered in the Repository, specify an appropriate Product Module to be associated with it:



- 2 ♦ In the wizard for the Structured Procedure, provide the name, description, and purpose for the procedure, which will be added to the comments section at the beginning of the procedure as internal documentation.
- 3 ♦ When you complete the wizard, choose the **Edit** icon in the AppBuilder. This displays the AppBuilder's Section Editor.
- 4 ♦ Select **Procedures** from the Section drop-down list.

- 5 ♦ Choose the **New** button to create a new procedure, providing **EditAction** as the name.

The following shows code for the EditAction procedure, which uses its input parameter (which you specified when you defined the Actions for the Cut, Copy, and Paste Items in the Toolbar Designer) to handle all three operations:

```

/*-----
Purpose:   Perform cut/copy/paste operations
Parameters: pcAction = "Cut", "Copy", or "Paste"
Notes:
-----*/

DEFINE INPUT  PARAMETER pcAction AS CHARACTER  NO-UNDO.

DEFINE VARIABLE lTextSelected AS LOGICAL      NO-UNDO.
DEFINE VARIABLE lOK           AS LOGICAL      NO-UNDO.

CASE pcAction:
  WHEN "Cut":U THEN
    DO:
      lTextSelected = FOCUS:TEXT-SELECTED NO-ERROR.
      IF lTextSelected THEN
        DO:
          lOK = FOCUS:EDIT-CUT() NO-ERROR.
          IF lOK THEN
            APPLY "VALUE-CHANGED":U TO FOCUS.
          END.
        END.
      END.
  WHEN "Copy":U THEN
    DO:
      IF FOCUS:TEXT-SELECTED THEN
        lOK = FOCUS:EDIT-COPY() NO-ERROR.
      END.
  WHEN "Paste":U THEN
    DO:
      IF FOCUS:EDIT-CAN-PASTE THEN
        DO:
          lOK = FOCUS:EDIT-PASTE() NO-ERROR.
          IF lOK THEN
            APPLY "VALUE-CHANGED":U TO FOCUS.
          END.
        END.
      END.
    END.
  END CASE.
END PROCEDURE.

```

- 6 ♦ Enter this code and save your procedure where it will be found in your Propath.
- 7 ♦ Open your dynamic window (custfoldwin or whatever window you are using) using the Open Object dialog box again.

- 8 ♦ Enter the name of your procedure (including a relative pathname if needed) in the field labeled Custom Super Procedure.

NOTE: You are associating the procedure with the window, not directly with the Toolbar. This is because Toolbar Item actions that run internal procedures look for them in the container, not in the Toolbar object itself, if there is no specific action link to another object.

- 9 ♦ Save the dynamic window and relaunch it (or the browse window that precedes it) via the Dynamic Launcher. Now you will see the Cut, Copy, and Paste Items enabled (both as menu items and as toolbar buttons), and you can use them to edit, for example, the Comments field for the Customer.

12.4.4 Attaching a custom super procedure to all windows

Instead of defining a custom super procedure for a particular window, you can also associate it with **all** the Progress Dynamics windows in your application. The same principle applies to other types of objects as well. For more information on this topic, see the chapter on extending classes in the *Progress Dynamics Programming Handbook*.

12.4.5 Editing toolbar properties

You manage the dynamic properties of any toolbar from the Dynamic Properties sheet, accessible from the Container Builder or the Toolbar and Menu Designer.

Defining Progress Dynamics Application Security

This chapter covers applying security to application components, such as windows, menu items, folder tabs, database tables, individual records, or ranges of data. Progress Dynamics provides comprehensive support for defining and maintaining users and their passwords. The framework also supports applying access rights to your application based on user ID, on group membership, or on the login company (or equivalent organizational entity).

This chapter includes the following sections:

- [Introduction](#)
- [Using the Security Control tool](#)
- [Menu security](#)
- [Menu item security](#)
- [Action security](#)
- [Field security](#)
- [Data range security](#)
- [Data security](#)
- [Login company security](#)

13.1 Introduction

For developers, there are four key concepts to keep in mind as you design your application security:

- Security model
- Security user types
- Security structures
- Security allocations

13.1.1 Security model

The first step is to choose which security model is right for your application. Progress Dynamics performs access control by either revoking access or by granting access, but not both. All application functionality in a single Dynamics repository must use either the **revoke security model** or the **grant security model**.

With a revoke model, a user or group has unlimited access rights unless you specifically revoke rights from them. With the grant model, a user or group has no access rights unless you specifically grant rights to them. You must choose one or the other—you cannot have a mixed security model. The default is a revoke model. See the [Progress Dynamics Administration Guide](#) for a discussion on which model is better suited for your Progress Dynamics application. The examples in this chapter mostly demonstrate the revoke model.

NOTE: If you define security using one security model and then decide you want to use the other security model, Progress Dynamics asks you to confirm that you want to change models and delete your current security allocations.

13.1.2 Security user types

User types are the *who* of the security system. The Progress Dynamics framework provides a built-in mechanism for defining several user types. Security user types are authorized application users (users), abstract users (profile users), and groups of such users (groups). The types are explained below:

- **User** — An individual authorized to use your application.
- **User profile** (retained for backward compatibility) — A preset security definition that may be applied to any number of users as a base security definition for that user. You can modify the security definitions of actual application users that have a profile applied. Changes made to the actual user do not alter the associated profile or alter the security definition of other users with that profile. If you change the user profile, existing users associated with it do not automatically inherit the changes. You control the propagation of security allocations to the profile users. The functionality of user groups has been largely replaced in the new security model by groups.
- **Group** — A collection of users and other groups aggregated to efficiently assign common security definitions. For example, you may use a group to represent the users who have the role of application administrators. Thus, you can alter the security definition of all administrators by altering the security definition of the group. **Dynamics security groups are not hierarchical.** For example, suppose a new employee is part of the Trainee group, which does not allow access to customer information, but the new employee is also a member of the Executive group, which does allow this access. In this case of direct conflict between two settings for the exact same access right, the security system grants the user the least restrictive access rights it finds associated with the user. There is no notion of rights defined in one group overriding rights defined in another group.

Group security allocations are inherited by users at session startup, but they are not physically duplicated under each user. This is different from user profiles. Because of this difference, you do not need to take any further action after making a change to group security to have it affect all appropriate users.

NOTES:

- You can use security groups to consolidate other security groups. If a security group does not have any security definitions of its own, Dynamics does not include it in security checks. This approach minimizes overhead and maximizes performance. Once a group has one or more security definitions, Dynamics includes the group in its security checking.
- When allocating security at the user level, the Security Control tool displays a radio set allowing the administrator to turn security on or off for the particular user. Security assigned in this way always overrides security allocations at the group level.

13.1.3 Security structures

Security structures are the *what* of the security system. They are application functionality you need to control in different ways, depending on the user. Each business security rule you want to create will associate users to structures within Dynamics. Here are the structures you can define security on:

- **Action** — Conceptually, actions are an abstraction for controlling visual objects in your application. For example, perhaps you want to restrict access to a particular tab in a tab folder window. Other possible targets include menu items and toolbar buttons. Actions also include some built in actions (Add, Cancel, Copy, Delete, Modify, View). Security on these actions in particular context automatically prevent built-in framework behavior from providing access. For example, if you could normally double-click a browse row to bring up an update window, a restricted user would get a message and be able to see the update window in view-only mode. The structure can be set universally or at the product module level.
- **Container** — Containers are your main UI application objects and the security system is already aware of all containers. There is no need to create an abstract structure in the security system to represent them. Associate users with containers and effectively revoke or grant access to all the application functionality in the container.
- **Data** — Data security structures represent application-level security of your database records. Since database records are represented with the application as entities, data security is entity security. The security system is aware of application entities, so you do not need to create an abstract structure in the Security control tool to represent them. Data security does not come with preprogrammed behavior. You need to program the behavior that you require when a user has a data security allocation.
- **Data range** — Data range structures are purely abstract codes you create in the Security control tool to represent a data range that you want to modify depending on the user. When you associate the code with users, you specify the specific object, instance attribute, and from-to values. Data range security does not come with automatic behavior; you need to program the behavior that you require when a user has data range security allocations.

- **Field** — Field security structures identify database fields and application widgets you want to define additional security on. Normally, you define field-level security through your SDOs and SBOs (what is visible and what is not). Use field security when you want to vary your normal security by login company, for example. When you define a field security structure it applies globally across your application. You can narrow the restriction anyway you like when you define the structure. When you associate a field security structure with users, you specify the type of restriction (Full Access, Read Only, or Hidden).
- **Login company** — While a login company conceptually represents an organizational entity, and thus an aggregation of users, it functions more as a security structure than as a user type. Think of login companies as structures that groups and users are granted or restricted access to. Login companies, in effect, are the primary security structure because each security rule you create requires you to specify one or all login companies as part of the rule.
- **Menu item** — Any item appearing on a menu or toolbar. The security system is aware of all menu items, so there is no need to define an abstract security structure to represent them in the Security control tool.
- **Menu structure** — Any collection of menu item, called bands in Dynamics, that can appear on a menu or toolbar. The security system is aware of all bands, so there is no need to define an abstract security structure to represent them in the Security control tool.

13.1.4 Security allocations

Security allocations are the *how* of the security system. *Who* (user types) may use *what* (security structures) *how*, or, *in what manner* (security allocations)? Allocations associate users to structures, define key parameters of the association, and document the purpose of the association. At the user level, an allocation amounts to a security rule or access right.

Some allocations are effectively yes or no constructs with automatic behavior supplied by the framework, others provide useful ways to qualify access rights, and still others are abstract concepts that give you ways to hook custom programming into the security system.

Security allocations map one for one to the security structures. However, the following structures are automatically supplied by the framework, so there is no corresponding node in the Security maintenance node of the Security control tool: containers, data, menu items, and menus.

13.1.5 Resolving security

Progress Dynamics checks security in this order, from most to least specific:

- Progress Dynamics first searches for an allocation against the specified user for the specified login company. If it finds an allocation, Progress Dynamics uses it immediately.
- Next, Progress Dynamics searches for an allocation against the specified user in all login companies. If it finds an allocation, Progress Dynamics uses it immediately.
- Next, Progress Dynamics checks all groups the user belongs to for any applicable allocation. If Progress Dynamics finds more than one allocation, it applies the cumulative least restrictive security.
- Next, Progress Dynamics checks security allocations set up against all users. Within this category, it first checks for allocations against all users within the specified login company. Finally, Progress Dynamics checks allocations against all users for all login companies.

NOTE: Global security structures and security allocations are cached at session startup. For AppServer sessions, the agents need to be trimmed then added back if any global security structures or allocations are added or updated.

13.2 Using security documentation

Administrators and programmers set up and maintain security with the framework's Security Control tool. You can access the tool through the Security menu in the Progress Dynamics Administration window.

The *Progress Dynamics Administration Guide* describes the security functions that are system-wide settings or those that administrators use to define and maintain users, groups, and profiles. This chapter describes how to define security on application objects and apply them to specific users, groups, and application elements. [Table 13–1](#) summarizes which parts of the Security Control tool documentation you can find in the *Progress Dynamics Administration Guide* and which documentation you can find here.

Table 13–1: Security Control tool documentation*(1 of 2)*

Tree node	Description	Documented in
Security control	The topmost node of the tool's tree. Provides the administrator with system-wide settings for the application and the development environment.	<i>Progress Dynamics Administration Guide</i>
Security maintenance: <ul style="list-style-type: none"> Login companies Users Security groups User categories Users 	Lets you define and maintain users, groups, and organizational entities.	<i>Progress Dynamics Administration Guide</i>
Security maintenance: <ul style="list-style-type: none"> Actions Data ranges Fields 	Lets you define and maintain security for application objects.	This chapter
Security Allocation	Lets you define specific access rights for users, groups, or companies. You can define access based on an action, a field, or a data range. You can define access rights for any menu, menu item, container, or login company. You can also define access rights to specific database tables and even specific records within tables.	This chapter

Table 13–1: Security Control tool documentation

(2 of 2)

Security enquiry	Gives you several ways to search and filter existing security settings. Demonstrates how security allocations will be resolved at runtime.	This chapter
Security processing	Gives you the ability to create a new security group based on an existing user.	<i>Progress Dynamics Administration Guide</i>

13.3 Using the Security Control tool

The Security Control tool allows you to define user types, security structures, and security allocations, which set the parameters of what administrators and users often call access rights.

To access the Security control tool:

- 1 ♦ Select **Tools**→**Administration** from the AppBuilder window.
- 2 ♦ Select **Security**→**Security control** from the Administration window.

- 3 ♦ To see all the Security control functions, expand the nodes of the tree, as shown in [Figure 13–1](#).

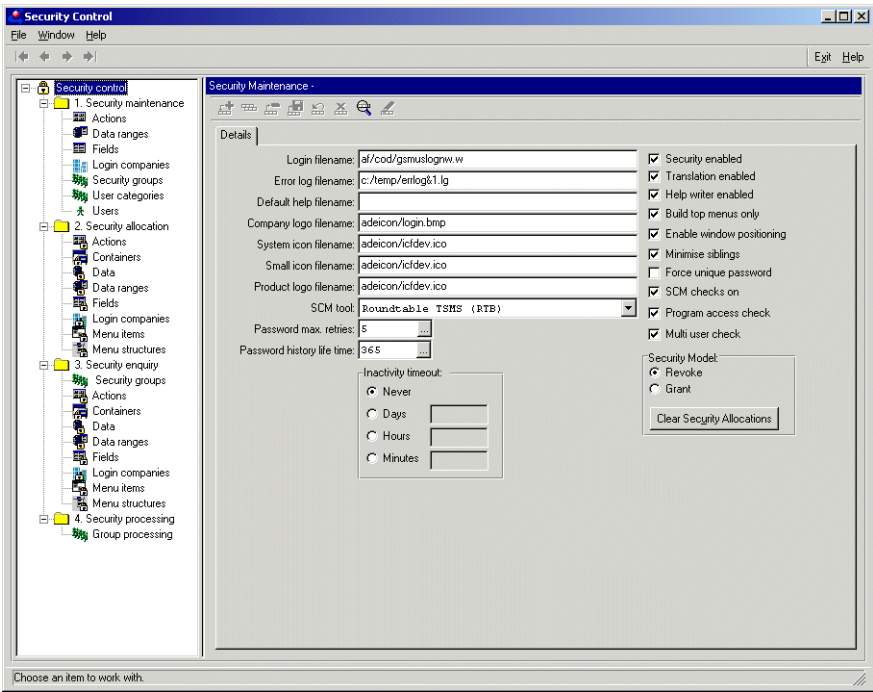


Figure 13–1: Security Control tool

There are five major parts to the Security Control tool, as defined in [Figure 13–2](#).

Table 13–2: Security Control tool parts (1 of 2)

Node	Purpose
Security control	The top node. All the global settings for security are found on this panel. See the Progress Dynamics Administration Guide for more information.
Security maintenance	Create and modify login companies, groups, users, and user categories as well as security structures for actions, data ranges, and fields. Security structures are security elements that identify existing application objects (for example, fields) to the security system or create application objects used by the security system (for example, data ranges).

