OpenEdge Development:
ADM and SmartObjects
## Contents

1. **Overview** .......................... 1–1  
   Introduction .......................... 1–2  
   SmartObject application benefits 1–3  
   Enhanced code reusability 1–3  
   Customizing SmartObjects 1–3  
   ADM classes .......................... 1–4  
   Standard ADM class files and templates. 1–5  
   Custom ADM Class Files. 1–6  
   SmartObjects .......................... 1–8  
   SmartObject types .......................... 1–8  
   The lifecycle of a SmartObject 1–10  
   SmartObject templates and masters 1–10  
   Instantiation .......................... 1–11  
   SmartLinks .......................... 1–12  
   Properties .......................... 1–13  
   Super procedures .......................... 1–14  
   External procedures, internal procedures, and functions 1–14  
   Super procedures in the ADM 1–16  

2. **SmartObjects** .......................... 2–1  
   SmartObject summary .......................... 2–3  
   Accessing SmartObjects 2–4  
   SmartObject instance properties 2–4  
   AppBuilder-maintained procedures 2–4  
   SmartContainers .......................... 2–5  
   SmartContainers and their contents 2–5  
   Paging .......................... 2–6  
   Pass-through links. 2–6  
   SmartWindows .......................... 2–6  
   SmartFrames. 2–6  
   SmartDialogs .......................... 2–7
<table>
<thead>
<tr>
<th>SmartDataObjects</th>
<th>2–8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmartDataObject instance properties</td>
<td>2–8</td>
</tr>
<tr>
<td>SmartDataObject usage notes</td>
<td>2–10</td>
</tr>
<tr>
<td>SmartDataBrowsers</td>
<td>2–14</td>
</tr>
<tr>
<td>SmartDataBrowser instance properties</td>
<td>2–14</td>
</tr>
<tr>
<td>Dynamic versus static SmartDataBrowsers</td>
<td>2–17</td>
</tr>
<tr>
<td>SmartDataBrowser usage notes</td>
<td>2–18</td>
</tr>
<tr>
<td>SmartDataViewers</td>
<td>2–21</td>
</tr>
<tr>
<td>SmartDataViewer instance properties</td>
<td>2–21</td>
</tr>
<tr>
<td>SmartDataViewer usage notes</td>
<td>2–22</td>
</tr>
<tr>
<td>SmartDataFields</td>
<td>2–23</td>
</tr>
<tr>
<td>SmartDataField instance properties</td>
<td>2–24</td>
</tr>
<tr>
<td>SmartDataField usage notes</td>
<td>2–24</td>
</tr>
<tr>
<td>SmartPanels</td>
<td>2–26</td>
</tr>
<tr>
<td>SmartPanel instance properties</td>
<td>2–27</td>
</tr>
<tr>
<td>SmartPanel usage notes</td>
<td>2–29</td>
</tr>
<tr>
<td>SmartFolders</td>
<td>2–30</td>
</tr>
<tr>
<td>SmartFolder instance properties</td>
<td>2–30</td>
</tr>
<tr>
<td>SmartFolder usage notes</td>
<td>2–31</td>
</tr>
<tr>
<td>SmartFilters</td>
<td>2–32</td>
</tr>
<tr>
<td>SmartFilter instance properties</td>
<td>2–32</td>
</tr>
<tr>
<td>SmartFilter usage notes</td>
<td>2–38</td>
</tr>
<tr>
<td>SmartToolbars</td>
<td>2–39</td>
</tr>
<tr>
<td>SmartToolbar instance properties</td>
<td>2–40</td>
</tr>
<tr>
<td>SmartToolbar usage notes</td>
<td>2–42</td>
</tr>
<tr>
<td>SmartSelects</td>
<td>2–43</td>
</tr>
<tr>
<td>SmartSelect instance properties</td>
<td>2–43</td>
</tr>
<tr>
<td>SmartSelect usage notes</td>
<td>2–48</td>
</tr>
<tr>
<td>Simple SmartObjects</td>
<td>2–51</td>
</tr>
</tbody>
</table>

3. **SmartLinks**                          | 3–1 |
| SmartLink overview                      | 3–2 |
| SmartLink types                         | 3–2 |
| Source and target objects               | 3–4 |
| SmartLink events                        | 3–5 |
| Container SmartLink                     | 3–6 |
| Navigation SmartLink                    | 3–6 |
| Page SmartLink                          | 3–7 |
| PageN SmartLink                         | 3–8 |
| TableIO SmartLink                       | 3–8 |
| GroupAssign SmartLink                   | 3–9 |
| Data SmartLink                          | 3–9 |
| Update SmartLink                        | 3–10|
| Commit SmartLink                        | 3–10|
| Filter SmartLink                        | 3–11|
| Invoking behavior in other linked objects | 3–12|
| Pass-through links                      | 3–13|
| Getting records from an additional source | 3–15|
| Using SmartLinks                        | 3–17|
| Defining SmartLinks in Progress applications | 3–18|
| Defining static SmartLinks              | 3–18|
| Defining dynamic SmartLinks             | 3–19|
## 4. Data Management in the ADM

- Displaying and browsing data sets
  - SmartDataObjects
  - SmartDataViewers
  - SmartDataBrowsers
  - Initializing the foreign key fields
  - Compatibility checking
- Filtering records
- Using SmartPanels and the SmartToolbar
  - Using SmartPanels
  - Using SmartToolbars
- Transactions and record locking

## 5. SmartObject Interactions

- General structure of a SmartObject
  - Directory and file structure
  - Support include files
  - Super procedures
  - Property and prototype include files
  - Prototype include files
  - SmartObject file relationships
- RUN protocol
  - RUN statement
- Localizing standard SmartObject behavior
- ADM properties
  - ADMProps temp-table and object properties
  - Get and set functions for object properties
  - (get) and (set) pseudo-syntax for object properties
  - Instance properties
  - Translatable properties
  - Functions for accessing properties
  - Dynamic properties

## 6. Advanced ADM Topics

- Parameterizing SmartObject instances as variables
- Writing local event procedures and functions
  - Writing a local event procedure
  - Writing a local event function
- Monitoring SmartObject applications
  - Procedure Object Viewer
  - The Application Debugger
  - Pro*Spy Plus
- AppBuilder requirements for SmartObjects
- Using GroupAssign SmartLinks

## 7. Developing Your Application’s Business Logic

- SmartDataObject features and operating characteristics
- SmartDataObject query and update operations
  - Query operations and the RowObject temp-table
  - Update operations and the RowObjUpd temp-table
  - Nodata fields in the RowObject and RowObjUpd temp-table
- ADM event procedures and functions in the SmartDataObject
  - Opening a database query
  - Dynamic query manipulation
## Contents

**SmartDataObjects and SmartLinks** ............................................. 7–12
  Validation procedures ......................................................... 7–13
    Data Dictionary validation .............................................. 7–13
    Field-level validation procedures .................................. 7–13
    Record-level validation procedures ................................. 7–15
    Transaction-level validation procedures ......................... 7–16
    Database trigger procedures ......................................... 7–18
  Running SmartDataObjects in a distributed environment .......... 7–19
    Setting up the SmartDataObject master for conditional compilation 7–20
    AppBuilder modifications to the SmartDataObject master .......... 7–20
    Initializing the SmartDataObject ..................................... 7–22
  Managing SmartDataObjects in distributed mode ................. 7–23
    SmartDataObject deployment files .................................. 7–23
    Service choices and SmartDataObject execution ............... 7–25
    AppServer session operating modes and SmartDataObjects ........ 7–26
  AppServer-related SmartDataObject properties .................. 7–27
  Running SmartDataObjects from a non-Progress client .......... 7–28
    Batch-mode operations ................................................ 7–28
    Single-row operations ................................................ 7–28
  Java applications and SmartDataObjects ............................ 7–29

8. **Developing ADM Extensions** ............................................. 8–1
   Introduction ........................................................................ 8–2
   Creating a new class at the bottom of the ADM class hierarchy .... 8–4
     Planning issues .......................................................... 8–4
     Creating the new class files ....................................... 8–5
     Modifying the new class files ..................................... 8–10
     Customizing an existing ADM class ............................... 8–10
   Adding application logic to your new or customized ADM class .... 8–12
     Custom class files ......................................................... 8–12
     Design rules ..................................................................... 8–13
     Adding properties ....................................................... 8–14
     Writing super procedures .......................................... 8–15
   Examples ........................................................................... 8–18
     Creating the myviewer class ......................................... 8–18
     Customizing the visual class .................................... 8–23

A. **ADM Standard and Custom Class Files** ............................. A–1
   Container class files ...................................................... A–2
   Standard class files ......................................................... A–2
   Templates ........................................................................ A–14
   Custom class files ........................................................ A–20

B. **ADM1 to ADM2 Conversion Utility** .................................. B–1
   Conversion support for ADM SmartObjects in later versions .... B–2
     Code changes to SmartObject container type objects .......... B–3
     Additional miscellaneous points .................................. B–3
   Using the ADM1 to ADM2 conversion utility ...................... B–4
     Accessing the conversion utility .................................. B–4
     Defining conversion options ....................................... B–5
     Building the list of files to process ............................. B–7
     Performing the conversion .......................................... B–8
     Reviewing conversion results ..................................... B–10
   Customizing the conversion utility ................................ B–12

Index .................................................................................. Index–1
Tables

Table 1–1: Standard ADM class files .............................................. 1–5
Table 1–2: Custom ADM class files ............................................. 1–6
Table 1–3: SmartObject summary ................................................. 1–8
Table 2–1: SmartObject types .................................................... 2–3
Table 2–2: SmartWindow files .................................................... 2–6
Table 2–3: SmartFrame files ..................................................... 2–7
Table 2–4: SmartDialog files ..................................................... 2–7
Table 2–5: SmartDataObject files .............................................. 2–8
Table 2–6: SmartDataBrowser files .......................................... 2–14
Table 2–7: SmartDataViewer files ........................................... 2–21
Table 2–8: SmartDataField files ............................................ 2–23
Table 2–9: SmartPanel files ................................................... 2–26
Table 2–10: SmartFolder files ............................................... 2–30
Table 2–11: SmartFilter files .................................................. 2–32
Table 2–12: SmartToolbar files .............................................. 2–39
Table 2–13: SmartSelect files ................................................. 2–43
Table 3–1: ADM-recognized SmartLink types ................................ 3–2
Table 3–2: Container events .................................................... 3–6
Table 3–3: Navigation events ................................................... 3–7
Table 3–4: Page events .......................................................... 3–7
Table 3–5: TableIO internal procedures .................................... 3–8
Table 3–6: GroupAssign events .............................................. 3–9
Table 3–7: Data events .......................................................... 3–9
Table 3–8: Commit Events ...................................................... 3–10
Table 8–1: Basic tab fill-in fields ............................................. 8–6
Table B–1: Setting up file conversion options ............................. B–5
Table B–2: Files in the browser ............................................... B–7
Table B–3: Refining file selection actions .................................. B–8
Table B–4: Conversion status labels displayed in the browser .... B–10
Table B–5: ADM1 to ADM2 SmartObject conversions ................ B–10
Table B–6: Specific file conversion changes ............................. B–11
Figures