Table 13–2: Security Control tool parts (2 of 2)

Node	Purpose
Security allocation	Associate users or groups with security structures to restrict or grant access to application functionality.
Security enquiry	<p>The security of an enterprise application can be extensive. The Security enquiry node is the best way for you to search for pertinent details before you make changes and to confirm the effect of your changes after they are made.</p> <p>With security enquiry, you can select a particular user and see how security allocations are resolved for that user. This is especially useful when the user belongs to multiple security groups or combined groups, or when the user is logged in under a particular login company.</p>
Security processing	<p>Provides an option to convert a user security definition into a group definition. The process is as follows:</p> <p>Create a new group.</p> <p>Copy all security allocation of the user to the new group.</p> <p>Remove all security allocations from the user.</p> <p>Assign the user to the new group.</p>

When using the Security allocation nodes, you have several options for specifying who the security applies to and the login company context for those users. The effects of specifying login companies in allocations is described in [Table 13–3](#).

Table 13–3: User names and login company security

User or group name entered	Login company entered	Security applies to...
Yes	No	The specific user or all members of the specified group (depending on your selections) regardless of what login company he or she logs in under (including no specific login company at all).
No	Yes	All users who log in under that login company.
Yes	Yes	The specific user or all members of the specified group (depending on your selections) when logged in under that company.

13.4 Menu security

[Chapter 12, “Using the Toolbar and Menu Designer”](#) showed you how menus and toolbars are divided into Bands, that is, groups of related items or toolbar buttons. The term menu structure refers to these bands.

Security allocations of this type have the following effects:

- **Revoke model** — Prevents a user or a group from seeing a menu or toolbar. By default, Dynamics allows all users to see the menus and toolbars of the containers they have access to.
- **Grant model** — Allows a user or group to see a menu or toolbar. By default, Dynamics prevents all users from seeing menus and toolbars in a container.

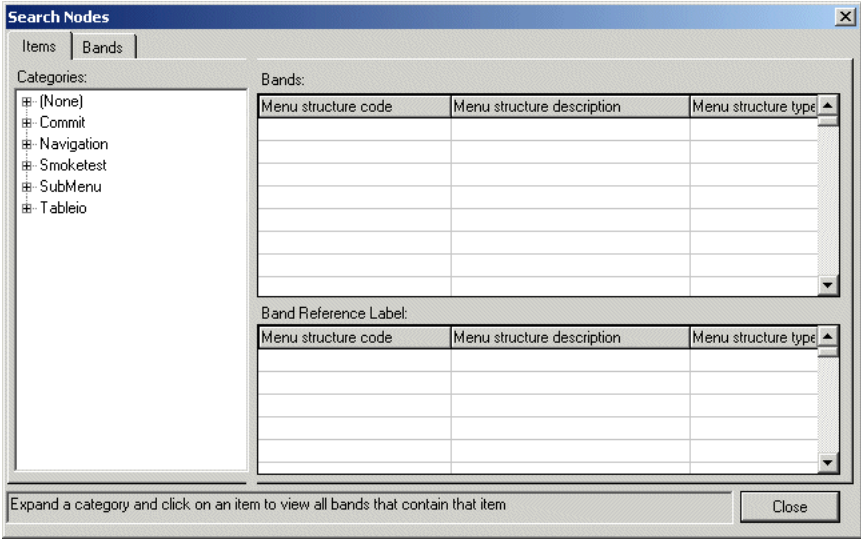
The security system is aware of all menu structures, therefore you do not have to create a structure in the Security Control tool to represent them.

13.4.1 Creating menu security allocations

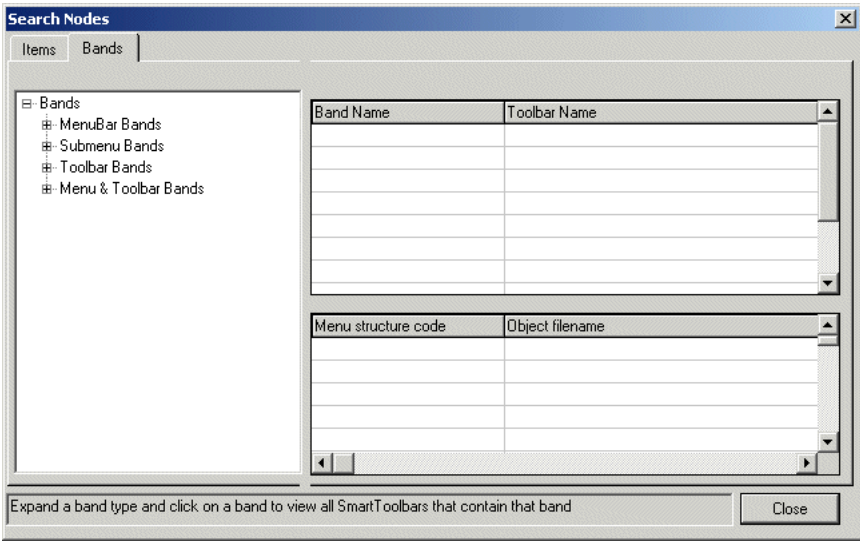
Suppose you want to restrict access to the Security menu in the Progress Dynamics Administration menu for a user named Anthony. You may need to identify the menu structure before you can create an allocation.

To determine the name of a target menu structure:

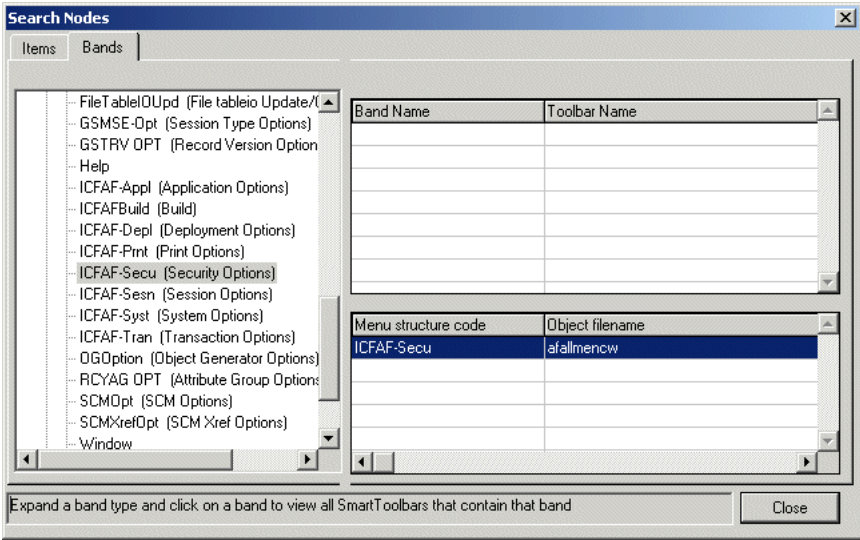
- 1 ♦ Open the Toolbar and Menu Designer tool, then choose the **Lookup** icon in the toolbar:



- 2 ♦ Because you are looking for a menu structure, or Band, select the **Bands** tab. The band you are looking for is a top-level menu item that acts as the parent for all the security items. This is a SubMenu band:



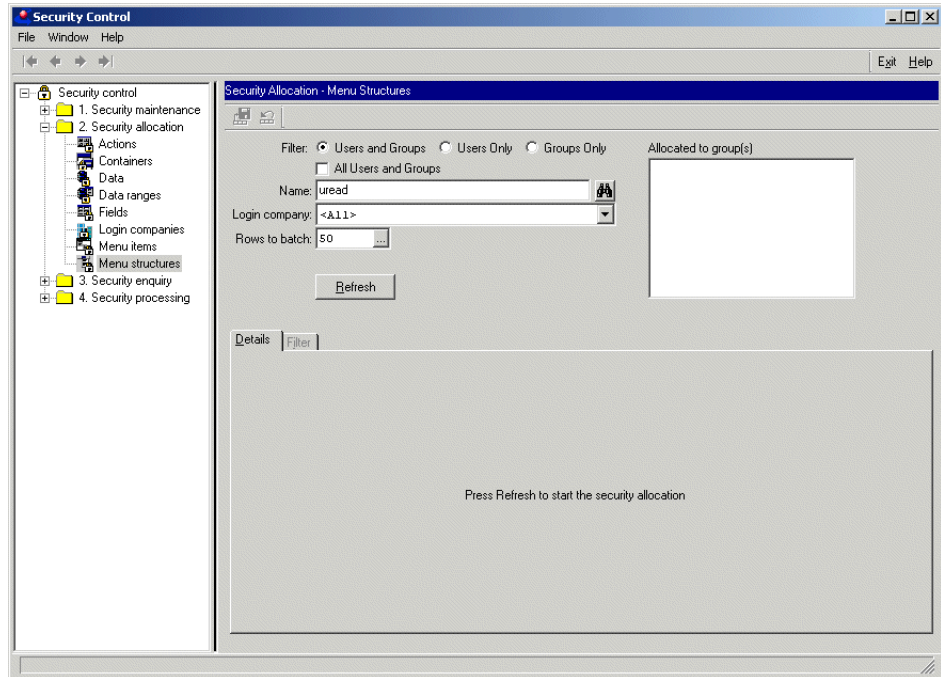
- 3 ♦ Open the **SubMenu** node in the TreeView and locate the entry with the description **Security Options**.
- 4 ♦ Select it. Its key code displays on the right, along with all the containers where it is used:



In this case, the menu structure identifier, or Band Code, is ICFAF-Secu, and the one place where it is used is in the container afa11mencw. This is the logical object name of the Progress Dynamics Administration window. If you need more information about a logical object such as afa11mencw, open it using the **Open Object** button in the AppBuilder, or you can examine it in the Repository Maintenance tool.

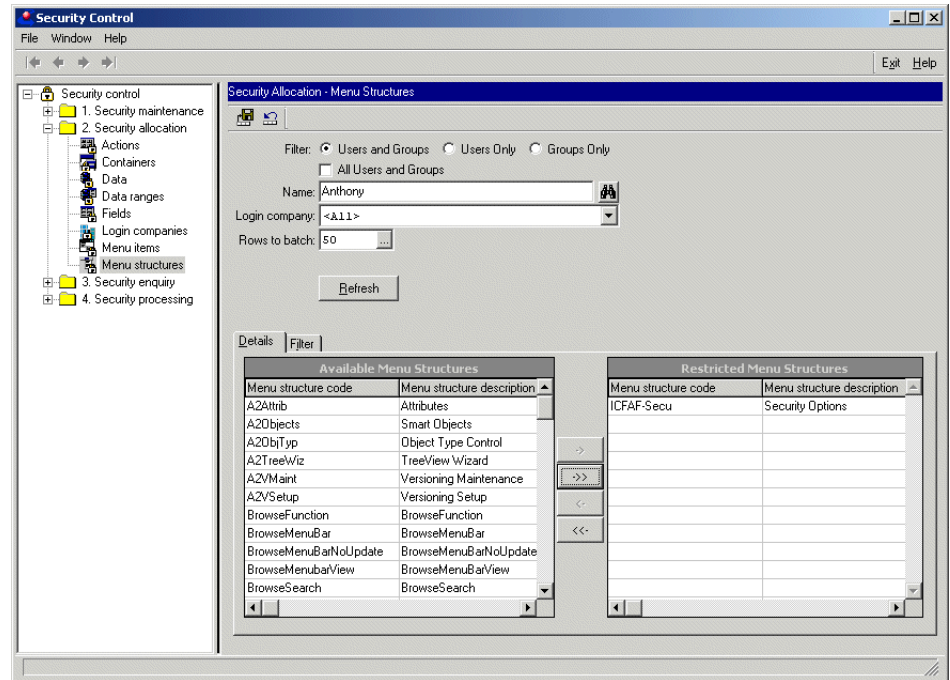
To apply security to a menu structure:

- 1 ♦ Open the **Security allocation** node and the **Menu structures** subnode:

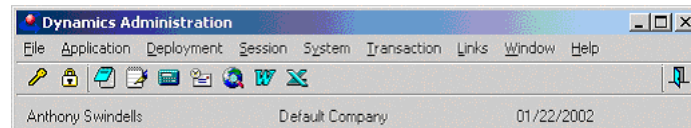


- 2 ♦ Specify the user or group and login company for this allocation. For the example, define the allocation for the user name **Anthony** and Login company **<All>**.

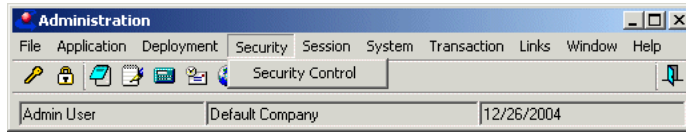
- 3 ♦ Choose the **Refresh** button to see a list of menu structures, or use the Filter tab to identify it.



- 4 ♦ Select the **ICFAF-Secu** row in the browser and click→to define the allocation.
- 5 ♦ Click **Save**:
- 6 ♦ To see the effects of your change, exit Dynamics Development.
- 7 ♦ Restart Dynamics Development and log in as the user Anthony.
- 8 ♦ Choose Tools→Administration. You see the Administration window as Anthony will see it, without the Security structure:



- 9 ♦ Repeat steps 6 through 8, logging in as Admin. Notice the normal Administration window with the Security menu:



This section can use the Administration window as an example of security settings because this window, like most Progress Dynamics framework tools, is implemented in the framework itself.

13.5 Menu item security

This section describes how to define security on the individual items you defined in the Toolbar and Menu Designer. These can be menu items, toolbar buttons, or a single item visualized as both. The Security Control tool, however, uses the term “Menu items” to refer to all of these.

Security allocations of this type have the following effects:

- **Revoke model** — Prevents a user or a group from selecting a menu item or toolbar button. The item appears grayed out. By default, Dynamics allows all users to select all menu items and buttons.
- **Grant model** — Allows a user or group to see a menu or toolbar. By default, Dynamics prevents all users from seeing menu items and toolbar buttons.

The security system is aware of all menu items, therefore you do not have to create a structure to represent them in the Security Control tool.

13.5.1 Creating menu item security allocations

To define an allocation for a menu item:

- 1 ♦ Open the **Security allocation** node and the **Menu items** subnode.
- 2 ♦ Specify the user, group, and login company information for the allocation.
- 3 ♦ (Optional) Set a filter. In the security system, there are usually long lists of items to scroll through. Using the standard Filter tab can speed up your work.

4 ♦ Click **Refresh**.

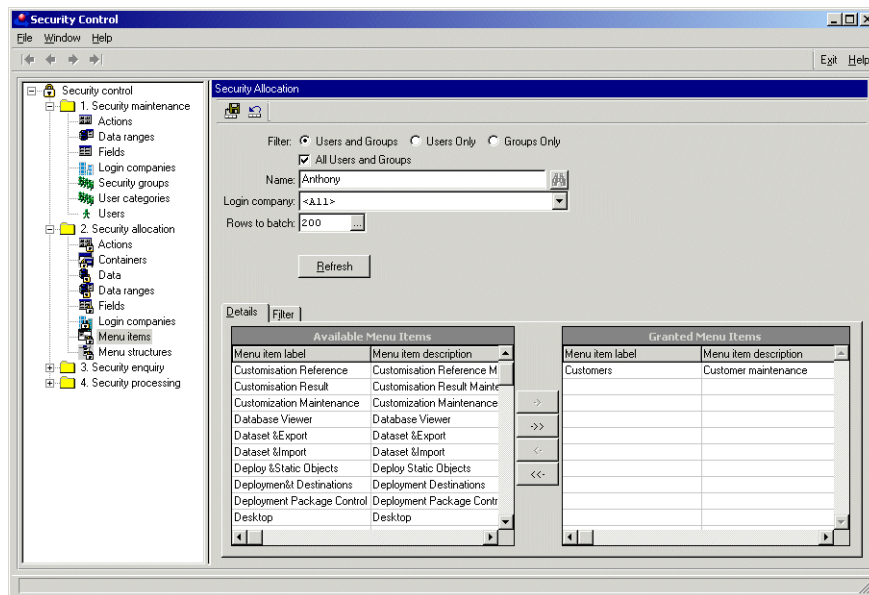
NOTE: When the browser displays more items than specified in the Rows to Batch field, scrolling to the end of the browse causes the next batch to be retrieved. Use a filter to reduce the list, or increase the size of Rows to Batch to fetch more values.