Figure 1–1: ADM hierarchical class structure ............................. 1–4
Figure 1–2: SmartObject lifecycle ........................................... 1–10
Figure 1–3: External procedures, internal procedures, and functions 1–15
Figure 1–4: Inheritance and the super procedure stack .................... 1–16
Figure 2–1: SmartContainer and contained SmartObject .................. 2–5
Figure 2–2: SmartDataObject Properties dialog box ....................... 2–8
Figure 2–3: Dynamic SmartDataBrowser Properties dialog box ........ 2–15
Figure 2–4: SmartDataBrowser Properties dialog box ..................... 2–16
Figure 2–5: SmartDataViewer instance properties dialog box ............ 2–21
Figure 2–6: SmartDataField instance properties dialog box .............. 2–24
Figure 2–7: Navigation SmartPanel Attributes dialog box ............... 2–27
Figure 2–8: Update SmartPanel instance properties dialog box ........ 2–28
Figure 2–9: SmartFolder Properties dialog box .......................... 2–30
Figure 2–10: SmartFolder page 0 ........................................... 2–31
Figure 2–11: SmartFilter Properties dialog box ........................... 2–33
Figure 2–12: SmartToolbar Properties dialog box ......................... 2–40
Figure 2–13: SmartToolbar menus .......................................... 2–40
Figure 2–14: SmartToolbar toolbar ......................................... 2–41
Figure 2–15: SmartSelect Properties dialog box ......................... 2–43
Figure 2–16: Keyboard Event dialog box .................................. 2–48
Figure 2–17: Simple SmartObject .......................................... 2–51
Figure 3–1: Pass-through link .............................................. 3–13
Figure 3–2: Pass-through link: inner connection .......................... 3–14
Figure 3–3: Pass-through link: inner and outer connections ............. 3–15
Figure 4–1: Inheriting foreign key field values ............................ 4–4
Figure 5–1: Super procedures and master interactions ................... 5–3
Figure 5–2: Include file structure for SmartDataViewer ................. 5–5
Figure 5–3: Super procedure schematic for SmartDataViewer .......... 5–6
Figure 6–1: Procedure Object Viewer icon on PRO*Tools palette ....... 6–6
Figure 6–2: Procedure Object Viewer ...................................... 6–6
Figure 6–3: Selecting procedure objects and internal entries .......... 6–7
Figure 6–4: Pro*Spy Plus icon on PRO*Tools palette .................... 6–9
Figure 6–5: Pro*Spy Plus window .......................................... 6–9
Figure 6–6: Grouping updates with GroupAssign SmartLinks .......... 6–11
Figure 7–1: The SmartDataObject and its temp-tables .................... 7–3
Figure 7–2: Java open client access to a remote SmartDataObject .... 7–29
Figure 8–1: ADM hierarchical class structure for SmartFrames ........ 8–2
Figure 8–2: New ADM Class menu option ................................ 8–5
Figure 8–3: New ADM Class (initial) ..................................... 8–5
Figure 8–4: Basic tab with fill-in fields ................................... 8–6
Figure 8–5: Custom Files tab ............................................... 8–8
Figure B–1: SmartObject conversion utility icon ......................... B–4
Figure B–2: ADM1 to ADM2 SmartObject Conversion Utility window B–5
Figure B–3: Conversion Utility window after a conversion ............... B–9
## Procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcust.i</td>
<td>4–6</td>
</tr>
<tr>
<td>dcust.i</td>
<td>4–6</td>
</tr>
<tr>
<td>dcustreverse.i</td>
<td>4–6</td>
</tr>
<tr>
<td>smrtprop.i</td>
<td>5–12</td>
</tr>
<tr>
<td>dataprop.i</td>
<td>5–14</td>
</tr>
<tr>
<td>containr.i</td>
<td>A–2</td>
</tr>
<tr>
<td>contain.prop.i</td>
<td>A–4</td>
</tr>
<tr>
<td>containr.p</td>
<td>A–7</td>
</tr>
<tr>
<td>containr.to.i</td>
<td>A–12</td>
</tr>
<tr>
<td>containr.to.m</td>
<td>A–14</td>
</tr>
<tr>
<td>containr.custom.i</td>
<td>A–20</td>
</tr>
<tr>
<td>containr.propcustom.i</td>
<td>A–22</td>
</tr>
<tr>
<td>containr.propcustom.p</td>
<td>A–24</td>
</tr>
<tr>
<td>containr.proto.custom.i</td>
<td>A–26</td>
</tr>
<tr>
<td>containr.exclcustom.i</td>
<td>A–26</td>
</tr>
<tr>
<td>containr.defscustom.i</td>
<td>A–26</td>
</tr>
</tbody>
</table>
Preface

This Preface contains the following sections:

- Purpose
- Audience
- Organization
- Typographical conventions
- Examples of syntax descriptions
- OpenEdge messages
- Third party acknowledgements
Purpose

This book describes the Application Development Model (ADM) and explains how to use Progress® SmartObjects in Progress 4GL database applications.

Audience

This book is intended for Progress 4GL application developers.

For the latest documentation updates see the OpenEdge Product Documentation category on PSDN http://www.psdn.com/library/kbcategory.jspa?categoryID=129.

Organization

Chapter 1, “Overview”

Provides an overview of the ADM and SmartObjects.

Chapter 2, “SmartObjects”

Provides a summary of SmartObject capabilities and types, followed by a description of each SmartObject.

Chapter 3, “SmartLinks”

Provides an overview of SmartLinks and a list of types, followed by a description of the relationship established for each SmartLink type and some programming considerations.

Chapter 4, “Data Management in the ADM”

Describes how various SmartObjects work together to manage database records in your applications.

Chapter 5, “SmartObject Interactions”

Describes how SmartObjects interact; in particular, how they communicate, get and set ADM properties, and pass data.

Chapter 6, “Advanced ADM Topics”

Discusses advanced ADM topics, including parameterizing SmartObject instances as variables, customizing ADM internal procedures and functions, adding your own SmartObjects, monitoring SmartObjects applications, and AppBuilder requirements for SmartObjects.

Chapter 7, “Developing Your Application’s Business Logic”

Provides information on developing and deploying SmartObject-based applications in heterogeneous (single or combined client/server, n-tier, or Web-enabled) environments.
Chapter 8, “Developing ADM Extensions”

Describes how to use the AppBuilder’s New ADM Class tool and ADM custom class files to extend the ADM classes.

Appendix A, “ADM Standard and Custom Class Files”

Shows the contents of the class files for the viewer class, as an example of the class files for all ADM classes.

Appendix B, “ADM1 to ADM2 Conversion Utility”

Describes how to use the conversion utility to convert ADM1 (Versions 8.1 and 8.2) SmartObjects to ADM2 (Version 9 and later) SmartObjects.

Typographical conventions

This manual uses the following typographical conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Bold typeface indicates commands or characters the user types, or the names of user interface elements.</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Italic typeface indicates the title of a document, provides emphasis, or signifies new terms.</td>
</tr>
<tr>
<td>SMALL, <strong>BOLD CAPITAL LETTERS</strong></td>
<td>Small, bold capital letters indicate OpenEdge™ key functions and generic keyboard keys; for example, GET and CTRL.</td>
</tr>
<tr>
<td><strong>KEY1</strong> <strong>KEY2</strong></td>
<td>A hyphen between key names indicates a <em>simultaneous</em> key sequence: you press and hold down the first key while pressing the second key. For example, CTRL-X.</td>
</tr>
<tr>
<td>KEY1 KEY2</td>
<td>A space between key names indicates a <em>sequential</em> key sequence: you press and release the first key, then press another key. For example, ESCAPE H.</td>
</tr>
<tr>
<td><strong>Syntax:</strong></td>
<td></td>
</tr>
<tr>
<td>Fixed width</td>
<td>A fixed-width font is used in syntax statements, code examples, and for system output and filenames.</td>
</tr>
<tr>
<td><em>Fixed-width</em> italics*</td>
<td>Fixed-width italics indicate variables in syntax statements.</td>
</tr>
<tr>
<td><strong>Fixed-width</strong> <strong>bold</strong></td>
<td>Fixed-width bold indicates variables with special emphasis.</td>
</tr>
<tr>
<td>UPPERCASE fixed width</td>
<td>Uppercase words are Progress® 4GL language keywords. Although these always are shown in uppercase, you can type them in either uppercase or lowercase in a procedure.</td>
</tr>
<tr>
<td>Period (.) or colon (:)</td>
<td>All statements except DO, FOR, FUNCTION, PROCEDURE, and REPEAT end with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon.</td>
</tr>
<tr>
<td>[]</td>
<td>Large brackets indicate the items within them are optional.</td>
</tr>
</tbody>
</table>
Examples of syntax descriptions

In this example, ACCUM is a keyword, and aggregate and expression are variables:

```
ACCUM aggregate expression
```

FOR is one of the statements that can end with either a period or a colon, as in this example:

```
FOR EACH Customer:
    DISPLAY Name.
END.
```

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

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

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

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

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

```
{ &argument-name }
```
In this example, EACH, FIRST, and LAST are optional, but you can choose only one of them:

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

In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:

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

In this example, you must specify MESSAGE and at least one expression or SKIP [ (n) ], and any number of additional expression or SKIP [ ( n ) ] is allowed:

```
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 \}:

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

**Long syntax descriptions split across lines**

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

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

**Syntax**