The browse window populates with available items.

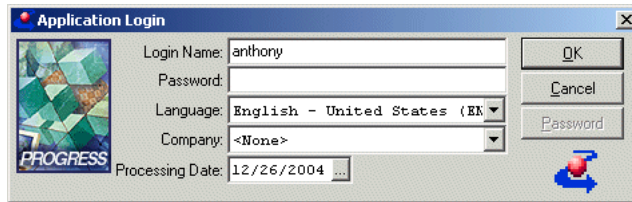
NOTE: While browsing, keep in mind that the item label and item description displayed are not necessarily unique. The unique Item Reference code is not displayed here. You can set a restriction for all occurrences of the label if you want the restriction to apply everywhere it is used. Otherwise, you might want to consider using action security instead.

NOTE: Another consideration when browsing is that by default the Items are sorted by Label. The ampersand (&) shortcut character is part of the Label, and sorts before alphabetic characters.

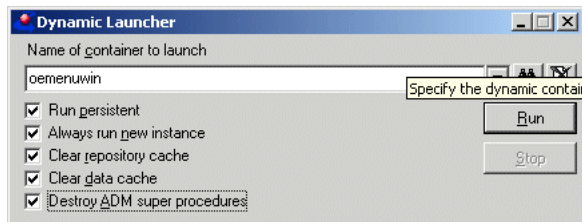
This example uses the objects from the *Getting Started with Progress Dynamics* tutorial. That sample application has a Menu Controller window with a standard Menu & Toolbar object, with an OrderEntry Band added to it. The Band has two Menu Items, Customers and Orders. Here you are applying a restriction for the user Anthony for the Customers item. Here the Customers item is used only in the oeband of the Menu Controller window that is in the sample application:



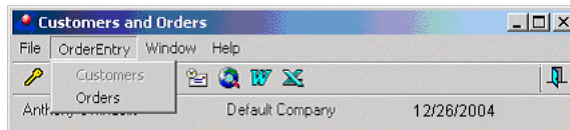
- 5 ♦ Select **Customers** from the Available Menu Items browser and click the -> button.
- 6 ♦ Click **Save**.
- 7 ♦ Start a new session:



- 8 ♦ Use the Dynamic Launcher to test the sample application window, oemenuwin.
- 9 ♦ Check the **Destroy ADM Super Procedures** check box to force a re-cache of menu information, in order to see the effects of the menu change:



- 10 ♦ Drop down the **OrderEntry** menu to see that the Customers item is disabled. Note that menu items that are restricted are disabled rather than being completely hidden, unlike Menu Structures (Bands):



13.6 Container security

Container security allows you to apply security to your visual application containers. You do not have to set security both on a container and the items that launch it. If you set security for a container, Dynamics automatically disables any menu items launching that container.

Security allocations of this type have the following effects:

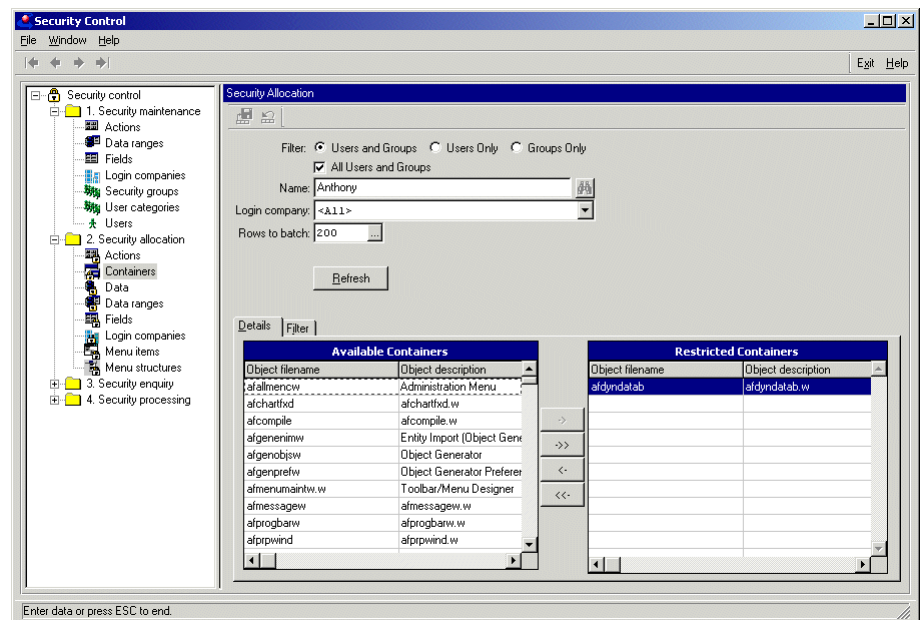
- **Revoke model** — Prevents a user or a group from accessing a container. By default, Dynamics allows all users to see all containers.
- **Grant model** — Allows a user or group to see a container. By default, Dynamics prevents all users from seeing containers.

The security system is aware of all containers, therefore you do not have to create a structure in the Security Control tool to represent them.

13.6.1 Creating container security allocations

To create a container security allocation:

- 1 ♦ Open the **Security allocation** node and the **Containers** subnode.
- 2 ♦ Specify the user, group, and login company information for the allocation.
- 3 ♦ Click **Refresh** the browse window populates with available containers:



- 4 ♦ Select the desired container and click the → button.
- 5 ♦ Click **Save**.

13.7 Action security

Actions let you define security at the level of an individual visual object in your application or in the framework itself. This can be a toolbar button or a tab on a folder. It can also be the built in actions Add, Cancel, Copy, Delete, Modify, and View. You can define security for all occurrences of an action or you can qualify it for a particular product module, container, or instance attribute (or any combination of these).

Security allocations of this type have the following effects:

- **Revoke model** — Prevents a user or a group from accessing a defined action. By default, Dynamics allows access to all actions.
- **Grant model** — Allows a user or group to access a defined action. By default, Dynamics prevents all users from accessing actions.

Defining actions for items (menu items and toolbar buttons) is an alternative to defining menu item security. Use actions if you want to apply the security only to a specific container or module. In addition, the effect of applying menu item security is different from applying action security. Menu item security disables items, but they are still visible. Action security removes them from view.

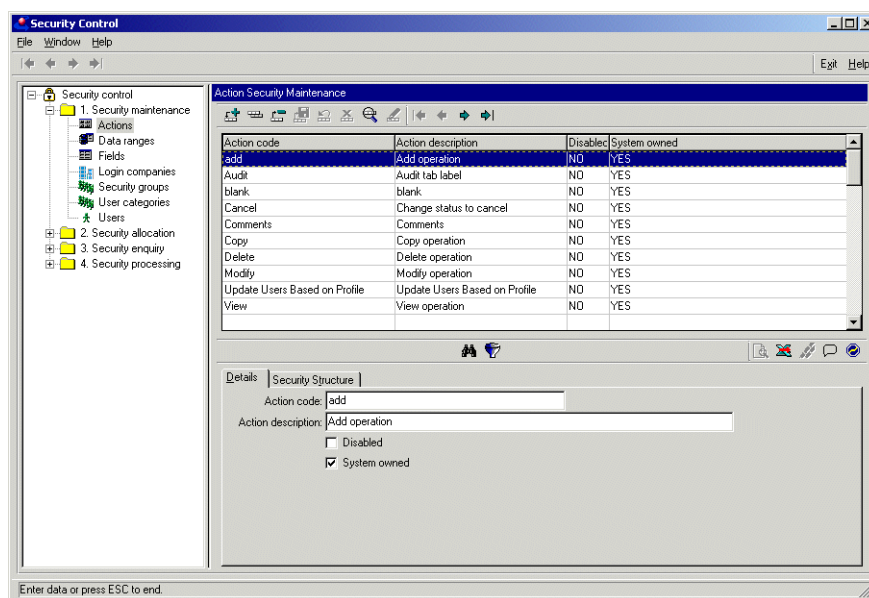
13.7.1 Creating action security structures

Actions are an abstract concept, so you need to set up a structure for them before you can allocate security. Setting up security structures means two things:

- Create a new action code to secure a particular item, like a tab label.
- Select an existing action code (including system included actions) and optimally add another scope (module, object, or attribute) where the structure applies. You add these scopes one at a time

To create a new action code:

- 1 ♦ Open the **Security maintenance** node and the **Actions** subnode:



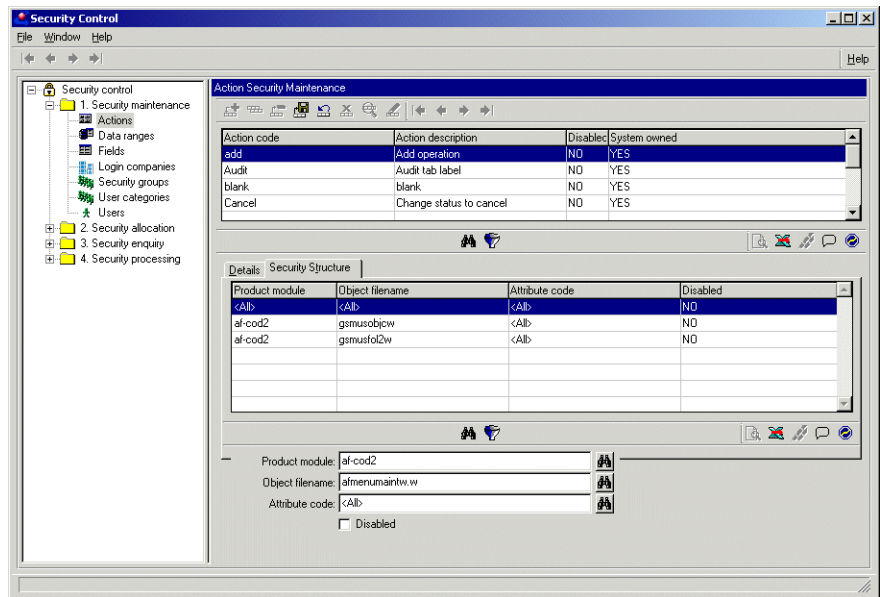
- 2 ♦ Click the **New** toolbar button.
- 3 ♦ The action name needs to be the same as the security token of the item being secured. Enter that name in the **Action Code** field.
- 4 ♦ Document the purpose of your new action in the **Description** field.
- 5 ♦ Click the **Save** button.

By default the new action code applies to all modules, objects, and attributes. You will probably want to change this to a more limited scope. The second part of setting up the security structure involves indicating where the action security can apply.

To define action security scope:

- 1 ♦ Select an action from the browse. The lower browse displays each place where the selected action security currently applies.
- 2 ♦ Click **New**.

- 3 ♦ Select the **Security structure** tab:



- 4 ♦ At the bottom of the tab, define one combination of module, object, and attribute.
- 5 ♦ Click **Save**.
- 6 ♦ Repeat to add more locations to apply the security.

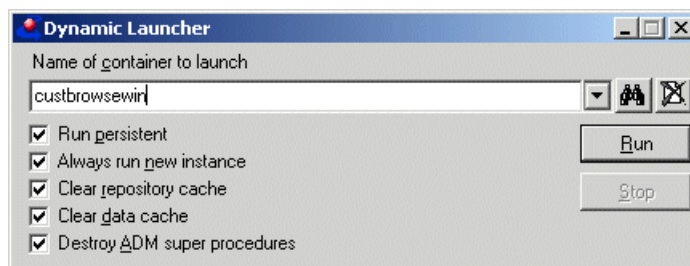
13.7.2 Restricting access to folder tabs

Follow these steps to create action security for a folder tab in the sample application:

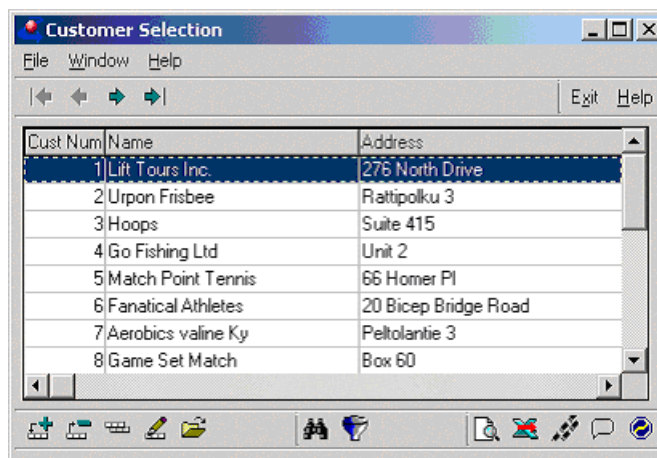
- 1 ♦ Open the action **Security maintenance** node and the **Actions** subnode.
- 2 ♦ Add an action for the code Orders/Lines. If you built the *Getting Started with Progress Dynamics* application just as instructed, this is the label of the second tab in the Customer maintenance window, custfoldwin. Otherwise, you should use whatever the security token is, or else define a similar restriction for some other tab folder available to you.

NOTE: When you are assembling windows in the Container Builder, there is a field where you can define a Security action for a tab folder page. Even though it is not actually displayed, the action is initialized to be the same as the tab label, just as for menu items. Use this default unless there is a need to define something different.

- 3 ♦ Enter **Order/Lines** in the Action Code field, then enter a description.
- 4 ♦ Select the **Security Structure** tab to see the default structure for the new action.
- 5 ♦ Open the **Security allocation** node and the **Actions** subnode.
- 6 ♦ Select user **Anthony** and click **Refresh**.
- 7 ♦ Double-click on the **Restricted** field for the new OrdersLines entry, then Save it. Note that these entries are identified and sorted by their description, not by the code itself.
- 8 ♦ Re-Logon as **Anthony**.
- 9 ♦ Launch the Customer browse window that leads to the folder window. Because the security information for actions is stored in the standard client-side Repository cache, you just have to select **Clear Cache** to see the results of your change:



- 10 ♦ In the Customer Selection window, double-click on a record to bring up the maintenance folder window:



When the maintenance window appears, you see that the Orders/Lines tab is disabled:

The screenshot shows a window titled "Customer Maintenance" with a menu bar (File, Edit, Search, Modules, Help) and a toolbar. Below the toolbar, there are two tabs: "Customer" (active) and "Orders/Lines" (disabled, indicated by a greyed-out label). The "Customer" tab contains the following fields:

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

If the tab for Page 1 was disabled, the framework would automatically suppress that page and select the first available (enabled) page of the folder. If **every** tab in a folder is disabled for a user, then it will not be brought up at all.

13.8 Field security

When you create SDOs, Viewers, and other application objects, you define which fields are updatable and which are not. This provides a default security behavior for your application. However, you might want to modify this default behavior for particular users, groups, or companies. To do this, you define field security.

NOTE: Progress Dynamics does not support security against simple text widgets. If you want to secure static text you can create a fill-in widget, enter the text as an initial value, and set the VIEW-AS-TEXT property to true.