```
WITH [ ACCUM max-length ] [ expression DOWN ]
   [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
   [ STREAM-IO ]
```
Complex syntax descriptions with both required and optional elements

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, ASSIGN requires either one or more field entries or one record. Options available with field or record are grouped with braces and brackets:

Syntax

\[
\text{ASSIGN} \{ \{ \text{FRAME} \ frame \} \\
\{ \text{field} \ [ = \ expression \] \} \\
\{ \text{WHEN} \ expression \} \\
\} \ldots \\
\| \{ \text{record} \ [ \text{EXCEPT} \ field \ldots \] \}
\]

OpenEdge messages

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

- **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.

- **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before 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 OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge 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 after execution messages.

- Returns to the Progress Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.

- Halts processing of a procedure and returns immediately to the Progress Procedure Editor. This does not happen often.

- Terminates the current session.
OpenEdge messages end with a message number in parentheses. In this example, the message number is 200:

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

If you encounter an error that terminates OpenEdge, note the message number before restarting.

**Obtaining more information about OpenEdge messages**

On Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

- Choose **Help → Recent Messages** to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.
- Choose **Help → Messages** and then enter the message number to display a description of a specific OpenEdge message.
- In the Progress Procedure Editor, press the **HELP** key or **F1**.

On UNIX platforms, use the Progress **PRO** command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

To use the **PRO** command to obtain a message description by message number:

1. Start the Progress Procedure Editor:

   ```
   install-dir/dlc/bin/pro
   ```

2. Press **F3** to access the menu bar, then choose **Help → Messages**.

3. Type the message number and press **ENTER**. Details about that message number appear.

4. Press **F4** to close the message, press **F3** to access the Progress Procedure Editor menu, and choose **File → Exit**.

**Third party acknowledgements**

OpenEdge includes Imaging Technology copyrighted by Snowbound Software 1993-2003.


OpenEdge includes software developed by the Apache Software Foundation (http://www.apache.org/). Copyright © 1999 The Apache Software Foundation. All rights reserved (Xerces C++ Parser (XML) and Xerces2 Java Parser (XML)); Copyright © 1999-2002 The Apache Software Foundation. All rights reserved (Xerces Parser (XML); and Copyright © 2000-2003 The Apache Software Foundation. All rights reserved (Ant). The names “Apache,” “Xerces,” “ANT,” and “Apache Software Foundation” must not be used to endorse or promote products derived from this software without prior written permission. Products derived from
this software may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission of the Apache Software Foundation. For written permission, please contact apache@apache.org. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.


OpenEdge includes software developed by Vermont Creative Software. Copyright © 1988-1991 by Vermont Creative Software.

OpenEdge includes code licensed from RSA Security, Inc. Some portions licensed from IBM are available at http://oss.software.ibm.com/icu4j/.

OpenEdge includes the UnixWare platform of Perl Runtime authored by Kiem-Phong Vo and David Korn. Copyright © 1991, 1996 by AT&T Labs. Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software. THIS SOFTWARE IS BEING PROVIDED “AS IS”, WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHORS NOR AT&T LABS MAKE ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

OpenEdge includes the RSA Data Security, Inc. MD5 Message-Digest Algorithm. Copyright ©1991-2, RSA Data Security, Inc. Created 1991. All rights reserved.

OpenEdge includes software developed by the World Wide Web Consortium. Copyright © 1994-2002 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All rights reserved. This work is distributed under the W3C® Software License [http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231] in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

OpenEdge includes Sonic software, which includes software developed by Apache Software Foundation (http://www.apache.org/). Copyright © 1999-2000 The Apache Software Foundation. All rights reserved. The names “Ant”, “Axis”, “Xalan”, “FOP,” “The Jakarta Project”, “Tomcat”, “Xerces” and/or “Apache Software Foundation” must not be used to endorse or promote products derived from the Product without prior written permission. Any product derived from the Product may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission. For written permission, please contact apache@apache.org.

OpenEdge includes Sonic software, which includes the JMX Technology from Sun Microsystems, Inc. Use and Distribution is subject to the Sun Community Source License available at http://sun.com/software/communitysource.

OpenEdge includes Sonic software, which includes software developed by the ModelObjects Group (http://www.modelobjects.com). Copyright © 2000-2001 ModelObjects Group. All rights reserved. The name “ModelObjects” must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this
software may not be called “ModelObjects”, nor may “ModelObjects” appear in their name, without prior written permission. For written permission, please contact djacobs@modelobjects.com.

OpenEdge includes Sonic software, which includes files that are subject to the Netscape Public License Version 1.1 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/NPL/. Software distributed under the License is distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is Mozilla Communicator client code, released March 31, 1998. The Initial Developer of the Original Code is Netscape Communications Corporation. Portions created by Netscape are Copyright © 1998-1999 Netscape Communications Corporation. All Rights Reserved.

OpenEdge includes software Copyright © 2003-2006, Terence Parr All rights reserved. Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes ICU software 1.8 and later - Copyright © 1995-2003 International Business Machines Corporation and others All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

OpenEdge includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). Copyright © 1998-2007 The OpenSSL Project. All rights reserved. This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com). Copyright © 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved. The names “OpenSSL Toolkit” and “OpenSSL Project” must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org. Products derived from this software may not be called “OpenSSL” nor may “OpenSSL” appear in their names without prior written permission of the OpenSSL Project. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes Sonic software which includes a version of the Saxon XSLT and XQuery Processor from Saxonica Limited that has been modified by Progress Software Corporation. The contents of the Saxon source code and the modified source code file (Configuration.java) are subject to the Mozilla Public License Version 1.0 (the “License”); you may not use these files except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/ and a copy of the license (MPL-1.0.html) can also be found in the installation directory, in the Docs7.5/third_party_licenses folder, along with a copy of the modified code (Configuration.java); and a description of the modifications can be found in the Progress SonicMQ and Progress Sonic ESB v7.5 README files. Software distributed under the License is distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND,
either express or implied. See the License for the specific language governing rights and
limitations under the License. The Original Code is The SAXON XSLT and XQuery Processor
from Saxonica Limited. The Initial Developer of the Original Code is Michael Kay
http://www.saxonica.com/products.html). Portions created by Michael Kay are Copyright
© 2001-2005. All rights reserved. Portions created by Progress Software Corporation are
Copyright © 2007. All rights reserved.

OpenEdge includes software developed by IBM. Copyright © 1995-2003 International
Business Machines Corporation and others. All rights reserved. Permission is hereby granted,
free of charge, to any person obtaining a copy of this software and associated documentation
files (the “Software”), to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, publish, distribute, and/or sell copies of the Software,
and to permit persons to whom the Software is furnished to do so, provided that the above
copyright notice(s) and this permission notice appear in all copies of the Software and that both
the above copyright notice(s) and this permission notice appear in supporting documentation.
Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either
express or implied. See the License for the specific language governing rights and limitations
under the License agreement that accompanies the product. Except as contained in this notice,
the name of a copyright holder shall not be used in advertising or otherwise to promote the sale,
use or other dealings in this Software without prior written authorization of the copyright holder.

OpenEdge includes Sonic software, which includes software developed by ExoLab Project
(http://www.exolab.org/). Copyright © 2000 Intalio Inc. All rights reserved. The names
“Castor” and/or “ExoLab” must not be used to endorse or promote products derived from the
Products without prior written permission. For written permission, please contact
info@exolab.org. Exolab, Castor and Intalio are trademarks of Intalio Inc.

OpenEdge includes Sonic software, which includes software Copyright © 1999 CERN -
European Organization for Nuclear Research. Permission to use, copy, modify, distribute and
sell this software and its documentation for any purpose is hereby granted without fee, provided
that the above copyright notice appear in all copies and that both that copyright notice and this
permission notice appear in supporting documentation. CERN makes no representations about
the suitability of this software for any purpose. It is provided “as is” without expressed or
implied warranty.

OpenEdge includes Sonic software, which includes software developed by the University
Corporation for Advanced Internet Development http://www.ucaid.edu Internet2 Project.
Copyright © 2002 University Corporation for Advanced Internet Development, Inc. All rights
reserved. Neither the name of OpenSAML nor the names of its contributors, nor Internet2, nor
the University Corporation for Advanced Internet Development, Inc., nor UCAID may be used
to endorse or promote products derived from this software and products derived from this
software may not be called OpenSAML, Internet2, UCAID, or the University Corporation for
Advanced Internet Development, nor may OpenSAML appear in their name without prior
written permission of the University Corporation for Advanced Internet Development. For
written permission, please contact opensaml@opensaml.org.

OpenEdge includes DataDirect products for the Microsoft SQL Server database which contains
a licensed implementation of the Microsoft TDS Protocol.

OpenEdge includes Sonic software, which includes code licensed from Mort Bay Consulting
Pty. Ltd. The Jetty Package is Copyright © 1998 Mort Bay Consulting Pty. Ltd. (Australia) and
others.
OpenEdge includes software authored by David M. Gay. Copyright © 1991, 2000, 2001 by Lucent Technologies (dtoa.c); Copyright © 1991, 1996 by Lucent Technologies (g_fmt.c); and Copyright © 1991 by Lucent Technologies (rnd_prod.s). Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software. THIS SOFTWARE IS BEING PROVIDED “AS IS”, WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

OpenEdge includes software authored by David M. Gay. Copyright © 1998-2001 by Lucent Technologies All Rights Reserved (decstrtod.c; strtodg.c); Copyright © 1998, 2000 by Lucent Technologies All Rights Reserved (decstrtof.c; strtod.c); Copyright © 1998 by Lucent Technologies All Rights Reserved (dmisc.c; gdtoa.h; gethex.c; gmisc.c; sum.c); Copyright © 1998, 1999 by Lucent Technologies All Rights Reserved (gdtoa.c; misc.c; smisc.c; ulp.c); Copyright © 1998-2000 by Lucent Technologies All Rights Reserved (gdtoaimp.h); Copyright © 2000 by Lucent Technologies All Rights Reserved (hd_init.c). Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of Lucent or any of its entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

OpenEdge includes Infragistics NetAdvantage for .NET v2008 Vol 1 CLR 2.0. Copyright © 1996-2008 Infragistics, Inc. All rights reserved.
Overview

This chapter provides an overview of the Application Development Model (ADM). The chapter contains the following sections:

- Introduction
- ADM classes
- SmartObjects
- SmartLinks
- Properties
- Super procedures
The Application Development Model (ADM) is a methodology for building Progress® 4GL applications based on reusable components called SmartObjects. It was updated for OpenEdge™ to take advantage of enhancements to the Progress® 4GL.

All support files for the ADM architecture are in a directory structure under src/adm2; however, Progress also maintains support for SmartObject applications built in Version 8 by including the ADM Version 1.1 support files under a separate directory structure under src/adm and parallel gui directories.

The ADM architecture has four basic components:

- A hierarchical set of object classes from which to develop SmartObjects. These are reusable application objects (visualizable or nonvisualizable) that map to source files or compilation units.
- SmartLinks, to establish a messaging system by defining links (communication paths) among SmartObjects.
- Properties, to define the public data associated with SmartObjects and to get and set these data values.
- Super procedures, to perform SmartObject actions.

The design of this architecture relies on several important features of the Progress 4GL: super procedures, persistent procedures, internal procedures and functions, preprocessor names, temporary tables, and include files. This design is implemented in the body of the architecture’s support code and is fully supported by the AppBuilder. For information about the AppBuilder, see OpenEdge Development: AppBuilder.

The following sections summarize the benefits of using SmartObjects in Progress 4GL applications and provide notes on additional SmartObjects advantages. The rest of this chapter is an overview of the components of the ADM architecture.
SmartObject application benefits

SmartObjects offer a dimension of object integration and flexibility of use that you cannot achieve with basic objects (Progress 4GL widgets). Unlike basic objects, which are built into the 4GL, each SmartObject is represented by an entire 4GL procedure file.

The benefits of using SmartObjects in Progress 4GL applications are summarized below:

- **Application assembly** — Each SmartObject master is a predefined application component that can encapsulate more complex functionality than an individual basic object, so you can assemble more sophisticated applications in less time.

- **Object interoperability** — Because the ADM gives SmartObjects standardized ways of interacting with each other, it is easier to integrate individual object instances into applications.

- **Code reusability** — Because a single object can have multiple run-time instances, you can create code that can be used repeatedly.

- **Maintainability** — Your development process is organized so as to make complex applications easy to maintain. SmartObject code is easily maintainable because changes to a single SmartObject master are reflected in all instances of that SmartObject.

- **Standard look** — Different applications can instantiate the same SmartObject master, so you can establish a consistent look and feel across multiple applications.

- **Module independence** — You can write each object without specific knowledge of other objects, thus reducing large-scale problems into application components that you can work on independently.

Enhanced code reusability

SmartObjects are defined in external procedures, thus creating a potential for code reuse far greater than that afforded by basic objects alone. There is no limit to the number of different application files that can instantiate the same SmartObject master; moreover, you can assign different run-time properties to each instance while still creating each from a single master.

The potential for code reuse that SmartObjects create makes it much easier to maintain complex, enterprise-wide applications, because a change you make in a SmartObject master is reflected in every instance of the SmartObject in every application that uses it.

Customizing SmartObjects

Once you become familiar with the mechanics of the ADM, you can customize a SmartObject type or override its default behaviors. You also can create new SmartObjects (represented by new SmartObject templates), giving them different behaviors better suited to the needs of your applications. For information about creating new SmartObject types and customizing existing types, see Chapter 6, “Advanced ADM Topics.”
ADM classes

The ADM classes, collectively, are a system of Progress include files and collections of internal procedures and functions. Each class is a particular subset of these files that defines a standard behavior specific to that class. The various classes reference each other through include files, and relate to each other in a hierarchical fashion: lower-level classes inherit behavior and properties from higher-level classes.

Figure 1–1 illustrates the ADM classes and their hierarchical structure.

![ADM hierarchical class structure](image)

Figure 1–1: ADM hierarchical class structure

The objects derived from a particular ADM class are described according to the class from which they inherit. For example:

- Objects derived from the `visual` class or the classes that inherit from it—`panel`, `datavis`, `container`, `field`, `filter`, `action`, `viewer`, `browser`, and `toolbar`—are called visual objects.

- Objects derived from the `datavis` class and the `viewer` and `browser` classes that inherit from it are called data visualization objects.

- Objects derived from the `query` class and the `data` class that inherits from it are called query objects.

User-defined classes that inherit from an existing class use the same terminology; for example, a user-defined class that inherits from the `filter` class is a visual object.

Each ADM class is supported by two sets of class files: a standard set and a custom set. In addition to the class files, the ADM provides a set of templates, each of which references the primary include file of a specific class and serves as a base class for developing one of the many types of SmartObjects that you can use, along with basic and ActiveX objects, to build a Progress 4GL application. See the next several sections for details on standard and class files and on templates.

The body of code that supports ADM classes resides in the `%DLC%/src/adm2` (Windows) or `$DLC/src/adm2` (UNIX) directory and its subdirectories and is installed as part of OpenEdge. For details, see *OpenEdge Getting Started: Installation and Configuration*. 
Standard ADM class files and templates

Each ADM class is supported by a set of standard class files of specific types: a primary include file, a property file, a super procedure, and a prototype file. These files, along with a set of templates, provide the basis for the standard SmartObjects available in the ADM. Each ADM class contains one standard class file of each type, but class files of the same type are set up differently for different classes. For example, the property file for the panel class is different from the property file for the viewer class. The standard class files for all ADM classes are in the %DLC%\src\adm2 (Windows) or $DLC/src/adm2 (UNIX) directory.

Table 1–1 describes the standard class files for each class. In the Filename column, type is a character string indicating the particular class; for example, view or viewer for the viewer class.

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary include file</td>
<td>type.i</td>
<td>• Starts the super procedure for the class</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Can initialize some properties</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• References the property file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Is referenced directly in a template or in another class’ primary include file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• References the custom primary include file, if relevant</td>
</tr>
<tr>
<td>Property file</td>
<td>typeprop.i</td>
<td>• Defines properties</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• References the prototype file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Can define instance properties</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Can define the instance property dialog</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• References the custom instance definition file, if relevant</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• References the custom property file, if relevant</td>
</tr>
<tr>
<td>Super procedure file</td>
<td>type.p</td>
<td>• Contains functions and internal procedures that define the behavior for the class</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• References the property file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Is run only once for the entire session</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• References the custom exclude definition file, if relevant</td>
</tr>
<tr>
<td>Prototype file</td>
<td>typeprto.i</td>
<td>• Defines prototypes for all internal entries (functions, internal procedures) of the super procedure for the class</td>
</tr>
</tbody>
</table>

The type.i, typeprop.i, and type.p custom class files are described in the “Custom ADM Class Files” section on page 1–6.

For example, the following standard class files support the viewer class:

- Primary include file: viewer.i
- Property file: viewprop.i
- Super procedure: viewer.p
- Prototype file: viewprto.i
For examples of the contents of standard class files, see Appendix A, “ADM Standard and Custom Class Files.”

**Templates**

Unlike the standard class files, templates reside in the `template` subdirectory of the `%DLC%\src\adm2` (Windows) or `$DLC/src/adm2` (UNIX) directory. They are files that have at least a design-time visualization in the AppBuilder and, accordingly, have the suffix `.w`; for example, `viewer.w`. A SmartObject template serves as a basis for including a SmartObject in your application. Conceptually, it is a procedure file that serves as a general starting point for building one or more specific SmartObject masters (SmartObject external procedures).

A template contains very little code; generally, it contains only the following:

- A reference to the primary include file of the class on which the associated SmartObjects are based
- Definitions of Procedure Type, Supported Links, and other constant values
- A base definition of the SmartObject’s frame or visualization, if appropriate
- The code used to execute the template’s development wizard, if any

For a description of how SmartObject masters and templates relate, see the “SmartObject templates and masters” section on page 1–10. For an example of the contents of a template file, see Appendix A, “ADM Standard and Custom Class Files.”

**Custom ADM Class Files**

Each ADM class also is supported by a set of custom class files of specific types: a primary include file, a property file, a super procedure, a prototype file, an exclude definition file, and an instance definition file. These files can provide the basis for developing custom SmartObjects and/or classes based on existing SmartObjects and classes. As with the standard class files, each ADM class contains one file of each type. Custom ADM class files reside in the `custom` subdirectory of the `%DLC%\src\adm2` (Windows) or `$DLC/src/adm2` (UNIX) directory.

Table 1–2 describes the custom class files for each class. As in Table 1–1, `type` in the Filename column is a character string indicating the particular class; for example, `view` or `viewer` for the `viewer` class.

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom primary include file</td>
<td>typecustom.i</td>
<td>• Starts the custom super procedure for the class (commented out in the standard primary include file)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Can initialize some properties</td>
</tr>
<tr>
<td>Custom property file</td>
<td>typepropcustom.i</td>
<td>• Defines properties</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• References the custom prototype file</td>
</tr>
</tbody>
</table>
For example, the following custom class files support the viewer class:

- Custom primary include file: `viewercustom.i`
- Custom property file: `viewpropcustom.i`
- Custom super procedure: `viewercustom.p`
- Custom prototype file: `viewprotocustom.i`
- Custom exclude definition file: `viewerexclcustom.i`
- Custom instance definition file: `viewerdefscustom.i`

For examples of the contents of custom class files, see Appendix A, “ADM Standard and Custom Class Files.” For information on using custom class files to extend the ADM, see Chapter 8, “Developing ADM Extensions.”

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
</table>
| Custom super procedure file   | `typecustom.p` | - Contains functions and internal procedures that define additional behavior for the class  
                                  |                 | - References the property file                                              |
|                               |                | - Is run only once for the entire session                                   |
| Custom prototype file         | `typeprotocustom.i` | - References all internal entries (functions, internal procedures) of the super procedure for the class |
| Custom exclude definition file| `typeexclcustom.i` | - Defines EXCLUDE-type preprocessor variables that exclude functions or internal procedures from the super procedure for the class |
| Custom instance definition file| `typedefscustom.i` | - References definitions for instance properties  
                                  |                 | - References the definition of the instance property dialog box             |
SmartObjects

The ADM is grounded in the idea that you can create external procedure objects, called SmartObjects, that represent standardized components with useful behavior. A SmartObject is defined by a procedure file called a SmartObject master, which is external to the application that instantiates the SmartObject.

SmartObjects are smart because they provide the essential functionality required for building an application. A SmartObject application component is encapsulated: it contains all the information relevant to itself and all the actions it can perform. It gets its visual features, user input capabilities, and database access abilities by encapsulating the functionality of basic objects and 4GL constructs in its procedure files within the communication framework of the ADM. Note the difference between using a basic object and a SmartObject in an application:

- When an application uses a basic object directly, the 4GL statements that define that object are written into the application’s procedure file.
- When an application uses a SmartObject that encapsulates a basic object, the 4GL statements that define the basic object are written into the SmartObject’s procedure file.

The ADM achieves encapsulation by providing a standard architecture (described in the “ADM classes” section on page 1–4) for building reusable and maintainable objects, and by providing a standard communication interface (see the “SmartLinks” section on page 1–12) that allows these objects to interoperate.

The external procedure file that defines a particular SmartObject—the SmartObject master—begins as a copy of one of several SmartObject templates that the ADM provides. With certain exceptions, each SmartObject type corresponds to a different SmartObject template. Each SmartObject also has one or more standard include files (see the “ADM classes” section on page 1–4) that define its properties and other aspects of its functionality. In addition, each SmartObject has one or more super procedures, which contain the internal procedures and functions that provide the SmartObject with its smart behavior. Any entity that interacts with a SmartObject—another object in the application or a user running the application—does so by telling the SmartObject to perform an action of some type.

SmartObject types

The ADM provides the SmartObjects described in Table 1–3. For detailed descriptions, see Chapter 2, “SmartObjects.”

<table>
<thead>
<tr>
<th>SmartObject type</th>
<th>Description</th>
<th>ADM class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmartContainer:</td>
<td>Holds instances of other SmartObjects, including other SmartContainers.</td>
<td>container</td>
</tr>
<tr>
<td>SmartDialog, SmartFrame, SmartWindow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmartDataBrowser</td>
<td>Browses records and allows updates to records in a browse widget; includes static and dynamic versions.</td>
<td>browser</td>
</tr>
<tr>
<td>SmartDataField</td>
<td>Displays a single data field in a SmartDataViewer.</td>
<td>field</td>
</tr>
</tbody>
</table>
Table 1–3: SmartObject summary

<table>
<thead>
<tr>
<th>SmartObject type</th>
<th>Description</th>
<th>ADM class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmartDataObject</td>
<td>Defines a set of records to fetch from a database and the update logic for those records.</td>
<td>data</td>
</tr>
<tr>
<td>SmartDataViewer</td>
<td>Displays data fields in a frame and coordinates their display with other SmartObjects. Can serve as a highly specialized type of SmartContainer.</td>
<td>viewer</td>
</tr>
<tr>
<td>SmartFilter</td>
<td>Displays data fields in a frame, to allow filtering of a data set by user-specified selection criteria.</td>
<td>filter</td>
</tr>
<tr>
<td>SmartFolder</td>
<td>Provides an interface (tab folder and paging mechanism) for selecting pages in a SmartContainer.</td>
<td>folder</td>
</tr>
<tr>
<td>SmartPanel</td>
<td>Groups functionally related buttons (for example, navigation or update buttons) on a frame.</td>
<td>panel</td>
</tr>
<tr>
<td>SmartSelect</td>
<td>A SmartDataField that performs a “lookup” on a single data field.</td>
<td>select</td>
</tr>
<tr>
<td>SmartToolbar</td>
<td>Provides a toolbar and/or menu interface to application commands.</td>
<td>toolbar</td>
</tr>
<tr>
<td>Simple SmartObject</td>
<td>Provides a template for creating new visual SmartObjects.</td>
<td>smart</td>
</tr>
</tbody>
</table>
The lifecycle of a SmartObject

The lifecycle of a SmartObject begins with its creation and ends when a user instantiates it in an application screen at run time. Figure 1–2 depicts the lifecycle of a typical SmartObject. This is a general picture; the lifecycle of specific SmartObjects might differ.

Figure 1–2: SmartObject lifecycle

SmartObject templates and masters

You can designate a SmartObject procedure as a persistent procedure; that is, a procedure that is run persistently. You do this from the AppBuilder, in the Procedure Settings dialog box for that procedure. For more information about persistent procedures and the WAIT-FOR statement, see OpenEdge Getting Started: ABL Essentials and OpenEdge Development: ABL Reference.

Before you can run persistent instances of a SmartObject, however, you must build its external procedure (its SmartObject master). You can build an individual SmartObject master from a template, which the AppBuilder reads into memory as an untitled procedure. (You can either use one of the templates provided with the AppBuilder or build your own templates.) Alternatively, you can create a new master by saving an existing file as another file while in the AppBuilder (choose File→ Save As on the AppBuilder menu bar).