Security allocations of this type have the following effects:

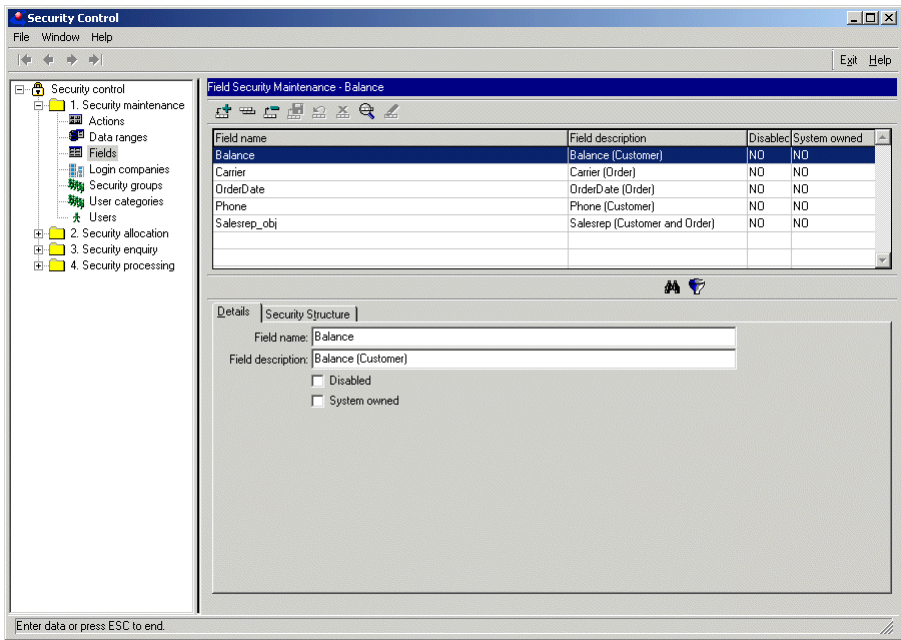
- **Revoke model** — By default, Dynamics allows access to all fields defined in the security system. Depending on the mode of field security (Full-Access, Read-Only, Hidden), the user will experience some limitation with the field.
- **Grant model** — Allows a user or group to interact with an application field that has been set up in the security system. By default, Dynamics limits all users access to the field, depending on the mode of the field security. This does not affect all application fields; only those that have a security structure defined.

13.8.1 Creating field security structures

No field security is possible until you provide the security structure for it. Then you can create restrictions by associating that structure with users, groups, or companies.

To create security structures for fields:

- 1 ♦ Select **Security maintenance** node and the **Fields** subnode:



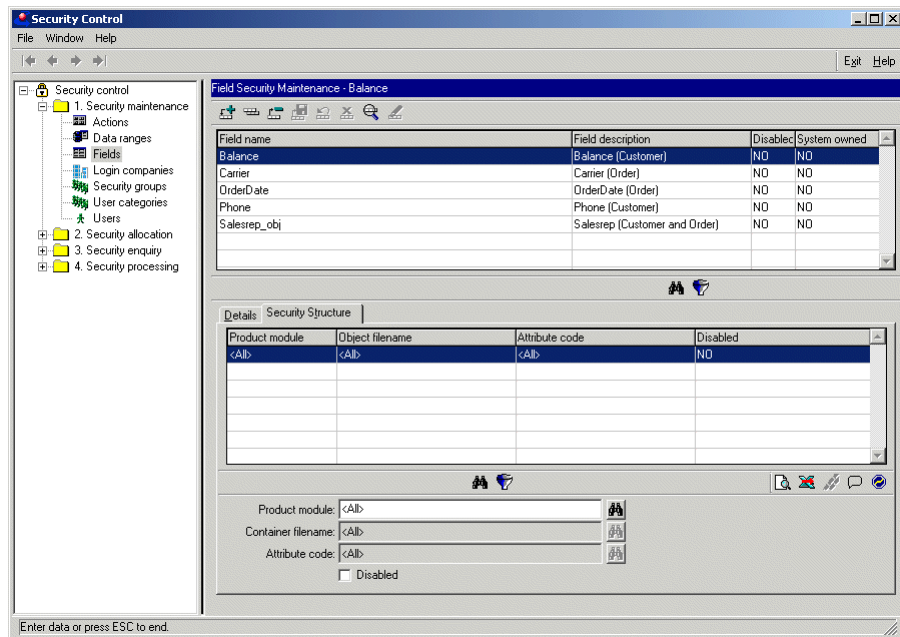
- 2 ♦ Click the **Add** button to create a new structure.

- 3 ♦ Type the field name in the **Name** field and document the purpose of the security in the **Description** field. This should be a valid field name (not qualified by database or table name). In this example, the Balance field from the Customer table is used.
- 4 ♦ Click the **Save** button.

As with actions, the first structure for a field applies to all modules, containers, and attribute values by default. You can then use the **Security Structure** tab to define more specific structures.

To define the security structure:

- 1 ♦ Click the field name you want to modify in the browse. The second browse populates with existing security structures for the field.
- 2 ♦ Select the **Security Structure** tab.



Field name	Field description	Disabled	System owned
Balance	Balance (Customer)	NO	NO
Carrier	Carrier (Order)	NO	NO
OrderDate	OrderDate (Order)	NO	NO
Phone	Phone (Customer)	NO	NO
Salesrep_obj	Salesrep (Customer and Order)	NO	NO

Product module	Object filename	Attribute code	Disabled
<All>	<All>	<All>	NO

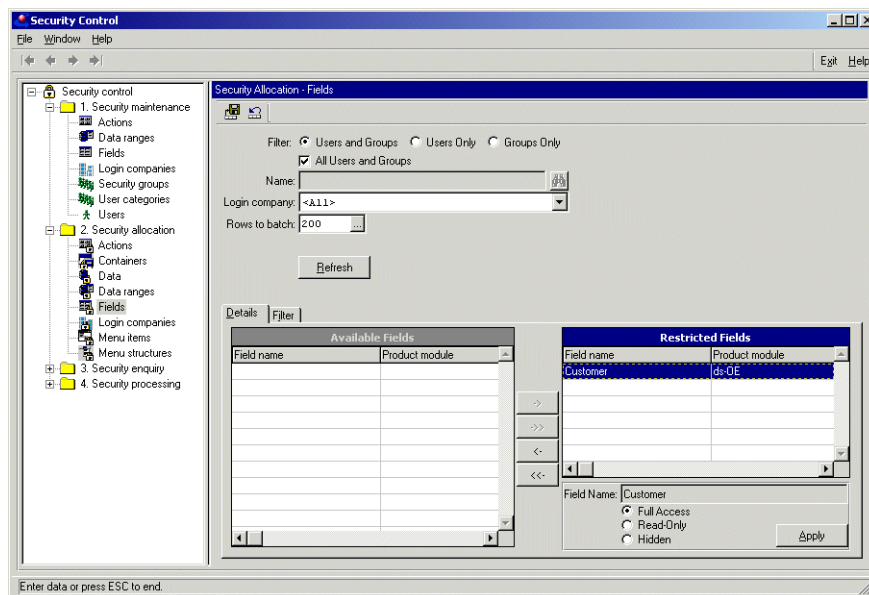
Product module: <All> Container filename: <All> Attribute code: <All> ☐ Disabled

- 3 ♦ Click the **Add** button.
- 4 ♦ Use the **Product module**, **Container filename**, and **Attribute code** fields to specify a new combination.
- 5 ♦ Click **Save**.

13.8.2 Creating field security allocations

To create an allocation on a field security structure

- 1 ♦ Open the **Security allocation** node and the **Fields** subnode.
- 2 ♦ Specify the appropriate combination of user or group, and login company.
- 3 ♦ Click **Refresh**.



- 4 ♦ Select the target field in the left browse and click the→button.
- 5 ♦ Select the field in the right browse.

6 ♦ Choose the security level.

For each structure, there are three possible values for the Access Level:

- **Full Access** — No restriction is applied.
- **Read Only** — The field is disabled for update.
- **Hidden** — The field and its label is hidden from the user or company.

7 ♦ Click **Save**.

When a non-restricted user views the Login company node and selects a record, all the fields on are enabled.

Suppose user Anthony is restricted from viewing the Login Company Name field. When Anthony views the same frame, the Login Company Name field is hidden from view.

13.9 Data range security

The framework provides for security structures and allocations for data ranges, but it does not define standard behavior for them. For example, suppose you want your sales managers to oversee high-value invoices. The concept of a high-value invoice varies according to the experience of the sales person. You could define a range called InvLim. You could then allocate data range security to different sales groups specifying different From and To values. Your code could then govern which invoices were accessible by which sales people.

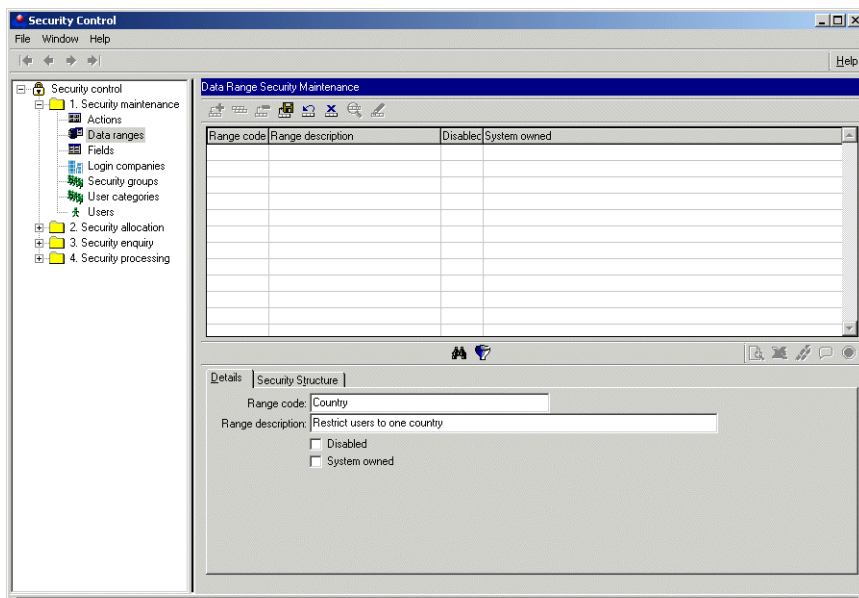
Data range security allocations do not have behavior supplied by the framework.

13.9.1 Creating data range security structures

Data ranges are abstract concepts, so you need to define a structure within the security system to create security allocations.

To create a structure:

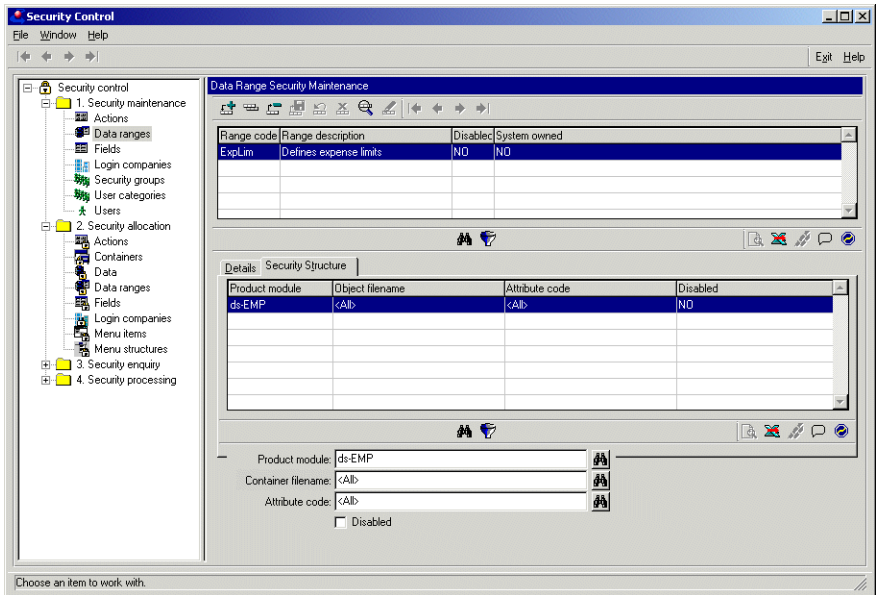
- 1 ♦ Expand the **Security maintenance** node and select **Data ranges**:



- 2 ♦ Type a name in the **Range code** field and document the purpose in the **Description** field. Use **Country** to follow the example.
- 3 ♦ Click **Save**.

The data range security structure you created applies to all modules, objects, and attributes. You can create additional structures if you want to specify a narrower scope. To add a structure:

- 1 ♦ Click the target data range.
- 2 ♦ Select the Security Structure tab.

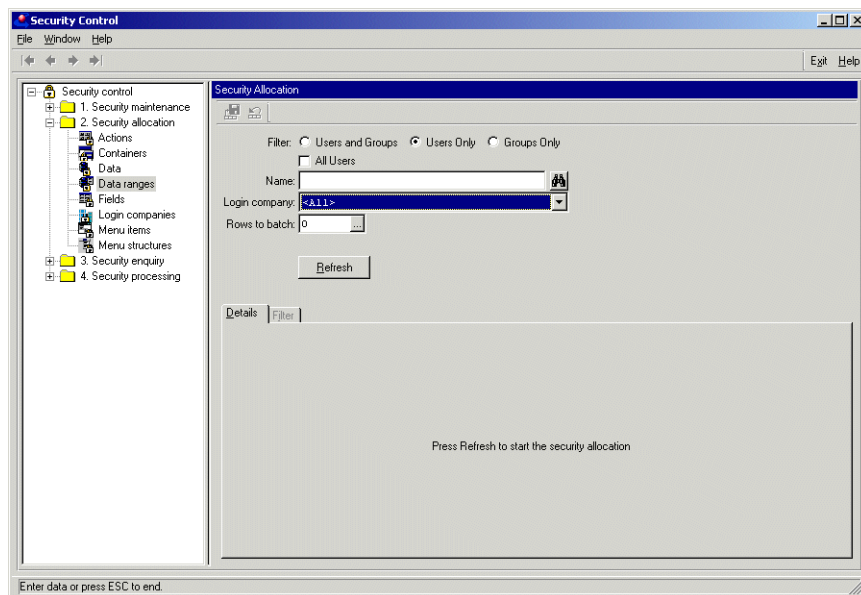


- 3 ♦ Click **New**.
- 4 ♦ Specify the module, object, and attribute combination in the supplied fields.
- 5 ♦ Click **Save**.