Recall that conceptually, a SmartObject template is a procedure file that serves as a general starting point for building one or more specific SmartObjects masters. A template usually has minimal user-interface features and very little code (other than included code); therefore, it is not meant to be run directly. Adding too much data and too many user-interface features to a template might hinder its general usefulness and maintainability; however, some templates have associated data and user-interface elements, as appropriate. For example, the template for a SmartDialog has OK, Cancel, and Help buttons, user-interface elements that are standard to dialog boxes.

A SmartObject template generally includes a single support include file: the primary include file for the class on which the SmartObject is based. This include file has the same name as the template but the filename suffix .i instead of .w. For instance, the SmartDataViewer template viewer.w includes the primary include file viewer.i. A primary include file such as viewer.i does not directly contain most of the code needed to support the SmartObject. It contains only those program elements required to compile a master built from that template; all other support code for an object type resides in a super procedure (see the “Super procedures” section on page 1–14.)

**Instantiation**

At run time, the top-level application procedure (a SmartContainer; typically, a SmartWindow) instantiates the other SmartObjects it contains. Instantiation has several steps, during which the top-level SmartContainer does the following:

1. Runs the master for each SmartObject persistently to create an instance of that SmartObject.
2. Sets instance properties in each SmartObject and sets the position of each SmartObject relative to the SmartContainer.
3. Tells the SmartObject how it relates to the other objects in the SmartContainer.
4. Instructs the SmartObject to initialize and display itself. This is done in the generated procedure adm-create-objects.

The SmartContainer does not directly dictate the contents of SmartObjects when it tells them to initialize themselves. Each SmartObject responds differently to this request according to its type. The SmartContainer does not need to be aware of this; it sends exactly the same request to all SmartObjects and allows them to handle the details themselves.
SmartLinks

An important feature of the ADM is the messaging system it provides: you use objects called SmartLinks to create communication paths between pairs of SmartObjects. The ADM includes the following SmartLink types:

- **Container** — Links a containing object to an object that it contains.
- **Navigation** — Links an object that provides a navigation interface to an object that supplies a query through which to navigate.
- **TableIO** — Links an object that provides a record-modification interface to an object through which to enter changes.
- **Page** — Links an object that provides a page-selection interface to an object that manages the hiding and viewing of pages.
- **PageN** — Links an object that manages the hiding and viewing of a numbered set of pages to the SmartObjects on a specific page.
- **GroupAssign** — Links an object that manages a record to an object that manages a related record or another view of the same record.
- **Data** — Links an object that finds records to a visualization object or to another object that manages records.
- **Update** — Links a visualization object to a SmartDataObject for the purpose of submitting modifications to the data.
- **Commit** — Links a Commit SmartPanel to a SmartDataObject (for committing multiple updates in a single transaction).
- **Filter** — Links a SmartFilter to an object that directly or indirectly supplies a query to filter (SmartDataObject or SmartDataBrowser).

When you connect objects with a SmartLink, the SmartObject from which the link originates is the source object, and the object to which it connects is the target object.

The ADM implements SmartLinks using the Progress 4GL general PUBLISH/SUBSCRIBE event mechanism. For basic information on PUBLISH/SUBSCRIBE, see *OpenEdge Development: ABL Reference*. For more information on linking in the ADM and detailed descriptions of specific SmartLinks, see Chapter 3, “SmartLinks.”
Properties

The property file, an include file, defines the basic properties of a SmartObject and starts its super procedure, if necessary. Properties are stored in a temp-table defined for each SmartObject and are accessed using get and set property functions defined for each property (one get/set pair for each property). In addition, you can define additional properties for a SmartObject based on your application needs. For more information on properties and property files, see Chapter 5, “SmartObject Interactions.”
Super procedures

In the ADM methodology, most of the support code for an object type resides in a *super procedure*: a separately compiled procedure built as a structured .p file. For instance, the super procedure for SmartDataViewers is *viewer.p*. A super procedure can include both object-specific routines and object-specific versions of standard routines defined further up the ADM class hierarchy. (See Figure 1–1.)

Super procedures are a Progress 4GL concept rather than an ADM concept. The remainder of this section describes how the ADM uses Progress 4GL super procedures.

External procedures, internal procedures, and functions

An understanding of Progress 4GL procedure types is basic to an understanding of ADM super procedures. The ADM uses two Progress 4GL procedure types: external procedures and internal procedures. In addition, it uses 4GL functions. Internal procedures and functions are collectively referred to as *methods*:

*External procedures* are procedure files that typically have a .w or .p extension. External procedures can become *persistent procedures* at run time. A persistent procedure is a run-time instance of an external procedure that stays in memory until it is explicitly deleted. SmartObjects are implemented as run-time instances of external procedures.

*Internal procedures* are contained by external procedures. An internal procedure is a named entry point that can be executed by its containing external procedure or by other external procedures. Internal procedures represent actions that a SmartObject can perform.

*Functions* also are contained by external procedures. A function, like an internal procedure, can be executed by its containing external procedure or by other external procedures. Functions also represent actions a SmartObject can perform.
Figure 1–3 illustrates the relationship among external procedures, internal procedures, and functions.

For more information on procedure types, see OpenEdge Getting Started: ABL Essentials.
Super procedures in the ADM

In the ADM, a super procedure is a persistent procedure that contains versions of internal procedures or functions designed to be run remotely on behalf of SmartObjects. The use of super procedures provides a measure of inheritance and the ability to customize each internal procedure or function. Figure 1–4, which illustrates the inheritance process, shows how different versions of the same internal entry (in this example, the initializeObject procedure) exist in different super procedures.

Figure 1–4: Inheritance and the super procedure stack

The super procedures designed for SmartObjects are designed to be shared and stateless. This means that a given super procedure such as viewer.p is run only once in a session, no matter how many SmartDataViewers there are; each SmartDataViewer designates the same instance of the compiled code for viewer.p as its super procedure.

ADM super procedures use the Progress 4GL SOURCE-PROCEDURE and TARGET-PROCEDURE functions to provide information to their routines about who called them and for whom:

- The SOURCE-PROCEDURE function allows a routine in a super procedure to know who ran it, eliminating the need to pass TARGET-PROCEDURE as a handle to the routine.
- The TARGET-PROCEDURE function allows a routine in a super procedure to know on whose behalf it was invoked.

For more information on the SOURCE-PROCEDURE and TARGET-PROCEDURE functions, see OpenEdge Development: ABL Reference.
This chapter provides detailed descriptions of Progress SmartObjects.

The chapter contains the following sections:

- SmartObject summary
- SmartContainers
- SmartDataObjects
- SmartDataBrowsers
- SmartDataViewers
- SmartDataFields
- SmartPanels
- SmartFolders
- SmartFilters
- SmartToolbars
- SmartSelects
- Simple SmartObjects

The SmartObject summary describes general capabilities, brief descriptions, and information on include files and several special SmartObject-related procedures. The remainder of the chapter provides the following information, as applicable, for each SmartObject:

- Related template procedure files or master procedure files.
- Primary include files.
- ADM/Progress Advisor-supported SmartLinks.
- Property dialog box files.
- Main Block code section include files.
- Instance properties dialog box and its properties.
- Usage notes.
SmartObject summary

All SmartObjects have a set of common capabilities that allows them to participate smoothly in an application. They can:

- Initialize and destroy themselves.
- Get and set properties.
- Add and remove SmartLinks.
- Communicate using the ADM’s standard communication interface.

These capabilities provide a standard interface for creating and destroying SmartObject instances and for making SmartObjects interact. A procedure object that does not have these capabilities is not a SmartObject.

**Note:** When the AppBuilder reads procedure files, it looks for specific information to determine whether a procedure is a SmartObject. For details, see Chapter 6, “Advanced ADM Topics.”

The ADM provides the SmartObjects described in Table 2–1. Much of this information appears in Table 1–3 but it is repeated in this chapter for reference.

**Table 2–1: SmartObject types**

<table>
<thead>
<tr>
<th>SmartObject type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmartContainer:</td>
<td>Holds instances of other SmartObjects, including other SmartContainers. The SmartContainer types are:</td>
</tr>
<tr>
<td>SmartWindow, SmartFrame,</td>
<td>• SmartWindow — Provides a frame <strong>and</strong> a window for grouping widgets or SmartObjects</td>
</tr>
<tr>
<td>SmartDialog</td>
<td>• SmartFrame — Provides a frame <strong>but not</strong> a window for grouping widgets or SmartObjects</td>
</tr>
<tr>
<td></td>
<td>• SmartDialog — Displays a dialog box and three default buttons (<strong>OK, Cancel, Help</strong>); also used for grouping widgets or SmartObjects</td>
</tr>
<tr>
<td>SmartDataObject</td>
<td>Defines a set of records to fetch from a database and the update logic for those records.</td>
</tr>
<tr>
<td>SmartDataBrowser</td>
<td>Browses records and allows updates to records in a browse widget; includes dynamic and static versions.</td>
</tr>
<tr>
<td>SmartDataViewer</td>
<td>Displays data fields in a frame and coordinates their display with other SmartObjects. Although a SmartDataViewer can be a type of SmartContainer, it is a highly specialized type and so is discussed separately.</td>
</tr>
<tr>
<td>SmartDataField</td>
<td>Displays a single data field in a SmartDataViewer.</td>
</tr>
<tr>
<td>SmartPanel</td>
<td>Groups functionally related buttons (for example, navigation or update buttons) on a frame.</td>
</tr>
<tr>
<td>SmartFolder</td>
<td>Provides an interface (tab folder and paging mechanism) for selecting pages in a SmartContainer.</td>
</tr>
</tbody>
</table>
Accessing SmartObjects

SmartObjects are available either as an icon on the AppBuilder palette or from the selection list in the File→New dialog box. For detailed instructions on using the AppBuilder to incorporate SmartObjects into Progress applications and modify them at design time, see OpenEdge Development: AppBuilder.

SmartObject instance properties

Because SmartObjects are procedure instances, they have different properties than basic objects (widgets). The instance properties for each SmartObject type are accessible in its Properties dialog box in the AppBuilder. This manual describes the instance properties available from the instance properties dialog box for a given SmartObject type. Note that descriptions of the instance properties also are available in the online help.

This manual does not describe the mechanics of using instance properties dialog boxes. For instructions, see OpenEdge Development: AppBuilder. It does, however, describe how to modify existing SmartObject instance properties dialog boxes and write your own instance dialog boxes. For details, see Chapter 6, “Advanced ADM Topics.”

AppBuilder-maintained procedures

The ADM provides several procedures that are generated by the AppBuilder at design time for use by SmartObjects. These procedures are as follows:

- **adm-create-objects** — Used by SmartContainers to start instances of contained objects.
- **disable_UI** — Deletes the SmartObject procedure handle (run from destroyObject).
- **enable_UI** — Enables basic objects in a SmartContainer being built; that is, makes them sensitive to user input (run from enableObject).

The AppBuilder handles these procedures specially, generating the appropriate code for them as it is needed. You can view these procedures in the AppBuilder’s Section Editor but they are marked Read Only. For information on using the Section Editor, see OpenEdge Development: AppBuilder.

Table 2–1: SmartObject types

<table>
<thead>
<tr>
<th>SmartObject type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmartFilter</td>
<td>Displays data fields in a frame, to allow filtering of a data set by user-specified selection criteria.</td>
</tr>
<tr>
<td>SmartToolbar</td>
<td>Provides a toolbar and/or menu interface to application commands.</td>
</tr>
<tr>
<td>SmartSelect</td>
<td>A SmartDataField that performs a lookup on a single data field.</td>
</tr>
<tr>
<td>Simple SmartObject</td>
<td>Provides a template for creating new visual SmartObjects.</td>
</tr>
</tbody>
</table>
**SmartContainers**

SmartContainers are ADM container-class objects that are designed to hold instances of other SmartObjects. There are four types of SmartContainers: SmartDataViewers, SmartDialogs, SmartFrames, and SmartWindows, each of which provides a different type of container widget. A SmartContainer can hold any other SmartObjects, including other SmartContainers; however, SmartDataViewers are intended to contain very specific types of objects. See the “SmartDataViewers” section on page 2–21 for details.

When you build a Progress application, the step immediately after creating your SmartObject masters is to build an application screen. The first step in assembling this screen is to open a SmartContainer. You then insert instances of SmartObjects into the SmartContainer and link them together to produce your application screen.

SmartContainers have three important functions in a SmartObject application:

- Creating, initializing, and destroying their contents
- Managing paging
- Defining links between SmartObjects

The following sections describe these functions, as well as the SmartWindow, SmartFrame, and SmartDialog types of SmartContainers. Because the SmartDataViewer is a highly specialized type of SmartContainer, it is described separately in the “SmartDataViewers” section on page 2–21.

**SmartContainers and their contents**

All SmartObjects that a SmartContainer contains are linked to it by a Container SmartLink. (For more information, see Chapter 3, “SmartLinks.”) Figure 2–1 illustrates this relationship.

![Figure 2–1: SmartContainer and contained SmartObject](image)

The term SmartContainer suggests a visual relationship similar to that of a frame and the widgets inside that frame. This kind of relationship is typical between a SmartContainer and the object it contains. The container relationship extends beyond this, however, especially when a SmartWindow contains another SmartWindow. In this situation, although the two SmartWindows appear separately on your screen, the ADM parents one SmartWindow to the other. As a result, if you minimize or destroy the parent window, you also minimize or destroy the contained window.