13.9.2 Creating data range security allocations

To define range security allocations:

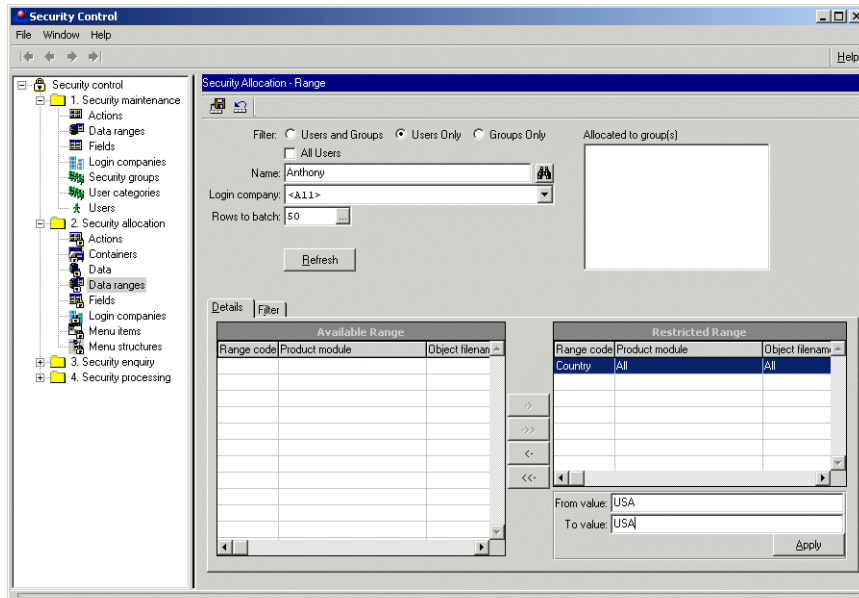
- 1 ♦ Expand the Security allocation node and select **Data range**:



- 2 ♦ At the top of the panel, select **Users and Groups**, **Users Only**, or **Groups Only**.
- 3 ♦ Either specify a user or group in the **Name** fill-in or select the **All...** check box just above the **Name** fill-in. In this example, select user Anthony.

NOTE: If you want this allocation to apply only within a specific login company, choose the appropriate company from the **Login company** list.

- 4 ♦ Click **Refresh** to see a list of available data ranges. If you specified a user or group name, a list of groups directly linked to the specified user or group also appears:



- 5 ♦ Select the data ranges that you want to apply security to and click→. The records move over to the second column. From this list you can apply security to one or more of the records.
- 6 ♦ Select the appropriate ranges in the second list (usually one) and specify the appropriate **From value** and **To value** for the specified user. If there is a single valid value to match, then you should enter the same value as both the From value and the To value. For example, enter **USA** in both fields.
- 7 ♦ Click **Apply**.
- 8 ♦ Click the Save button at the top of the panel to save the allocations.

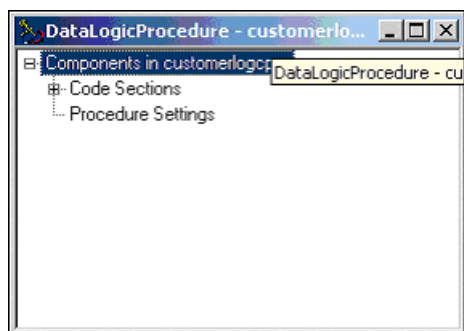
The Data Ranges security mechanism does nothing more than store and retrieve associations between restriction security structures and users or companies. It does not actually qualify any database queries automatically. It is up to you to retrieve the security information and apply it as appropriate.

13.9.3 Programming with data range security

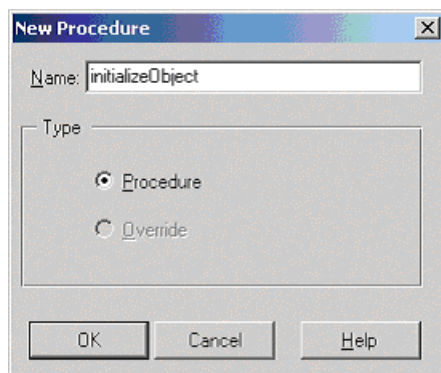
To show how this works, this section shows you how to use your data range allocation in a specific application object, the Customer SDO generated by the Object Generator. You will retrieve the restriction and apply it to the Customer query before it is opened by the SDO.

Remember that Progress Dynamics generates a separate custom super procedure, called the logic procedure, for each SDO. This procedure lets you customize the behavior of any SDO without modifying the base SDO procedure itself, which remains completely generic. You will edit the logic procedure `customerlogcp.p`, not the SDO `customerfull.o`. Follow these steps:

- 1 ♦ In the AppBuilder, choose the **Open** button to open the logic procedure. The logic procedure displays, with a nonvisual TreeView design window:



- 2 ♦ From the AppBuilder toolbar, choose **Edit** to bring up the Section Editor.
- 3 ♦ Drop down the Section list and select **Procedures**, then choose **New**. The New Procedure dialog box appears:



To change the Customer SDO query before it is ever opened, you will customize the standard ADM initializeObject procedure to make the change before any object initialization takes place.

NOTE: Because you are editing a separate procedure file from the SDO procedure itself, the AppBuilder's Section Editor is not able to refer to the list of all overridable (localizable) procedure names, so you cannot use the Override option to locate the one you want.

- 4 ♦ Enter the procedure name, then choose **OK**.
- 5 ♦ Enter the code shown in [Figure 13-2](#) for the customized version of initializeObject.

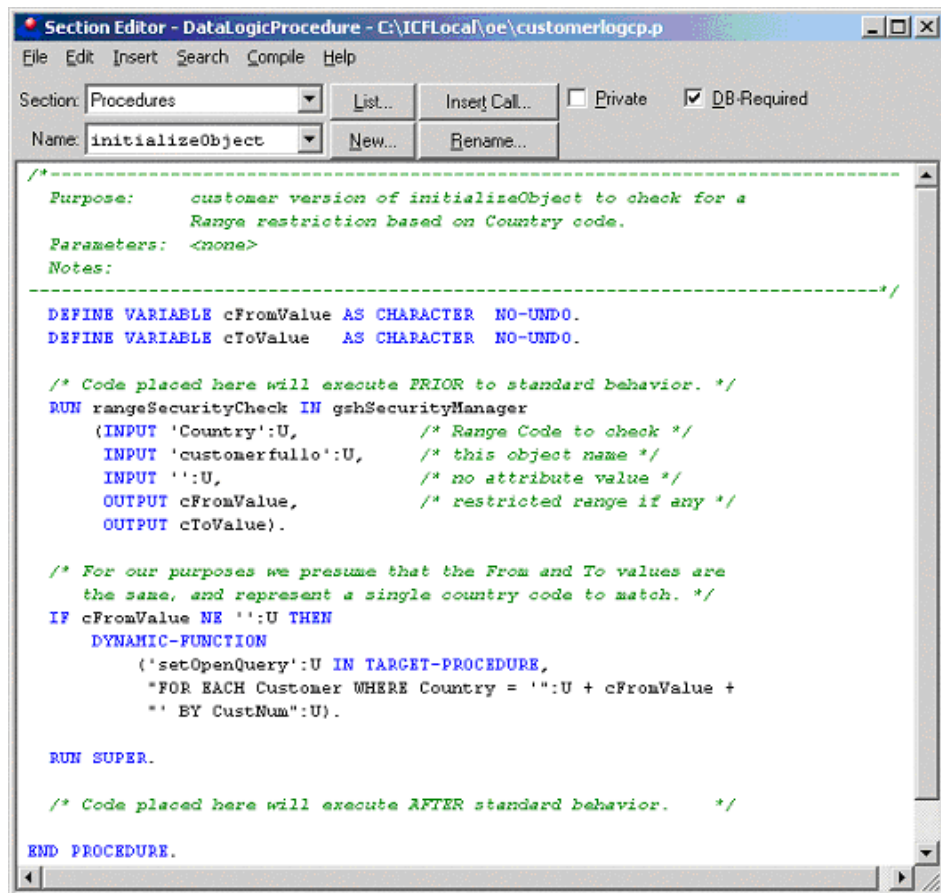


Figure 13-2: Example Customer Selection window

Here you are making direct use of the API for the Security Manager. In all of the security functions you have seen so far, the framework code accesses the Security Manager directly for you, so that the supporting behavior for establishing and applying security restrictions is done automatically, without you having to write any 4GL code at all. In this case however, the framework does not know exactly what use should be made of the range code, so you have to do this yourself. The API for the Security Manager, as well as the API for other Progress Dynamics Managers, is detailed in the [Progress Dynamics Managers API Reference](#). The example code makes use of the fact that the handles to the support procedures for the principal Progress Dynamics managers are available as global variables on the client.

The handle to the Security Manager is `gshSecurityManager`. The internal procedure to run within that handle to get the range security information is `rangeSecurityCheck`. It takes three input parameters:

- The name of the Range Code ('Country' in this example).
- The name of the object making the request (`customerfull0` in this case).
- The name of an attribute value to check (none in this case).

It returns two output parameters, both in character form:

- The From value for the range check.
- The To value for the range check.

In this case, the code assumes that the range is limiting access to a particular country, so only the From value is checked. If the value that comes back is blank, then there is no restriction for the current user and the query is not changed. Otherwise the Country code that comes back is used to qualify the SDO's query. The code sets the SDO's `OpenQuery` property to modify the base query for the SDO.

Once the query has been reset, the procedure executes a `RUN SUPER` statement to invoke the standard initialization for the SDO, which will include opening the query with its new `WHERE` clause.

Save and compile the logic procedure. To test it, launch the Customer selection window, `custbrowsewin`, from the Dynamic Launcher. Remember to check the **Destroy ADM Super Procedures** box so that the new version of the super procedure `customerlogcp.r` will be run. You should see the Customer records for the USA only.

This is just one simple example of how you can use the Data Range security structures. Because there is no specific built-in behavior for these records, you can do anything you want with them. For instance, using the Country code example, it might not be appropriate to have a hard-coded country associated with each different user. You can instead put some other kind of code into the From and To values. You can even use just a logical flag to indicate “yes, this user is restricted to seeing only data from his or her own country,” and then looking up the country in a user table in your own database.

If you look at generalizing the `initializeObject` code above, you could imagine both the Range Code and Object Name input parameters (and the attribute parameter as well, if you use it) as being variable values passed into the code as input parameters of their own. You could, for example, have a whole set of Data Range or other security structure codes that represented different database fields that require filtering, so that a single procedure run from `initializeObject` in any SDO in your entire application could apply filtering in a standard way to a number of different fields. The call to this single filtering procedure could then be a hook in a version of `initializeObject` in a custom super procedure that you define, which could then be made available to every SDO automatically.

Data range conflicts

When a user belongs to more than one security group, and different data range security has been set up against each security group, the framework will not be able to determine which data range security is least restrictive in the context of the application. The framework returns a CHR(3) delimited list of from values and to values to the calling program. It would then be up to the program to decide which values are least restrictive, and apply them.

13.10 Data security

Data security provides a mechanism for securing database records within the Dynamics application. You can select individual records in a table and apply security to those specific records. Data security is another part of the security framework that provides a mechanism to connect your custom programming with the framework's security system. Exactly what the effect of this kind of security on your application will be, is entirely dependent on your custom programming. One way of implementing security of this type might have the following meaning:

- **Revoke model** — Prevents a user from accessing these records. By default, Dynamics allows access to all records available to the application. This level of security is in addition to any other data security. For example, database level security is not affected by Dynamics data security.

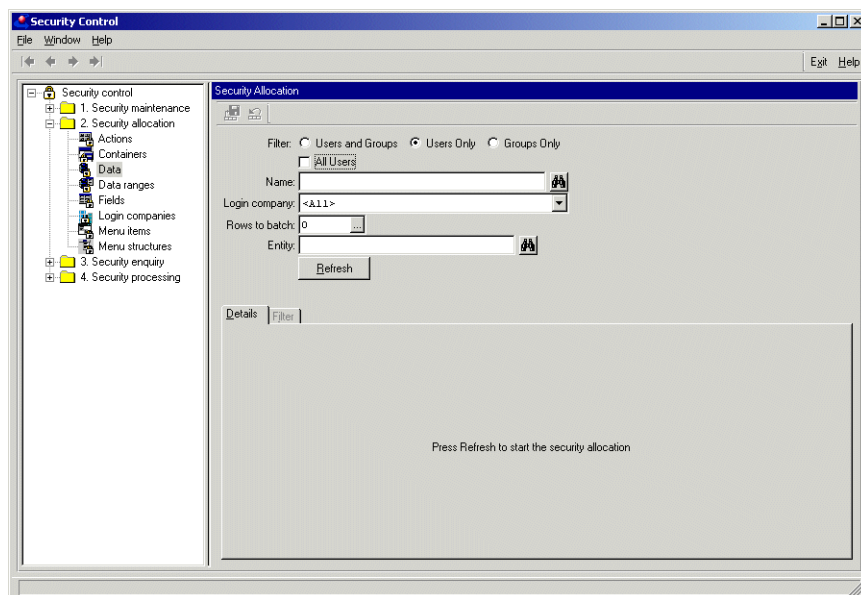
- **Grant model** — Allows a user to access these records. By default, Dynamics prevents all users from accessing database records.

The security system is aware of all records, so there is no need to define an abstract structure to represent them in the security system.

13.10.1 Creating data security allocations

To allocate security at the level of one or more records, follow these steps:

- 1 ♦ Expand the **Security allocation** node and select **Data**:

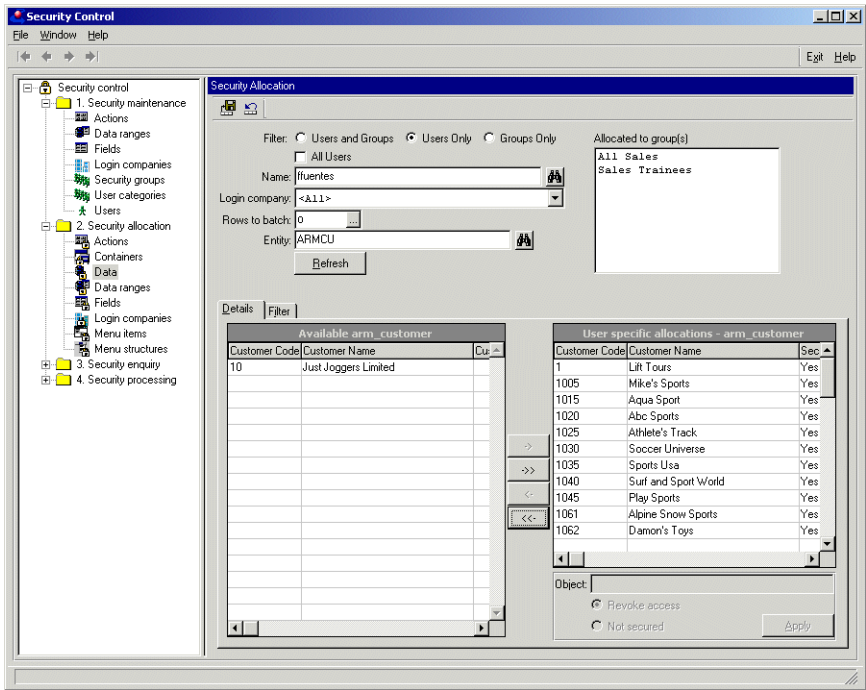


- 2 ♦ At the top of the panel, select **Users and Groups**, **Users Only**, or **Groups Only**.
- 3 ♦ Either specify a user or group in the **Name** fill-in or select the **All...** check box just above the **Name** fill-in.

NOTE: If you want this allocation to apply only within a specific login company, choose the appropriate company from the **Login company** list.

- 4 ♦ Specify the name of the **Entity** (the dumpname of the table).
- 5 ♦ Click **Refresh** to see a list of records from the entity. If you specified a user or group name, a list of groups directly linked to the specified user or group also appears.

6 ♦ In the example below, you see records from the Customer table of the DynSports database:



NOTE: If you have an extensive list, you can improve performance by setting **Rows to batch** to a smaller number or by using the Filter tab to limit the results. If you set this field to zero, the framework will attempt to fetch all values. You may not want to use the zero value if querying a large database.

7 ♦ Select the records that you want to apply security to and click→. The records move over to the second column. From this list you can apply security to one or more of the records.