The container relationship is, therefore, a logical relationship and not necessarily a visual relationship. Another example is a SmartContainer that contains a SmartDataObject. The SmartDataObject has no visualization, so at run time, you cannot see a visual relationship. Nevertheless, the SmartDataObject is contained in the SmartContainer.
At design time, when you insert SmartObject instances into a SmartContainer, the AppBuilder generates an `adm-create-objects` procedure in the SmartContainer. This procedure runs persistent instances of the SmartObjects contained in the SmartContainer and performs other initializations for those SmartObjects, such as setting their relative position, parenting them correctly to the SmartContainer, and adding links.

**Paging**

All SmartContainers can manage application pages: logical groupings of SmartObjects. When you select a given page, the ADM automatically hides all other pages (except page 0). Pages provide an easy-to-use mechanism for viewing and hiding objects, and help conserve screen space. For more information, see *OpenEdge Development: AppBuilder*.

**Pass-through links**

A SmartObject inside a SmartContainer cannot link directly to a SmartObject external to the SmartContainer; however, you can create a link called a pass-through link that allows the two objects to communicate. See the “Pass-through links” section on page 3–13.

**SmartWindows**

A SmartWindow is a SmartContainer that has a default frame and a window. A SmartWindow can be the main window of an application or a subwindow. It is an ADM container-class object.

Table 2–2 lists the SmartWindow files.

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template file</td>
<td><code>src/adm2/template/cntnrwin.w</code></td>
</tr>
<tr>
<td>Primary include file</td>
<td><code>src/adm2/containr.i</code></td>
</tr>
<tr>
<td>Main block include file</td>
<td><code>src/adm2/windowmn.i</code></td>
</tr>
</tbody>
</table>

**SmartFrames**

A SmartFrame is a SmartContainer that has a default frame but not a window. A SmartFrame groups logically related widgets (or SmartObjects) onto a frame that the AppBuilder can visualize in other SmartContainers. You also can use it to group SmartObjects into a single object that can be used as a logical unit. It is an ADM container-class object.
Table 2–3 lists the SmartFrame files.

**Table 2–3: SmartFrame files**

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template file</td>
<td>src/adm2/template/cntnrfrm.w</td>
</tr>
<tr>
<td>Primary include file</td>
<td>src/adm2/containr.i</td>
</tr>
</tbody>
</table>

**SmartDialogs**

A SmartDialog is a SmartContainer that has a default frame viewed as a dialog box and three default buttons: **OK**, **Cancel**, and **Help**. SmartDialogs are not designed to run persistently, be inserted in other SmartContainers, or have SmartLinks from other SmartObjects. Typically, you use a SmartDialog if you want to build a dialog box containing other SmartObjects or widgets. It is an ADM container-class object.

Table 2–4 lists the SmartDialog files.

**Table 2–4: SmartDialog files**

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template file</td>
<td>src/adm2/template/cntnrdlg.w</td>
</tr>
<tr>
<td>Primary include file</td>
<td>src/adm2/containr.i</td>
</tr>
<tr>
<td>Main block include file</td>
<td>src/adm2/dialogmn.i</td>
</tr>
</tbody>
</table>
SmartDataObjects

A SmartDataObject is an ADM data-class SmartObject that defines a set of records to fetch from a database. SmartDataObjects coordinate with other SmartObjects such as SmartDataViewers and SmartDataBrowsers to manage records in an application. Also, SmartDataObjects respond to navigation controls such as those available in SmartPanels and SmartToolbars.

The SmartDataObject is a nonvisual object; it has no run-time visualization and needs a visualization or user-interface (UI) object to display data. The visualization can exist in many forms, both Progress and non-Progress, and is defined independently of the SmartDataObject itself.

Table 2–5 lists the SmartDataObject files.

Table 2–5: SmartDataObject files

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename / SmartLinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template file</td>
<td>src/adm2/template/data.w (standard SmartDataObject template with wizard)</td>
</tr>
<tr>
<td>Primary include file</td>
<td>src/adm2/data.i</td>
</tr>
<tr>
<td>ADM/Progress Advisor-supported SmartLinks</td>
<td>Commit-Target Data-Source Data-Target Filter-Target Navigation-Target Update-Target</td>
</tr>
<tr>
<td>Properties dialog box</td>
<td>src/adm2/support/datad.w (source code) gui/adm2/support/datad.r (compiled code)</td>
</tr>
</tbody>
</table>

SmartDataObject instance properties

Figure 2–2 shows the SmartDataObject Properties dialog box.

Figure 2–2: SmartDataObject Properties dialog box
The properties in this dialog box are used as follows:

- **Partition** — This is the name of a *partition*: a logical grouping of one or more Progress procedures; for example, a set of procedures for managing inventory. The Partition Deployment Tool maps a partition to an Application Service, which defines where the SmartDataObject runs (which could be on an AppServer™).

- **Rows To Batch** — This is the number of rows to retrieve at a time from the data source and be stored by the client. The AppBuilder assigns the specified value to the `RowsToBatch` property. The default is 200.

- **Check Current Changed** — If this is checked, Progress checks on update whether records in the data source were changed since they were retrieved by this instance of the SmartDataObject. The AppBuilder sets the value for the `CheckCurrentChanged` property to YES if this toggle box is checked; otherwise, NO. The default is YES.

- **Rebuild Dataset on Reposition** — If this is checked, Progress rebuilds the temp-table on reposition starting with the specified record. If unchecked, the temp-table is rebuilt starting from the beginning of the data set. The AppBuilder sets the value for the `RebuildOnRepos` property to YES if this toggle box is checked; otherwise, NO. The default is NO.

- **Force to Stateful Operating Mode** — If this is checked, the SmartDataObject runs in state-aware mode regardless of the AppServer’s operating mode. If unchecked, the SmartDataObject inherits its operating mode from the AppServer. The default is NO.

- **Destroy on each stateless Web Request** — If checked, this destroys the SmartDataObject between requests. The default (unchecked) behavior is to keep the SmartDataObject alive between requests. Checking this property disables the Disconnect AppServer on each Web Request property.

  **Note:** This property is disabled when the SmartDataObject Properties dialog box is used for non-Web SmartDataObjects.

- **Disconnect AppServer on each Web Request** — If checked, this specifies that the AppServer part of the SmartDataObject is destroyed on each request but the client part remains alive. This allows you to use the Force to Stateful Operating Mode property without locking the AppServer for the lifetime of the SmartDataObject.

  **Note:** This property is disabled when Destroy on each stateless Web Request is checked. It also is disabled when the SmartDataObject Properties dialog box is used for non-Web SmartDataObjects.
SmartDataObject usage notes

This section provides specific usage notes on several SmartDataObject issues; however, because SmartDataObjects are the heart of any Progress ADM application, all remaining chapters of this manual (except Chapter 3, “SmartLinks”) also provide additional information on using them. In particular, see the following sections in Chapter 7, “Developing Your Application’s Business Logic”:

- The “SmartDataObject query and update operations” section on page 7–3 (RowObject and RowObjUpd temp-tables).
- The “Managing SmartDataObjects in distributed mode” section on page 7–23 (Partition property).
- The “AppServer-related SmartDataObject properties” section on page 7–27 (AppServer operating modes and the Force to Stateful Operating Mode property).
- The “Running SmartDataObjects in a distributed environment” section on page 7–19 (the client/AppServer configuration for SmartDataObjects).

Setting the number of database rows to batch

The result set retrieved by a SmartDataObject’s database query is transferred into a Progress temp-table called RowObject that consists of the columns defined for the SmartDataObject. For a description of the RowObject temp-table, see the “SmartDataObject query and update operations” section on page 7–3.

If the query defines a large data set, it might take Progress a considerable amount of time to load all the database records into this temp-table for transmission to the client. For this reason, Progress transfers rows to the RowObject table a batch at a time, rather than one at a time. The default number of rows to batch, stored in the RowsToBatch property, is 200. In many cases, this is a reasonable number; however, you might want to adjust the value of this property up or down, depending on your expectations about the database query and how it will be used. You do this by specifying a value for the Rows To Batch field in the SmartDataObject’s instance properties dialog box.

Suppose, for example, you expect your SmartDataObject’s query to yield a data set that is slightly larger than 200 rows. In this case, you might want to set Rows To Batch high enough to retrieve the entire data set when the query first opens. This ensures there are no further delays if it is necessary to fetch more rows later. Conversely, if the startup time for your application screen is very sensitive, you might be able to improve startup performance noticeably by setting Rows To Batch to a very small number, to display the first batch of rows to the user as fast as possible.

Once you set this value at design time, you are not locked into that value for the entire time the application is run. You might choose to set Rows To Batch to an initial value at design time, then modify it programmatically at run time, based on the particular query being executed or on other factors.

As a general rule, you should limit the size of the data set the application code retrieves. Allowing users to browse thousands of records to make a selection is often an inefficient and ineffective way to present data, especially if the records must all be passed across an AppServer connection. Using a SmartFilter or SmartSelect SmartObject or other technique to refine the retrieval to the actual records needed can greatly increase the efficiency of an application and essentially eliminate issues with the size of the data set.
Checking database records on update attempts

By default, when an application user tries to update a SmartDataObject row, Progress checks whether the database record from which the row was derived was modified since the record was read into the RowObject temp-table. If so, Progress rejects the update attempt and returns to the client the values that result from the update that occurred since the record was read (by another user) along with an error message, so the user can make and save the later updates.

**Note:** This check does not include a field-by-field comparison of fields modified by a later user with fields modified by an earlier user. Unless the check is disabled, any change to any field in the record will cause the rejection of a later update.

In general, this check for other updates is valuable because it prevents making conflicting changes to a record; specifically, having a later set of changes overwrite an earlier set. In some cases, however, it might not matter whether a record was changed since it was read into the RowObject temp-table. You can set the CheckCurrentChanged property to instruct Progress not to check for earlier updates. Suppose, for example, your database table contains information such as address data, and it does not matter whether one user changes a field value in this table after another user changes a field value (either the same field or another one in the same table). In this case, the check is not necessary, and you can disable it by unchecking the Check Current Changed toggle box in the SmartDataObject’s instance property dialog box.

Changing how Progress reloads the RowObject temp-table

By default, Progress loads the RowObject temp-table from beginning to end. If a SmartDataObject’s query defines a large data set, a fetchLast operation or other reposition to a place far down the data set can result in a significant wait while Progress loads this temp-table. This is because Progress might need to load many batches of rows before it reaches the desired row.

To eliminate this delay, use the SmartDataObject’s RebuildOnRepos property, accessible through the Rebuild Dataset on Reposition toggle box in the instance properties dialog box. If you check this toggle box, then whenever a reposition moves to a record that is not already in the RowObject table, Progress performs the reposition in the database query directly and rebuilds the RowObject temp-table from that point.

For example, if the RowsToBatch property is set to 200 for a particular SmartDataObject, the first 200 rows in the database query data set are transferred into the RowObject temp-table when the SmartDataObject’s query is opened. Now, suppose an application user presses the Last button in an associated SmartPanel or SmartToolbar. Progress repositions the database query directly to the last row, and the RowObject table is rebuilt to contain the last 200 rows in the query. (Note that repositioning to the last row is a very fast operation if the INDEXED-REPOSITION query attribute is set to on and the query can use an index to locate the last record.) Progress extends the RowObject temp-table in both directions as the user scrolls forward or backward.

In general, if a SmartDataObject’s query returns a large data set, you should turn the RebuildOnRepos property to on. If the likely number of rows in the data set is not much larger than the setting for RowsToBatch, however, it is better to leave this property off and get the entire data set.
The Define Temp-Tables option

Typically, a SmartDataObject directly accesses the database tables for which it is defined. However, the SmartDataObject wizard provides an option called Define Temp-Tables that you can use at design time to define your own data source. With this option, you define Progress temp-tables for the database tables, which the SmartDataObject accesses instead of the database; each of these temp-tables can contain a subset of the fields in the corresponding database table as well as summary and calculated fields.

Note: The temp-tables that this option creates have no connection to the RowObject and RowObjUpd temp-tables that a SmartDataObject uses to manage database data. Those temp-tables are created and maintained automatically.

To define temp-tables for a SmartDataObject, in the AppBuilder:

1. Select the SmartDataObject icon from the AppBuilder palette and specify New. The SmartDataObject wizard appears.
2. Click Next to go to Page 2 of the SmartDataObject wizard.
3. Click Define Temp-Tables. The Temp-Table Maintenance dialog box appears.
4. Specify the database tables you want the SmartDataObject to access, using the Add and Remove buttons as necessary. The AppBuilder creates temp-tables for the tables you specify.
5. Click OK to return to the SmartDataObject wizard.
6. Click Define Query. The Query Builder appears.
7. In the Query Builder’s Database field, select Temp-Tables from the drop-down list, then proceed as usual to build the SmartDataObject’s query.

Note: The compiler does not support the VALIDATE option on a temp-table field that is defined to be LIKE another temp-table field.

8. Click OK to return to the SmartDataObject wizard.
9. Click Next to go to Page 3.
10. Click Add fields. The Column Editor appears.
11. Specify the fields to be included in the SmartDataObject, then click OK to return to the SmartDataObject wizard.
12. Click Next to finish creating the SmartDataObject.