- 8 ♦ Select the appropriate records in the second list (usually all) and select the action you want to perform. See [Table 13–4](#) for an explanation of your options. The framework sets flags, but their actual meaning depends entirely on your programming. [Table 13–4](#) provides an example of what your code might do in these cases.

Table 13–4: Data security by security model

Model	Action	Intended effect
Revoke	Revoke access	Prevents the specified users from accessing the selected records.
Revoke	Not secured	Removes any existing data security for the specified users on the selected records.
Grant	Grant access	Allows the specified users to access the selected records.
Grant	No access	Prevents the specified users from accessing the selected records throughout the application.

- 9 ♦ Click **Apply**.
- 10 ♦ Click the Save button at the top of the panel to save the allocations.

13.11 Login company security

Login companies provide a mechanism for defining organizational entities that share an application. This mechanism provides the ability to customize application security for each organization.

13.11.1 Creating and maintaining login company security structures

Since administrators normally maintain login companies, you can find documentation on this topic in [Progress Dynamics Administration Guide](#).

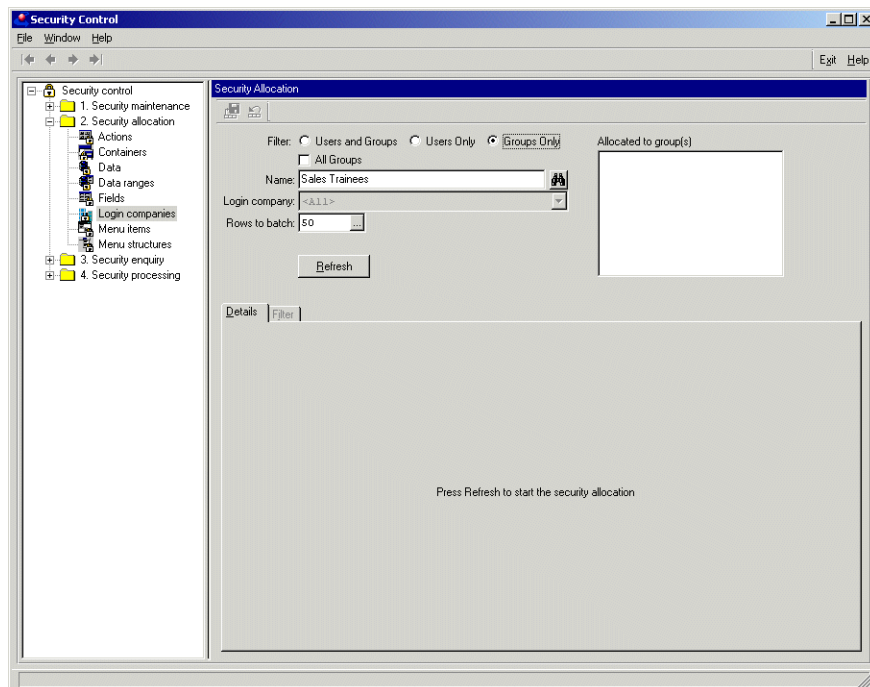
13.11.2 Defining security allocations for login companies

You can define security allocations linking users and groups to specific login companies. Security allocations of this type have the following effects:

- **Revoke model** — Prevents a user or group of users from logging into the application under a specific company. By default, Dynamics allows all users to login in under any of the companies defined in the Repository.
- **Grant model** — Allows a user to log into the application under a specific company. By default, Dynamics prevents all users from logging into the application under any of the companies defined in the Repository.

To define a login company allocation, follow these steps:

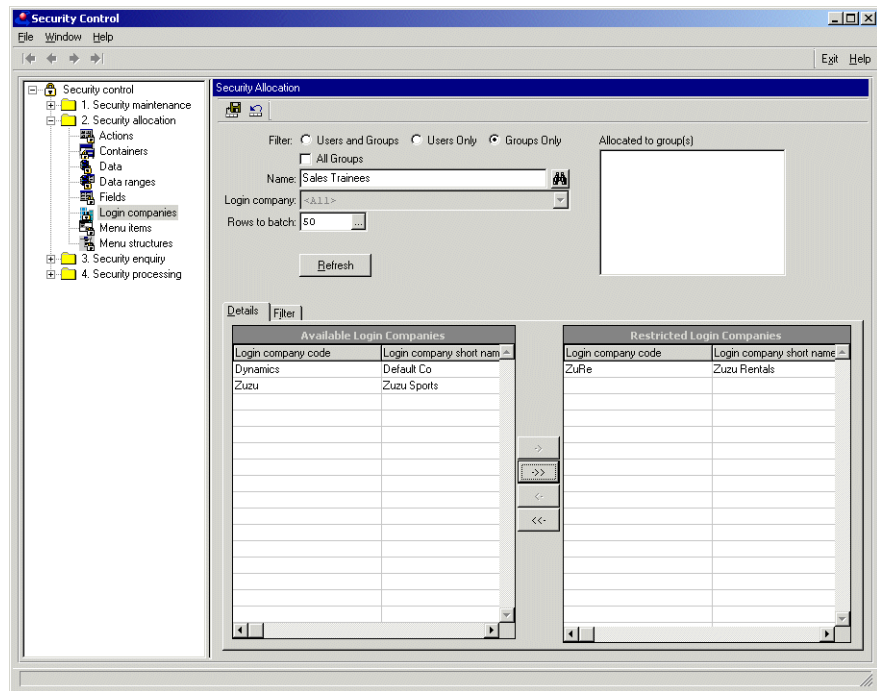
- 1 ♦ Under the Security allocation node select **Login companies**:



- 2 ♦ At the top of the tab, select **Users and Groups**, **Users Only**, or **Groups Only**.
- 3 ♦ Either specify a user or group in the **Name** fill-in or select the **All...** check box just above the **Name** fill-in.

NOTE: The Login Company fill-in is disabled, because it makes sense only to apply a login company allocation to a user or group.

- 4 ♦ Click **Refresh** to see a list of login companies. If you specified a group name, a list of groups directly linked to the specified group also appears:



- 5 ♦ You are now ready to create a link (allocation) between the specified entity and a login company or companies. Select the company or companies that you want to link to the specified user or group and click →.

- 6 ♦ Click the Save button to save the allocation. Depending on which security model you are using, you have:

Revoke model — Now prevented the selected user or group from logging in under companies that were formally available to the user or group.

Grant model — Now allowed the selected user or group to log in to companies that were formally not available to the user or group.

Figure 13–3 shows a member of the Sales Trainees group attempting to login using a restricted company.

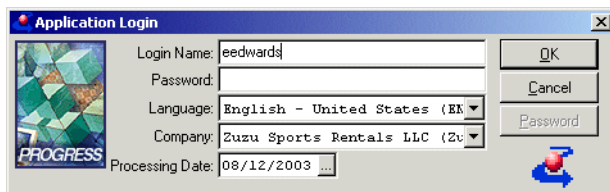


Figure 13–3: Login with restricted company

Figure 13–4 shows the error message that displays when Dynamics denies the login.

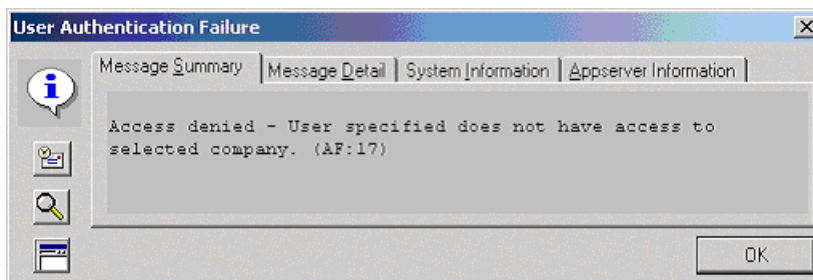


Figure 13–4: User Authentication Failure message

13.12 Searching with the Security Enquiry tool

The security definition for a complex application can be extensive. The Security Enquiry tool allows you to find existing security settings based on a wide array of characteristics. When you select the Security Enquiry node, several nodes appear. Each represents a type of security allocation and allows you to list details of that type as they apply to groups or users that you specify.

The table below describes the nodes:

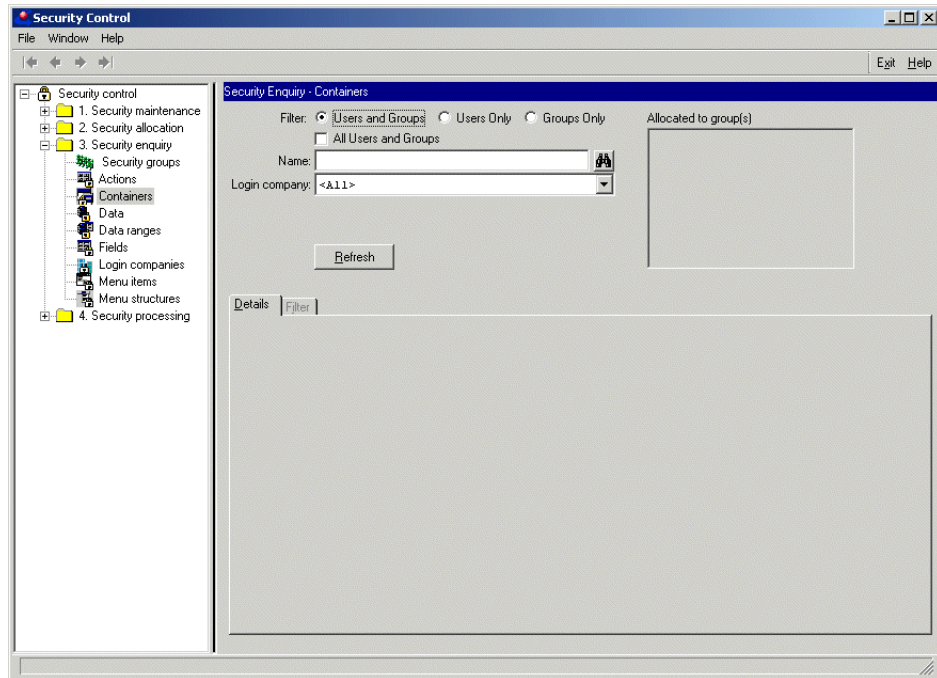
Option	Purpose
Security groups	Display the relationship of groups to selected companies, groups, or individuals.
Actions	Display details of action security allocations as they relate to the specified groups or individuals, including object, filename, the and whether an allocation applies at the object or instance level.
Containers	Display the security status of container application objects as they relate to the specified groups or individuals.
Data	Display the security status of data entities (records) as they relate to specific groups or individuals.
Data ranges	Display the details of defined data ranges as they relate to specific groups or individuals, including minimum and maximum values.
Fields	Display the security status of application field objects as they relate to specific groups or individuals.
Login companies	Display the security status of login companies as they relate to specific groups or individuals.
Menu items	Display the security status of application menu items as they relate to specific groups or individuals.
Menu structures	Display the security status of application menu structures as they relate to specific groups or individuals.

It is an excellent idea to always use the Security enquiry tool to see all relevant information before you make changes to your security definitions.

Each of the nodes uses the same technique to filter security data for you, although the details provided are different for each node. Details include whether the object is secured, how it is secured, and why the object is secured or not secured.

To use the tool, follow this general procedure:

- 1 ♦ From the Security Control window, expand the Security Enquiry node on the tree.
- 2 ♦ Select the type of security allocation you want to review. A panel appears. The example below is for the Containers node:



- 3 ♦ Specify the context of your search at the top of the panel. For example, specify a company, group, or user to query on. In some cases you need to specify an object name as well.
- 4 ♦ Click Refresh. The Security Control populates the details tab. As you change the context parameters of your search, you need to click the Refresh button each time to repopulate the Details tab.

- 5 ♦ At the top of the Details tab, there may be options to filter the number of rows. Simply click these as you see fit, and the list refreshes automatically. Details.e user selects the applicable node depending on what type of security he wants to query. He enters the user and organization he wants to enquire on, and presses the 'Refresh' button.
- 6 ♦ You can further filter the records, using the standard Dynamics Filter tab. To apply settings from the Filter tab, click the Apply button on the Filter tab. Pressing Refresh here has no effect, except to reset the Filter tab.

NOTES:

- The column labels are different, depending on whether you use the grant or revoke model. Note that you can export data to a Microsoft Excel spreadsheet by clicking the Excel icon at the bottom of the panel.
- The Security Enquiry tool derives much of its utility from the comments the system provides. The comment in the editor is built by the framework. The comment is meant to give you an idea of how the system determined which security to apply. For instance, if conflicting security had been picked up against different security groups, the comment would mention against which security groups security had been found and how it had been applied to give you the eventual security panel.

Index

Numbers

4GL code
 from ERwin macros 2–18
4GL procedures 5–6

A

Access
 defining 13–1
 distributed 2–10
 optimizing 10–6

Access control 13–4

Access Tokens 13–20

Accessing records 10–3

ActiveX control 9–2

Add to Repository dialog box 3–9, 11–7

Add to Repository menu option 3–5

Adding
 Apply button 9–25
 bands 12–19
 bands to toolbars 12–30

categories 12–7
dynamic combos to static viewers 7–3
files to Repository 3–26
lookup to viewer 7–13
modules to product 4–11
toolbars to windows 12–34

ADM. See Application Development Model

Administration access 13–4

Administration Security menu 9–28

Administration window 3–12
 example 13–15

aferrortxt.i file 5–19, 10–44

afglobals.i file 10–50

All option, defined 7–9

Ampersands shortcut character 13–17

APIs

 Security Manager 13–35
 TreeView 9–2

AppBuilder 3–1, 3–3, 5–3
 enhancements 3–3
 starting 3–3

- Application Development Model (ADM)
 - 1–2, 5–3
 - naming conventions 2–6

- Application Login window 3–3
 - example 13–18

- Application objects, creating 3–21

- Application security, defining 13–1

- Applications
 - distributed 10–3
 - example 1–8
 - host-based 10–3

- Apply button, adding 9–25

- AppServer partition name 6–4

- AppServers
 - calls from client 10–6
 - connection 10–52
 - goals 10–3
 - log file 7–12
 - optimizing access 10–6
 - stateless access 1–4

- AppService property 11–29

- Arguments, launch.i 11–14

- Array fields 2–8

- ASDivision property 11–29

- ASHandle property 10–13, 11–29

- Assigning
 - key values 11–39
 - login companies 4–11
 - product names 4–8
 - site numbers 4–7

- assignQuerySelection function 11–35

- Associating items with bands 12–27

- Attribute Editor, ERwin 2–16

- Attribute values, records 5–3

- Attributes instance 10–8

- Attributes menu 3–14

- Attributes. See Also Properties.

- Audience –xvii

- Audit history 8–31

- Auditing tables 4–23

- Auto Properform Strings 4–14

- AutoCommit property 10–16, 11–29, 11–37

B

- b_ prefix 5–19

- Bands
 - adding to toolbars 12–30
 - defined 12–19
 - Find and Filter 8–29
 - Navigation 8–29
 - Window vs WIndows 12–25

- Base Query String 7–6

- Batch Size, determining 10–18

- Batches, multiple 10–19

- Batching, rows 7–16, 10–18

- Before-image record 10–15

- beginTransactionValidate procedure 10–17,
10–31, 11–50

- Brokering, SDO data in SBO 11–32

- Browse sequence number 7–16

- Browser property sheet 6–6

- Browser Settings dialog box 5–23

- Browsers 5–2, 5–8
 - creating 6–5
 - customizing 6–6
 - design window 6–5
 - property sheet 3–29
- BrowseToolbar 8–29
- BrowseToolbarNoUpdate Toolbar 8–31
- Build Sequence option 7–10
- Building
 - objects 5–2, 5–10
 - queries 10–23
 - SBOs 11–26
 - TreeViews 9–4
- Built-in objects, viewing 12–3
- Business logic 5–5, 10–1
 - defining for SBOs 11–50
- Buttons
 - enabling 8–27
 - Filter 4–20
 - Save 4–24
 - Search 12–3
 - SpellCheck 8–29
 - Update 8–29
 - update 4–20, 8–29
- C**
- Caching data 1–4
- CAN-FIND validation 10–5
- canNavigate function 12–10
- Carat delimiter 10–45
- Categories
 - adding 12–7
 - creating 12–7
 - defining 12–6
 - SmartLinks 12–6
- Changes, losing 11–9
- CheckCurrentChanged property 10–6, 10–17
- checkerr.i file 10–47
- Checking error messages 10–47
 - _cl suffix 5–7
- Client access 10–4
- Client Proxy SDO 5–5
- Client proxy wrapper procedure 10–38
- Client SBO proxy 11–34
- Client support 1–4
- Client-server technology 10–3
- Client-side data validation 10–26
- Closing
 - files 3–31
 - objects 3–31
- Code structure 1–7
 - _code suffix 4–17
- Coding messages 10–46
- Column Editor 10–24
- Column validation 10–26
- ColumnLabel attribute 5–22
- colValues function 11–34
- Combining static and dynamic objects 7–4
- Combos
 - adding to viewers 7–3
 - defining 7–4
 - saving 7–13
- Comments Control 8–31
- Comments field 4–12
- Commit 10–15

- CommitSource property 11–30
- CommitSourceEvents property 11–30
- Companies, login 4–11
- Comparison operators 9–23
- Compile Menu 3–11
- Components standard 1–6
- Connecting visual objects at run time 11–12
- Constant data 2–5
- Constraints, avoiding 2–10
- constructObject procedure 10–12, 11–34
- ContainedDataColumns property 11–30, 11–32
- ContainedDataObjects property 11–30, 11–32
- Containers 5–3
 - running 3–29
- Controls, TreeView 9–2
- Conventions
 - naming dynamic objects 2–6
 - naming entities 2–3
- createPreTransValidate procedures 5–20
- Creating
 - Browsers 6–5
 - categories 12–7
 - dynamic viewers 5–25
 - individual objects 3–21, 6–2
 - items 12–8
 - menu item labels 12–17
 - multiple objects 3–23
 - objects 3–21
 - PLIPs 11–17
 - product modules 4–11
 - records 11–7
 - Repository data 4–12
 - SBOs 11–44

- SDOs 5–15
- SmartDataObjects 11–3
- static viewers 7–3
- TreeViews 9–4
- viewers 6–7
- .cst files 3–6
- CTRL key 4–13
- Cursor
 - fill-in field 7–4
 - hour glass 9–24
- Custom files 3–6
- Custom Super Procedures 6–6, 10–20
 - defining 12–36
- Customizations, ERwin 2–18
- Customizing
 - Browsers 6–6
 - data in tables 4–18
 - templates 5–13
 - triggers 11–13

D

- Data
 - accessing 11–2
 - caching 1–4
 - constant 2–5
 - deploying 4–23
 - extracting 9–17
 - loading 9–18
 - management 10–13
 - multi-level 9–2
 - raw 2–5
 - sharing 2–13
 - storing 1–3
 - transferring 10–3, 11–36
 - versioning 4–23
- Data access 13–4
- Data Dictionary 11–2
- Data Fields 5–3

- Data link hierarchy 11–32
- Data links 11–31
- Data management
 - example 11–35
 - SBOs 11–34
- Data ranges security 13–28
- Data records security 13–36
- Data sets
 - deployment 3–13
 - restricted 10–19
- Data source, original 11–2
- Data types, character 4–17
- Data validation, client-side 10–26
- data.p procedure 12–10
- dataAvailable event 11–34
- Databases
 - client access 10–4
 - designing 2–1
 - distributed access 2–10
 - dump name 4–21
 - name qualifier 11–11
 - NO-LOCK records 10–14
 - normalization 2–7
 - rows to batch 7–16
 - site numbers 4–2, 4–7
 - Sports2000 7–14
 - triggers 2–17, 11–12
- DataColumns property 11–29
- DataLogicProcedure 13–33
- dataObjectHandle function 11–40
- DataObjectNames property 11–30
- DataObjectOrdering property 11–30
- DataSevers 4–14
- DataTargetEvents property 11–30
- DataType attribute 5–22
- Dates, processing 3–3
- Dbname 4–21
- DB-REQUIRED flag 10–27
- DB-REQUIRED technique 10–38
- DEFINE TEMP-TABLE statement 10–25
- Defining
 - application structure 4–1
 - categories 12–6
 - custom super procedures 12–36
 - dynamic combos 7–4
 - field security allocations 13–24
 - keys 2–13
 - lookups 7–13
 - queries 10–21, 11–5
 - record security 13–36
 - security 13–1
 - security for data ranges 13–28
 - toolbar bands 12–19
 - TreeView nodes 9–4
 - TreeView windows 9–11
- Delimiters 2–3, 4–14
 - carat 10–45
- Deploying data 4–23
- Deployment data sets 3–13
- Deployment site number 4–2
- DeptCode trigger 9–23
- Derived dataset 11–2
- _desc suffix 4–17
- _description suffix 4–17
- Design conventions 2–1
- Design windows 3–19
 - Browser 6–5

- Designing
 - databases 2–1
 - tables 2–7
- destroyObject procedure 11–9
- Development window 3–14
- Dialog boxes
 - Add to Repository 3–9, 11–7
 - Browser Settings 5–23
 - Entity Mnemonic Import 4–13
 - Multi-Field Mapping 11–46
 - Multi-Field Selector 6–11
 - New 3–6, 11–18
 - Open Object 3–7, 6–3
 - Preferences 3–17
 - Record Menu Accelerator 12–14
 - Search Nodes 12–3, 13–13
 - Set Site Number 4–7
 - Suspended User 3–16
 - Table Selector 11–10
 - Viewer Settings 5–25
- Dictionary validation 11–6
- Disabled options 3–19
- Display fields, maintaining 4–24
- Display Fields option 4–18
- Displaying
 - fields 4–15
 - status 5–26
- Distributed applications, writing 10–3
- Distributed database access 2–10
- Distributed environment 1–4
- Dividing line, TreeViews 9–27
- Dump name 4–21
 - standards 2–4
- Dynamic browser property sheet 3–29

- Dynamic combos 7–3
 - defining 7–4
 - saving 7–13
- Dynamic launcher 3–30
- Dynamic lookups 7–3
 - adding to viewer 7–13
 - property sheet 7–16
- Dynamic objects 3–22
 - editing 3–19
 - naming 2–6
 - running 3–29
- Dynamic viewers, creating 5–25
- DynBrow 3–19
- DynFold 3–19
- DynMenc 3–19
- DynObjc 3–19
- DynToolbar 8–31

E

- EditAction procedure 12–38
- Editing
 - individual objects 6–2
 - properties 3–28
 - Repository objects 3–19
 - SDOs 5–15
 - viewers 6–7
- ENABLE statement 9–23
- Enabling
 - buttons 8–27
 - filters 9–22
- endTransactionValidate procedure 10–17,
10–31, 11–50
- Enhancements 3–3

- Entities
 - creating data for 4–12
 - keys 4–17
 - maintaining 4–18
 - naming conventions 2–3
 - record fields 4–16
 - Entity Import process 4–13
 - Entity maintenance window 4–21
 - Entity Mnemonic field 4–21
 - Entity Mnemonic Import dialog box 4–13
 - Environment, preparing 4–1
 - Error messages
 - ADM2 11–52
 - checking 10–47
 - formatting 10–34, 10–44
 - Error Summary Description 10–44
 - ERwin
 - described 2–16
 - modeling tool 11–12
 - naming length limits 2–6
 - template benefits 2–6
 - essential practices 1–5
 - Events, filterDataAvailable 9–22
 - Example application 1–8
 - Examples
 - Administration window 13–15
 - application 1–8
 - Application Logon window 13–18
 - EditAction procedure 12–38
 - foreign key fields 7–3
 - managing data 11–35
 - SmartDataViewer 7–5, 7–14
 - static viewer 6–11
 - Treeview 9–5
 - EXCLUSIVE-LOCK 10–17
 - Expanding
 - message dialog box 10–51
 - nodes 9–13
 - Extensions 3–20
 - Extract program guidelines 9–17
 - Extracting data 9–17
- ## F
- fetchContainedData procedure 11–36
 - Field Container objects 5–3
 - Field name values 2–14
 - Field value 9–23
 - FieldName attribute 5–22
 - FieldOrder attribute 5–22
 - Fields
 - creating data for 4–12
 - defining security 13–24
 - foreign keys 7–3
 - grouping 2–9
 - linking 7–17
 - loading 4–15
 - maintaining for display 4–24
 - naming conventions 2–3
 - non-updatable 5–15
 - numbers of 2–9
 - Object ID 2–13
 - object_path 3–20
 - parent 7–11
 - removing from SDOs 5–15
 - replication 4–22
 - separators 4–14
 - validation 11–6
 - File menu enhancements 3–5
 - Filenames 3–20

Files

- adding to Repository 3–26
- aferro.txt.i 5–19
- closing 3–31
- custom 3–6
- naming conventions 2–3
- opening 3–24

Fill-in field cursor 7–4

Filter button 4–20

Filter window 4–18

filterDataAvailable event 9–22, 9–23

Filtering

- lookups 7–23
- objects 4–9
- tables 4–18
- techniques for 7–25

Filters 3–18

- enabling 9–22

Find and Filter band 8–29

First menu item 12–3

Five letter acronym (FLA) 2–5

FLA. See Five letter acronym.

FolderPageTop Toolbar 8–27

FolderTop Toolbar 8–29

FolderTopNoSDO Toolbar 8–31

FOR EACH blocks 10–29

Foreign fields 7–3

- specifying 11–45

Foreign key value 7–16

Foreign keys 2–13, 2–14

ForeignFields property 11–29

ForeignValues property 11–29

Format attribute 5–22

Formatting

- error messages 10–44
- errors 10–34

Frames

- multiple 9–27
- viewer 7–1

Framework structure 10–1

Functions

- canNavigate 12–10
- dataObjectHandle 11–40
- getASHandle 11–23
- getDataHandle 11–31
- isCreate() 5–21
- isFieldBlank 5–19
- SDOs 11–41

G

General Manager 4–22

Generating

- Browsers 6–5
- Data Fields 5–3
- Dynamic Viewers 5–25
- objects 5–1

getASHandle function 11–23

getDataHandle function 11–31

getEntityDescription 4–22

getQueryPosition function 11–31

Getting started

- example application 1–8
- site numbers 4–2

Global references 10–50

Goals of Progress Dynamics 1–8

Granting access 13–4

gsc_entity_mnemonic table 4–12, 9–21

gsc_object table 3–20
gshSecurityManager handle 13–35
gsm_node table 9–14
Guidelines, indexing 2–11

H

Handles
 gshSecurityManager 13–35
 phTable 9–20
Handling messages 10–42
Hooks
 alternate 10–32
 validation 11–50
Host-based applications 10–3
Hour glass cursor 9–24
Hyphens 2–3, 4–8
 in entity names 4–14

I

ICFAF-Secu 9–28, 13–14
Importing
 entities 4–13
 tables and fields 4–13
Improving
 performance 5–15
 user interface 9–24
Include files
 aferro.txt.i 5–19
 checkerr.i 10–47
 launch.i 11–14
Indexes
 guidelines 2–11
 table 4–18
Inherit Dictionary Validation option 11–6

initializeObject procedure 9–22, 10–19
Initializing viewers 7–13
InitialValue attribute 5–22
Inner Lines option 7–9
Instance properties
 Data Field 5–22
 NavigationTargetObject 11–40
 setting 10–8
Instance Property dialog box Toolbar 11–33
Internal procedures
 rangeSecurityCheck 13–35
 SDOs 11–41
isCreate() function 5–21
isFieldBlank function 5–19
Items
 associating with bands 12–27
 creating 12–8
 defined 12–8
 defining security 13–16
 in Repository 13–16

J

Join fields 2–3
Joins 4–22, 5–15, 5–17
 problems 10–21

K

Key field 4–16
Key values, assigning 11–39
Keys 2–11
 defining 2–13
 entity 4–17
 Object ID 4–2

Keys fields 7–3

killPlip procedure 11–19

L

Label attribute 5–22

Labels, menu items 12–17

Language Code 10–43

launch.i file 11–14

Launching objects 3–30

launchProcedure 11–14

Layout Object Name 8–10

LEAVE trigger 9–23, 10–28

Letter icon 10–52

Limits

- for entity names 2–6
- performance 10–3

Link Advisor 11–45

Linking

- fields 7–17
- widgets 7–17

Links, assigning to layout 8–19

loadData procedure 9–18

Loading

- data 9–18
- fields 4–15
- temp-tables 11–2

Local variables 7–17

Locating data objects 11–40

Locking records 10–5

Log file, generated objects 5–26

Logic

- client-side 10–27
- table maintenance 10–36

Logic procedures 5–6, 10–20

- SDOs 5–5, 10–36

Logical database name 4–21, 11–2

Login company 3–3

- assigning 4–11
- restrictions 13–11, 13–40

Login Company Control 13–7

Login window 3–3

lookupDisplayComplete procedure 9–25

Lookups

- adding to viewer 7–13
- property sheet 7–16

Losing changes 11–9

M

Macros, ERwin 2–18

Magnifying glass icon 10–52

Maintaining

- display fields 4–24
- entities 4–18

Managers, client vs. server 1–4

Managing

- data 10–13
- data in SBOs 11–34
- transaction scopes 11–20

Mandatory attribute 5–22

Manual, organization of –xviii

Master attributes, Data Field 5–22

Master SDO 11–32

Master table 2–5

- MasterDataObject 11–40
- MasterDataObject property 11–30
- Master-detail relationship 11–28
- Menu items, labels 12–17
- Menu structures
 - defined 13–11
 - TreeViews 9–27
- Menus
 - Attributes 3–14
 - Compile 3–11
 - File 3–5
 - Preferences 3–16
 - Re-Logon 3–16
 - SmartObjects 3–14
 - Suspend 3–16
- Merging components 2–13
- Message Control 10–42
- Message dialog box 10–51
- Message Group 10–43
- Message handling 10–42
- Message Maintenance window 10–43
- Message management tools 10–1
- Message Number 10–43
- Messages
 - ADM2 error 11–52
 - coding 10–46
 - translating 10–43
- Modeling tool, ERwin 2–16
- Modifying SDO definitions 10–23
- Modules, adding to product 4–11
- Mouse cursor 9–24
 - fill-in field 7–4
- Multi-Field Mapping dialog box 11–46

- Multi-Field Selector dialog 6–11
- MultiInstanceActivated property 12–25
- Multi-level data 9–2
- Multiple frames 9–27
- Multiple objects, creating 3–23
- Multiple record updates 10–29

N

- _name suffix 4–17
- Named arguments, launch.i 11–14
- Named template object 7–13
- Names, entity mnemonic 4–21
- Naming
 - objects 3–20
 - product modules 4–8
- Naming conventions
 - dynamic objects 2–6
 - entities 2–3
 - files 2–3
 - tables 2–3
- Naming length limits 2–6
- Navigation band 8–29
- Navigation buttons, enabling 8–27
- Navigation events 11–33
- NavigationSourceEvents property 11–30
- Navigation-Target
 - SBOs 11–40
- NavigationTargetObject 11–40
- Nested menus 9–2
- New button 6–5

New dialog box 3–6, 11–18

New repository objects 3–21

node_obj field 9–14

Nodes

 defining 9–4

 structured 9–13

NO-LOCK 10–5

None option defined 7–9

Non-indexed field 4–19

Non-updatable fields 5–15, 10–24

NO-RETURN variables 10–50

Normalization 2–7

_number suffix 4–18

Numbers of fields 2–9

O

o_obj domain 4–17

_obj suffix 2–13, 4–14, 4–17

ObjcTop Toolbar 8–28

Object attributes 10–9

Object Generator tool 4–1, 4–12, 4–22, 5–1,
5–10, 11–3

Object ID field 2–13

Object IDs 4–2

 benefits 2–13

 for tables 2–13

Object properties, editing 3–28

Object Properties toolbar button 6–3

object_path field 3–20

objectDescription procedure 11–19

ObjectMapping property 11–30, 11–33

ObjectName attribute 5–22

ObjectName property 11–31

Objects

 closing 3–31

 connecting at run time 11–12

 creating 3–21

 editing 3–19

 filtering 4–9

 locating 11–40

 naming 3–20

 opening in AppBuilder 3–23

 organizing 4–8

 registering 11–7

 resizing 8–4

 running 3–29

 saving 3–24

 Toolbar 8–27

 types of 3–19

Obtaining site numbers 4–2

One-to-many joins 10–21

One-to-one joins 10–21

Open File menu option 3–5

Open Object dialog box 3–7, 6–3

Open Object menu option 3–5

Opening

 files 3–24

 objects 3–23

 objects in AppBuilder 3–19

 SDOs 6–2

Operators 9–23

Original data source 11–2

Overrides, labels 7–19

P

- Pages 5–2
- Panel class 11–40
- Parent Fields option 7–11
- Parent Filter Query option 7–12
- parent_node_obj field 9–14
- Partition property, SBO 11–47
- Password for site code 4–4
- Pathnames 3–20
- pcParentNodeKey 9–20
- Performance 10–6, 10–26
 - Combos and Lookups 7–27
 - improving 5–15
 - limitations 10–3, 10–4
- Persistent Libraries of Internal Procedures.
 - See PLIPs
- Persistent procedure handle 10–13
- Persistent procedures, deleting 11–14
- phTable handle 9–20
- Physical file 3–24
- PLIPs 9–18
 - creating 11–17
 - defined 11–14
 - vs SDO logic procedures 11–22
- plipSetup procedure 11–19
- plipShutdown procedure 11–19
- Populating
 - SDO temp-tables 11–7
 - TreeViews 9–17
- postTransactionValidate procedure 10–17, 10–32, 11–50
- Preferences dialog box 3–17
- Preferences menu 3–16
- Prefixes
 - b_ 5–19
 - for filenames 3–20
 - for tablenames 2–4
 - length of table names 4–14
- Preprocessors, definitions 5–6
- preTransactionValidate 10–17
- preTransactionValidate procedure 5–8, 10–17, 10–30, 11–50
- Procedures
 - client proxy wrapper 10–38
 - constructObject 10–12, 11–34
 - createPreTransValidate 5–20
 - custom super 12–36
 - destroyObject 11–9
 - EditAction example 12–38
 - in SDO 5–7
 - loadData 9–18
 - lookups 9–25
 - objectDescription 11–19
 - registering 12–37
 - serverSendRows 10–13
 - server-side 10–6
 - transaction scoping 11–20
 - validation 10–26
 - writePreTransValidate 5–21
- Processing date 3–3
- Product, RY 4–8
- Product Control window 4–9
- Product Maintenance window 4–10
- Product Module tab 4–11
- Product modules
 - creating 4–11
 - defining 4–8
 - security restrictions 4–9
- Progress AppBuilder 3–1

- Progress Dynamics
 - components 1–2
 - description 1–2
 - essential practices 1–5
- Properties, editing 3–28
- Properties button 6–3
- Properties. See Also Attributes.
- Property sheets
 - example 3–29
 - Lookups 7–16
 - opening 3–19, 10–23
 - SDOs 11–47
- property sheets, Browsers 6–6
- Proxy SDO 5–3, 10–37
- pushTableAndValidate procedure 11–39

Q

- Queries 10–13
 - defining 11–5
 - editing 6–4
 - guidelines for 7–6
 - parent filter 7–12
- Query Builder 10–23, 11–10
- Query String 7–6
- QueryPosition property 12–10
- Question arguments 10–49

R

- Range restrictions, defining 13–31
- Range Security Control 13–8
- Ranges security 13–28
- rangeSecurityCheck procedure 13–35

- Raw data 2–5
- Rebuild On Reposition toggle 10–20
- RebuildOnRepos property 10–20
- Recaching menu information 13–18
- RECID 2–12
- Record Menu Accelerator dialog box 12–14
- Record Version table 4–24
- record_ref field 9–21
- Records
 - accessing 10–3
 - creating 11–7
 - locking 10–5
 - NO-LOCK 10–14
 - security 13–36
 - selecting multiple 4–13
 - updating 10–6, 10–29
- Records to retrieve 7–23
- _reference suffix 4–17
- References, global 10–50
- Referential Integrity (RI) constraints 11–12
- Refresh button 12–3
- Regenerating entity information 4–15
- Registering
 - objects 11–7
 - procedures 12–37
 - site numbers 4–4
- Relational database design 2–1
- Relative Path 4–12
- Relative pathname storage 4–9
- Re-Logon menu 3–16
- Re-Logon menu option 3–5
- removeQuerySelection function 11–35

Removing fields from SDOs 5–15

Replicating fields 4–22

Repository database

- adding file 3–26
- benefits of 1–3
- names for objects 2–6
- site number 4–2

Repository objects

- closing 3–31
- creating 3–21
- editing 3–19

Resizing

- objects 8–4
- TreeViews 9–27

Restricting access 13–4

Restrictions

- Login Company 13–11
- naming 2–6
- user name 13–11

Retrieving

- error messages 10–44
- records. See Rows To Batch.

Roundtable 5–13, 6–10

ROWID 2–12

RowMod field 10–15

RowObject 9–23

- temp-table 11–2

rowobject table 5–3

RowObjectState property 11–30

rowObjectValidate procedure 5–8, 10–28, 10–33

RowObjUpd table 5–5, 10–15

Rows

- batching 10–18
- in layout design 8–3

Rows To Batch 7–16, 7–23

- initial setting 10–20

RUN SUPER statement 9–22, 10–20

Run time architecture 1–5

Running

- lookup window 7–22
- objects 3–29
- TreeViews 9–26

ry prefix 2–5

RY Product 4–8

ryc_smartobject table 4–23

rysttvieww.w template 6–9

S

Save button 4–24

Saving

- combos 7–13
- objects 3–24

SBOs. See SmartBusinessObjects.

Schema

- validation 5–16, 10–4

SCM Field Name 4–23

SCM. See Source Code Management.

Scoping 10–3

- transactions 10–5
- within procedures 11–20

SCREEN-VALUE 9–23

SDO procedures 5–7

SDO Startup 10–12

SDOs, programming interface 11–41

SDOs. See SmartDataObjects.

- Search button 12–3
- Search Nodes dialog box 12–3, 13–13
- Security
 - defining 13–1
 - Items 13–16
- Security allocation 13–7
- Security Control window 13–7
- Security Manager API 13–35
- Security menu 9–28
- Security tokens 13–20
- Selecting multiple records 4–13
- Separators 2–3
 - carat 10–45
 - for field names 4–14
- Sequences, Build 7–10
- serverContainedSendRows procedure 11–37
- serverFetchContainedData procedure 11–36
- ServerSDO 5–5
- serverSendRows procedure 10–13
- Server-side procedures 10–6
- Server-side validation 10–29
- Session Manager 11–14
- Session Reset menu option 3–5
- Set Site Number dialog box 4–7
- Sharing data 2–13
- _short_desc suffix 4–18
- _short_name suffix 4–17
- Showing fields 4–15
- Signature property 11–32
- Site count 4–6
- Site numbers 2–13, 4–2
 - allocation 2–15
 - globally unique 2–15
 - limit 4–6
- Sizing objects 8–4
- SmartBusinessObjects
 - Partition property 11–47
 - properties 11–29
- SmartBusinessObjects (SBOs) 2–9, 5–2
 - as Data-Target 11–42
 - building 11–26
 - business logic 11–50
 - creating 11–44
 - defined 10–1
 - files 11–29
 - guidelines 11–27
 - Navigation-Target 11–40
 - when to use 11–28
 - wizard 11–44
- SmartDataBrowsers 5–2, 5–8
 - creating 6–5
- SmartDataFields 7–3, 10–22
- SmartDataObjects
 - functions 11–41
 - internal procedures 11–41
 - procedure handles 11–32
- SmartDataObjects (SDOs) 4–1, 5–2
 - creating 5–15, 11–3
 - defined 5–3, 10–1
 - defining 10–23
 - field validation 11–6
 - guidelines 10–6
 - logic procedure 5–5
 - opening 6–2
 - property sheet 11–47
 - proxy version 10–37
 - query definition 10–21
 - vs PLIPs 11–22

- SmartDataViewers (SDVs) 5–2, 5–9, 10–28, 11–34, 11–37
 - defined 7–1
 - example 7–5, 7–14
 - generating 5–25
 - SmartFilter object 10–18
 - SmartFolder 5–2
 - SmartLinks
 - associating with categories 12–6
 - new 8–21
 - SmartObjects 5–2
 - SmartObjects menu 3–14
 - SmartSelect 7–3, 7–27
 - SmartToolbar 5–2, 12–1
 - defined 12–29
 - SmartWindows 5–2
 - Sorting fields 7–16
 - Source Code Management (SCM) 5–13
 - Source procedures 5–6
 - SpellCheck button 8–29
 - Sports2000 database 7–14, 9–17
 - SQL databases 4–14
 - Stack trace 10–52
 - Standard code structure 1–7
 - Standard components 1–6
 - Standard toolbar objects 8–27
 - Standard validation procedures 10–26
 - Standards, naming 2–3
 - Starting AppBuilder 3–3
 - Stateless environment 11–13
 - Static files, adding to Repository 3–26
 - Static objects 3–19
 - editing 3–19
 - running 3–29
 - Static SmartDataObject 11–3
 - Static SmartDataViewer, example 6–11
 - Static viewers
 - building 7–3
 - vs Dynamic 7–3
 - Status display 5–26
 - Storing
 - data 1–3
 - static objects 3–19
 - Structure framework 10–1
 - Structured nodes 9–13
 - Structured PLIP 9–18, 11–18
 - SubMenu category 12–4
 - Suffixes
 - _cl 5–7
 - _obj 2–13, 4–14
 - Super procedures 5–6, 6–6
 - data.p 12–10
 - Support for clients 1–4
 - Suppressing validation 5–16
 - Suspend menu 3–16
 - Suspended User dialog 3–16
 - System information tab 10–52
- ## T
- Tab folders 5–2
 - Table design 2–7

Table names, unqualified 11–11

Table Selector dialog box 11–10

Table trigger editor, ERwin 2–17

TableName attribute 5–22

Tables

- auditing 4–23

- creating data for 4–12

- customizing data 4–18

- dump name 4–21

- filtering 4–18

- gsc_entity_mnemonic 4–12, 9–21

- gsc_object 3–20

- joins 5–17

- maintenance logic 10–36

- naming conventions 2–3

- Object IDs 2–13

- prefix length 4–14

- prefixes 2–5

- Record Version 4–24

- regenerating 4–15

- RowObjUpd 10–15

- structured 9–13

Targets, SBOs 11–42

Template code editor, ERwin 2–17

Templates

- combos 7–13

- customizing 5–13

- ERwin 2–16

- rysttvieww.v 6–9

Temp-tables 10–4

- defining like database table 11–10

- loading 11–2

- populating 11–7

- reasons for use 10–7

- RowObject 11–2

- rowobject 5–3

- RowObjUpd 5–5

_tla suffix 4–17

Token Security Control 13–7

Tokens 13–20

Toolbar and Menu Designer tool 12–1

Toolbar Bands, defining 12–19

Toolbar Designer. See Toolbar and Menu Designer tool.

Toolbar objects 8–27

Toolbars

- adding to windows 12–34

- Browse 8–29

- BrowseToolbarNoUpdate 8–31

- defined 12–29

- DynToolbar 8–31

- FolderPageTop 8–27

- FolderTop 8–29

- FolderTopNoSDO 8–31

- ObjcTop 8–28

Tools

- ERwin 2–16

- Object Generator 4–12, 5–1, 5–10, 11–3

- Toolbar and Menu Designer 12–1

ToolTips 3–18

Transaction logic 11–37

Transaction scoping 10–3, 10–5
within procedures 11–20

Transactional table 2–5

Transferring data 10–3, 11–36

Translating messages 10–43

Tree Node Maintenance window 9–7

TreeView 9–2

TreeView control 12–2

TreeView windows, defining 9–4, 9–11

TreeViews

- example 9–5

- menu structures 9–27

- populating 9–17

- running 9–26

Triggers 11–12
 customizing 11–13
_type suffix 4–17

Types of objects 3–19

U

UDP. See User-defined properties.
Underscore separator 2–3, 4–14
Unique index 4–18
Unique site number 2–13, 4–2
Unique table names 2–4
Unqualified table names 11–11
Updatable joins 10–21
Update buttons 4–20, 8–29
 enabling 8–27
Update Display Field list 4–15
Update transactions 11–26
UpdateableColumns property 11–29
Updates through SBO 11–37
Updating records 10–6
URL for site numbers 4–2
User interface, improving 9–24
User Name restrictions 13–11
User-defined properties 4–23

V

validateDatasetQuery procedure 10–50

Validation
 client-side 10–26
 Dictionary 11–6
 hooks 11–50
 logic 5–5
 procedures 5–18, 10–26
 schema 10–4
 server-side 10–29
 suppressing 5–16

Variables
 local 7–17
 NO-RETURN 10–50

Versioning data 4–23

Viewer Settings dialog box 5–25

Viewers 5–2, 5–9
 adding lookup 7–13
 building 7–3
 changing labels 7–19
 creating 5–25, 6–7
 initializing 7–13
 static vs dynamic 7–3
 static vs. dynamic 6–7

Viewing built-in objects 12–3

VisualizationType attribute 5–22

W

WebClient 1–4

WHERE clause 10–19

Widgets, linking 7–17

Window band vs Windows band 12–25

Windows

- adding toolbars 12–34
- Administration 3–12
- Application Login 3–3
- designing 3–19
- Development 3–14
- Entity Maintenance 4–21
- Filter 4–18
- Product Control 4–9
- Product Maintenance 4–10
- Tree Node Maintenance 9–7
- TreeView 9–4

Windows. See Also SmartWindows.

Wizards

- New PLIP 11–18
- SBO 11–44
- SDO 11–2

writePreTransValidate procedure 5–21

Writing

- distributes applications 10–3
- queries 7–6