You must be connected to a database with the name of the temp-table you create, to use it. When the SmartDataObject executes the query, it accesses these temp-tables rather than accessing the database tables directly.

**Note:** Progress neither populates the tables this option creates nor manages any other of their interactions with the database, so you must write the code to perform these activities. If you do not, you will receive a warning message when you run the application, and any visual objects connected to the SmartDataObject will not display data in their fields.
SmartDataBrowsers

A SmartDataBrowser is an ADM browser-class SmartObject that displays records and allows updates to records in a browse widget. In addition, it coordinates the display and updating of records with other SmartObjects, typically SmartDataViewers and SmartDataObjects. A SmartDataBrowser uses a SmartDataObject as a data source.

Table 2–6 lists the SmartDataBrowser files.

### Table 2–6: SmartDataBrowser files

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename / SmartLinks</th>
</tr>
</thead>
</table>
| Template file and master file (Because dynamic SmartDataBrowsers are customized at design time or even at run time, a template is not necessary.) | src/adm2/dynbrowser.w (master for dynamic SmartDataBrowser; no wizard)  
src/adm2/template/browser.w (template for static SmartDataBrowser; includes wizard) |
| Primary include file | src/adm2/browser.i |
| ADM/Progress Advisor-supported SmartLinks | Data-Source  
Data-Target  
Navigation-Target  
TableIO-Target |
| Properties dialog box | src/adm2/support/dynbrowserd.w (source code for dynamic SmartDataBrowser)  
gui/adm2/support/dynbrowserd.r (compiled code for dynamic SmartDataBrowser)  
src/adm2/support/visuald.w (source code for static SmartDataBrowser)  
gui/adm2/support/visuald.r (compiled code for static SmartDataBrowser) |

**SmartDataBrowser instance properties**

As the first row in Table 2–6 indicates, there are two types of SmartDataBrowsers: dynamic and static. Each has a different set of instance properties and, therefore, different instance properties dialog box.
Dynamic SmartDataBrowser instance properties

Figure 2–3 shows the Dynamic SmartDataBrowser Properties dialog box.

![Dynamic SmartDataBrowser Properties dialog box](image)

You use the instance properties in this dialog box as follows:

- **SmartDataObject** — This field enables you to choose the SmartDataObject to drive the dynamic SmartDataBrowser.

- **Edit Displayed field list** — This opens a list box that lists the fields in the SmartDataObject that is the SmartDataBrowser’s Data-Source, from which you can select the fields to be displayed in the dynamic SmartDataBrowser.

- **Edit Enabled field list** — This opens a list box that lists the fields in the SmartDataObject that is the SmartDataBrowser’s Data-Source, from which you can select the fields to be enabled for update.

- **Search Field** — If a field is specified, the first line of the SmartDataBrowser’s frame is allocated to a fill-in field in which the application user can enter a value for a search field. The SmartDataObject query is re-sorted automatically by the search field, and on each keystroke entered, the browse is repositioned to the first row in which the specified field has a value greater than or equal to what was typed so far.

- **Enable** — If checked (the default), this enables the dynamic SmartDataBrowser for use.

- **Layout** — This property is disabled for dynamic SmartDataBrowsers.

- **View** — If checked (the default), this makes the dynamic SmartDataBrowser visible.
• **Down** — This determines the number of rows to display in the dynamic SmartDataBrowser.
  
  – If this is 0 (zero, the default), Progress calculates the number of rows displayed in the dynamic SmartDataBrowser based on the size of the browser. This might result in the display of a partial row at the bottom of the browse. (You can adjust the height of the browse manually to compensate.)
  
  – If this is a positive integer, Progress displays exactly this many rows in the browse. You cannot specify a decimal value, so the browse never displays a partial row.

  The value of this property is assigned to the browse’s `Down` attribute.

• **Calculate Width** — This determines the width of the dynamic SmartDataBrowser.
  
  – If this is unchecked (the default), the browse’s EXPANDABLE property is turned on, and Progress sets the width of the browser based on the width of the dynamic SmartDataBrowser. In this case, if the width of the browse columns is less than the width of the browse, the last column is expanded to fit the width of the browser.
  
  – If this is checked, Progress calculates the exact width of the dynamic SmartDataBrowser’s columns, compares the result with the `Max Width` value, and uses the smaller of the two values as the width of the browse. Checking this box turns off the browse’s EXPANDABLE attribute.

• **Max Width** — This specifies the maximum width of the browse (the default is 80.00). If you check `Calculate Width` and the calculated width is greater than `Max Width`, `Max Width` is used instead. This field is enabled only when `Calculate Width` is checked.

• **Scroll Remote Results List** — If checked, this causes the SmartDataBrowser to try to fill the browse with data when you scroll the browser to the end of a batch of records.

• **Fetch Data to Fill Browse on Reposition to End of Batch** — When checked, this is similar to **Scroll Remote Results List**, except it occurs on a reposition of the browse (as opposed to a scrolling of the browse).

**Static SmartDataBrowser instance properties**

The static SmartDataBrowser uses the **Visual SmartObject Properties** dialog box, shown in Figure 2–4.

![SmartDataBrowser Properties dialog box](image_url)
You use the instance properties in this dialog box as follows:

- **Enable** — If checked (the default), this enables the static SmartDataBrowser for use.

  **Note:** Do not uncheck (turn off) this property.

- **Layout** — This property is enabled for static SmartDataBrowsers when a custom layout is defined in the SmartDataBrowser master. This box is disabled if there are no custom layouts defined. Its value is the same as the browser's parent layout by default.

- **View** — If checked (the default), this makes the static SmartDataBrowser visible.

- **Scroll Remote Results List** — If checked, this causes the SmartDataBrowser to try to fill the browse with data when you scroll the browser to the end of a batch of records.

- **Fetch Data to Fill Browse on Reposition to End of Batch** — When checked, this is similar to **Scroll Remote Results List**, except it occurs on a reposition of the browse (as opposed to a scrolling of the browse).

---

**Dynamic versus static SmartDataBrowsers**

The primary difference between a dynamic and a static SmartDataBrowser is that a given dynamic SmartDataBrowser can be configured to display and update the query of any SmartDataObject, while a static SmartDataBrowser is specific to a particular SmartDataObject: The two types of SmartDataBrowsers are described in more detail below:

- **The dynamic** SmartDataBrowser displays and updates records from a set of fields that are determined at run time. At run time, the ADM assigns the display and update fields based on the current values of the SmartDataBrowser’s `DisplayedFields` and `EnabledFields` instance properties and takes the query from the SmartDataObject to which the SmartDataBrowser is linked. This is why you can configure a dynamic SmartDataBrowser to display and update the query of any SmartDataObject.

  You can set the `DisplayedFields` and `EnabledFields` properties of the dynamic SmartDataBrowser at design time through the **Dynamic SmartDataBrowser Properties** dialog box. If `DisplayedFields` and `EnabledFields` are unset at run time, their values are taken from the associated SmartDataObject’s `DataColumns` and `UpdatableColumns` properties. For details, see the “Setting the dynamic SmartDataBrowser field properties” section on page 2–20.

  Because a dynamic SmartDataBrowser instance’s display and update fields are not assigned until run time, it appears at design time as a browse widget with no columns.

- **The static** SmartDataBrowser displays and updates records from a set of fields that are assigned at design time. At design time, the wizard sets the display and update fields, as well as the data source’s query, based on the SmartDataObject to which the SmartDataBrowser is linked. This is why a given static SmartDataBrowser is specific to a particular SmartDataObject.

  Because a static SmartDataBrowser instance’s display and update fields are assigned at design time, it appears at design time as a browse widget with one column for each field selected in the wizard.
Another significant difference is that you create masters for static SmartDataBrowsers but not for dynamic SmartDataBrowsers. This is because a dynamic SmartDataBrowser uses the SmartDataObject to which it is linked at run time and, therefore, it does not need a master.

SmartDataBrowser usage notes

This section discusses special programming considerations for using SmartDataBrowsers.

The NO-ASSIGN property

The NO-ASSIGN property in the design window of a SmartDataBrowser master or the property sheet for the browse widget is set to TRUE. This allows the ADM code to handle all record updates.

**Caution:** Do not set NO-ASSIGN to FALSE, or the SmartDataBrowser will not be able to use the ADM methods to perform record updates.

Default triggers for SmartDataBrowsers

The master procedure file for a SmartDataBrowser defines a variety of triggers for the browse widget that it contains:

- **OFF-END** ({src/adm2/brsoffnd.i}) — Checks whether there are more rows in the query in the forward direction
- **OFF-HOME** ({src/adm2/brsoffhm.i}) — Checks whether there are more rows in the query in the backward direction
- **END** ({src/adm2/brsend.i}) — Performs a fetchLast operation
- **HOME** ({src/adm2/brshome.i}) — Performs a fetchFirst operation
- **CTRL-END** — Applies the END trigger, which performs a fetchLast operation, to the browse (does not have an include file of its own)
- **CTRL-HOME** ({src/adm2/brshome.i}) — Applies the HOME trigger, which performs a fetchFirst operation, to the browse (does not have an include file of its own)
- **ROW-ENTRY** ({src/adm2/brsentry.i}) — Displays the initial values for newly added or copied rows
- **ROW-LEAVE** ({src/adm2/brsleave.i}) — If the selected object is not a SmartPanel button (for example, a **Cancel** or **Reset** button), saves any changes made to the row, otherwise the button takes the action associated with the button
- **VALUE-CHANGED** ({src/adm2/brschng.e.i}) — Instructs the SmartDataObject linked to the browse to let other objects know that the record has changed
- **SCROLL-NOTIFY** ({src/adm2/brsscr01.i}) — Checks whether there are more rows in the query in the forward direction
Scrolling past the end of the result set

The SmartDataBrowser’s OFF-END and OFF-HOME triggers produce certain visual anomalies when your users browse the associated SmartDataObject query’s result set and try to scroll past the end of the result set.

When a SmartDataBrowser browses a SmartDataObject query, the associated RowObject temp-table might contain only a subset of the entire database query result set at any given time, so the SmartDataBrowser might browse only that subset. Although the supporting trigger code for the SmartDataBrowser is designed to make this as transparent as possible, the retrieval of multiple batches of rows into the SmartDataObject’s RowObject temp-table has some visible impact on the SmartDataBrowser.

In particular, the OFF-END trigger fires when the browse is scrolled to the bottom of the current result set, which happens when the application user does one of the following:

• Presses the DOWN arrow on the keyboard in an attempt to scroll beyond the last row in the current result set
• Presses the down arrow on the SmartDataBrowser’s vertical scroll bar until the last row is reached, then releases the arrow

When the OFF-END event occurs, the SmartDataBrowser code asks the SmartDataObject whether there are more rows to retrieve from the database. If so, an additional batch of rows is added to the RowObject temp-table, the temp-table’s query is reopened, and the browse is repositioned to the same row as before the event. This causes a delay (normally a brief one) while the records are retrieved and the query is reopened. It also means that the user must release the scroll bar arrow to see more rows.

If the SmartDataObject’s Rebu1d0nRepos instance property is set to YES, an attempt to scroll past the end of the result set might occur in either a forward or a backward direction. For example, if Rebu1d0nRepos is YES, and the application user either presses the Last button on a SmartPanel or SmartToolbar or presses CTRL-END on the keyboard to move to the last row in the result set, the RowObject table in the SmartDataObject is rebuilt from the end. Thus, if the user scrolls backwards (upwards) to the first row in the current result set, the OFF-HOME event causes the retrieval of the next batch of rows before the current result set, which causes a similar delay as the user scrolls up through the data. The vertical scroll bar might be similarly inconsistent and not reflect the size of the complete data set.

These visual anomalies are necessary to allow a client without a database connection to browse a potentially large result set without the entire data set being moved to the client at one time. If you do not want them to occur in your application, do one of the following:

• Set the SmartDataObject’s RowsToBatch instance property to retrieve the entire data set at one time.
• Set its Rebu1d0nRepos instance property to build the result set in a single direction from beginning to end.

Multiple selection

Multiple selection is not supported for SmartDataBrowsers. If you want to use this option, you must add supporting code to communicate the multiple selection to other SmartObjects as needed.
**Setting the dynamic SmartDataBrowser field properties**

Once you link a dynamic SmartDataBrowser to a SmartDataObject, you can set the SmartDataBrowser’s `DisplayedFields` and `EnabledFields` instance properties in its instance properties dialog box. The ADM uses the values at run time to determine the display and update fields for the SmartDataBrowser. Setting only one or neither property has these results:

- If you set `DisplayedFields` but not `EnabledFields`, and the SmartDataBrowser has a TableIO-Source (an Update SmartPanel), `EnabledFields` is set at run time to a list of all specified `DisplayedFields` in the associated SmartDataObject’s `UpdatableColumns` property. Otherwise, it is made blank (that is, there are no `EnabledFields`).

- If you set `EnabledFields` but not `DisplayedFields`, and the SmartDataBrowser has a TableIO-Source (an Update SmartPanel), `DisplayedFields` is set at run time to a list of all fields in the associated SmartDataObject’s `DataColumns` property.

- If you leave both properties unset, the `UpdatableColumns` property is set only if there is a TableIO-Source.

**The dynamic SmartDataBrowser SearchField property**

The dynamic SmartDataBrowser has a property called `SearchField` that is not available in the static SmartDataBrowser. This property allows you to specify a field on which to search at run time. If you specify a value, the first line of the SmartDataBrowser frame is allocated to a fill-in field in which the user can enter a search value at run time. The SmartDataObject query is automatically re-sorted by the `SearchField` value, and on each keystroke entered, the browse is repositioned to the first row whose search field has a value greater than or equal to the user-specified search value.

**Displaying row markers in the dynamic SmartDataBrowser**

Unchecking the `NO-ROW-MARKERS` property in the property sheet for the dynamic SmartDataBrowser does not display row markers (the expected behavior of this browser). You can get row markers, however, by explicitly setting `ROW-MARKERS` to `YES` in `dynbrowser.w`. Specifically, add the following code to the main block of `dynbrowser.w`:

```plaintext
ASSIGN BROWSE {&BROWSE-NAME}:ROW-MARKERS = YES.
```
SmartDataViewers

A SmartDataViewer is an ADM viewer-class SmartObject that displays database fields in a frame. A SmartDataViewer uses a SmartDataObject as a data source. A SmartDataViewer can be a type of SmartContainer; however, because it is a highly specialized SmartContainer, it is described separately in this section. For a general description of SmartContainers, see the “SmartContainers” section on page 2–5.

In addition to displaying database fields in a frame, a SmartDataViewer coordinates its display of records with other SmartObjects, typically SmartDataBrowsers and SmartDataObjects. SmartDataViewers can also coordinate with SmartPanels to modify records. SmartDataViewers typically use SmartDataObjects as data sources. For more information, see Chapter 4, “Data Management in the ADM.”

A SmartDataViewer is intended to contain only basic objects, SmartDataObjects that will be used with SmartSelects, and/or SmartDataFields. (See the “SmartSelects” section on page 2–43 and the “SmartDataFields” section on page 2–23.) You typically use contained SmartDataFields to provide specially formatted displays of data in selected fields in the SmartDataViewer. For example, you might develop a SmartDataField that appears as a calendar and use it in a SmartDataViewer to display a date field.

Table 2–7 lists the SmartDataViewer files.

Table 2–7:  SmartDataViewer files

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename / SmartLinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template file</td>
<td>src/adm2/template/viewer.w (standard SmartDataViewer template with wizard)</td>
</tr>
<tr>
<td>Primary include file</td>
<td>src/adm2/viewer.i</td>
</tr>
<tr>
<td>ADM/Progress Advisor-supported SmartLinks</td>
<td>Data-Target</td>
</tr>
<tr>
<td></td>
<td>GroupAssign-Source</td>
</tr>
<tr>
<td></td>
<td>GroupAssign-Target</td>
</tr>
<tr>
<td></td>
<td>TableIO-Target</td>
</tr>
<tr>
<td></td>
<td>Update-Source</td>
</tr>
<tr>
<td>Properties dialog box</td>
<td>src/adm2/support/visuald.w (source code)</td>
</tr>
<tr>
<td></td>
<td>gui/adm2/support/visuald.r (compiled code)</td>
</tr>
</tbody>
</table>

SmartDataViewer instance properties

The SmartDataViewer uses the Visual SmartObject Properties dialog box, shown in Figure 2–5.

![Figure 2–5: SmartDataViewer instance properties dialog box]
You use the instance properties in this dialog box as follows:

- **Enable** — If checked (the default), this enables the dynamic SmartDataViewer for use
- **View** — If checked (the default), this makes the dynamic SmartDataViewer visible
- **Layout** — This property is disabled for SmartDataViewers when an alternate layout has been defined in the SmartDataViewer master

### SmartDataViewer usage notes

This section discusses special programming considerations for using SmartDataViewers.

#### Adding SmartObjects to SmartDataViewers

You add SmartDataFields to a SmartDataViewer after you pick all of the fields that will appear in the viewer. For details on how to add SmartDataFields, see the “Adding SmartDataFields to SmartDataViewers” section on page 2–25.

The SmartDataViewer does not support communications with contained SmartObjects other than SmartDataFields. It is possible to drop other SmartObjects onto a SmartDataViewer—its adm-create-objects procedure will run and initialize them—but there is no built-in support for doing so.

#### Inheriting Data Dictionary validation expressions

The SmartDataViewer’s template is defined with the USE-DICT-EXPS property to allow the viewer’s fields to inherit Data Dictionary validation expressions and help messages. If this property is not enabled, the SmartDataViewer does not inherit Data Dictionary validation settings regardless of the Inherit Dictionary Validation Expressions column option of the viewer’s SmartDataObject.
A SmartDataField is an ADM field-class SmartObject that displays a single data field. It is intended for inclusion only in a SmartDataViewer. You typically use a SmartDataField to provide a specially formatted display of the data in a selected field in the SmartDataViewer. To do this, you replace the field on the SmartDataViewer with your SmartDataField. For example, you might develop a SmartDataField that appears as a calendar and use it in a SmartDataViewer to display a date field.

**Note:** The field to be replaced with a SmartDataField must be a RowObject field chosen from a SmartDataObject.

Because there is no standard representation for a SmartDataField, it does not have a wizard and opens initially as a frame. You, as the application developer, determine how to represent your SmartDataField; for example, as a pick list or an ActiveX Control. You can create any user interface as long as it does the following:

- Retrieves the containing SmartDataViewer’s `DataValue` property value for the associated field and assigns it to the SmartDataField’s visualization of the field.
- Accepts a modified value from the visualization and assigns it to the `DataValue` property for the associated field.

For information about creating a SmartDataField and adding it to a SmartDataViewer, see the “SmartDataField usage notes” section on page 2–24.

Note that you might want to add any special SmartDataFields you create to the AppBuilder palette so you can easily reuse them in your applications. The ADM already includes one special SmartDataField: a SmartSelect. This SmartDataField, which is available from the menu that appears when you right-click on the SmartDataField icon, performs a field lookup. See the “SmartSelects” section on page 2–43.

Table 2–8 lists the SmartDataField files.

### Table 2–8: SmartDataField files

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename / SmartLinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template file</td>
<td><code>src/adm2/template/field.w</code></td>
</tr>
<tr>
<td>Primary include file</td>
<td><code>src/adm2/field.i</code></td>
</tr>
<tr>
<td>ADM/Progress Advisor-supported SmartLinks</td>
<td>– (None)</td>
</tr>
<tr>
<td>Instance properties dialog box</td>
<td>Implementation dependent</td>
</tr>
</tbody>
</table>
SmartDataField instance properties

Figure 2–6 shows an example of an instance properties dialog box for a SmartDataField.

![Figure 2–6: SmartDataField instance properties dialog box](image)

Recall there is not a standard instance properties dialog box for a SmartDataField; therefore, the appearance of the instance properties dialog box for a SmartDataField—and the properties available from it—are implementation-dependent.

SmartDataField usage notes

This section discusses special programming considerations for using SmartDataFields.

Creating SmartDataFields

Because SmartDataFields do not have a wizard, you must create them yourself. Here are the general steps for creating a SmartDataField, which you perform in the AppBuilder:

1. Define the visualization of the SmartDataField; for example, by dropping an ActiveX Control onto it.

2. Write code to set the DataModified property; for example, when the field’s value is changed in an AfterUpdate ActiveX event trigger.

3. Implement the setDataValue and getDataValue functions (the template includes stubs for these functions):

   - The setDataValue function, which is run from displayFields in the SmartDataViewer, sets the value of the field’s visualization by passing in the SmartDataObject field value to be assigned to the visualization.
   - The getDataValue function returns the current value of the field’s visualization to the caller (typically collectChanges in the SmartDataViewer).
   - To signal the Table-IO source that the user modified data, create a VALUE-CHANGED trigger for the SmartDataField that contains the following single line:

     ```{set DataModified yes}```
4. Implement the `disableField` and `enableField` procedures to enable and disable the visualization (the template includes stubs for these procedures). There also are `hasDisplayField` and `hasEnableField` properties:

- The `displayField` property specifies whether the data value for the field is assigned to the SmartDataField automatically.
- The `enableField` property specifies whether the SmartDataField is updateable.

By default, these properties are set based on the corresponding properties for the original SmartDataObject field, but they can be set in the instance properties dialog box.

5. Modify the use list for the SmartDataField icon in the `smart.cst` file to include your special SmartDataField. This enables you to reuse your SmartObject by selecting it from the AppBuilder palette.

For instructions on using the AppBuilder, see *OpenEdge Development: AppBuilder*.

**Adding SmartDataFields to SmartDataViewers**

You add SmartDataFields to a SmartDataViewer after you pick all of the fields that will appear in the viewer (using the wizard). To add a SmartDataField, select it from the AppBuilder palette and drop it onto the SmartDataObject field it is to replace. As when you add SmartObjects to other SmartContainers, the AppBuilder then generates an `adm-create-objects` procedure to start the SmartDataField, including a `fieldName` property whose value is the name of the original SmartDataObject field.
SmartPanels

SmartPanels are ADM panel-class SmartObjects that group a number of functionally related buttons onto a frame. The ADM provides separate SmartPanel masters that support specific types of record-management activities, as follows:

- **Navigation SmartPanel masters** support navigating records. There are two Navigation SmartPanel masters: an Icon SmartPanel master that contains buttons using icons to indicate the navigation direction and a Labels SmartPanel master that uses text labels instead of icons. You link instances created from these masters to one or more SmartDataObjects with Navigation SmartLinks.

- **The Update SmartPanel master** supports modifications to database records. It runs in either save or update mode. You link an instance created from this master to one or more updateable objects, such as SmartDataViewers or SmartDataBrowsers, with TableIO SmartLinks.

- **The Commit SmartPanel master** supports committing multiple updates to database records in a single transaction. You link an instance created from this master to one or more SmartDataObjects with Commit SmartLinks.

For more information about using SmartPanels to manage records in the ADM, see Chapter 4, “Data Management in the ADM.”

Table 2–9 lists SmartPanel files.

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename / SmartLinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master files</td>
<td>src/adm2/pnavico.w&lt;br&gt;(source code for Navigation with Icons master)&lt;br&gt;gui/adm2/pnavico.r&lt;br&gt;(compiled code for Navigation with Icons master)&lt;br&gt;src/adm2/pnavlbl.w&lt;br&gt;(source code for Navigation with Labels master)&lt;br&gt;gui/adm2/pnavlbl.r&lt;br&gt;(compiled code for Navigation with Labels master)&lt;br&gt;src/adm2/pupdsav.w&lt;br&gt;(source code for Update master)&lt;br&gt;gui/adm2/pupdsav.r&lt;br&gt;(compiled code for Update master)&lt;br&gt;src/adm2/pcommit.w&lt;br&gt;(source code for Commit master)&lt;br&gt;gui/adm2/pcommit.r&lt;br&gt;(compiled code for Commit master)</td>
</tr>
<tr>
<td>Primary include file</td>
<td>src/adm2/panel.i</td>
</tr>
</tbody>
</table>
Table 2–9: SmartPanel files (2 of 2)

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename / SmartLinks</th>
</tr>
</thead>
</table>
| ADM/Progress Advisor-supported SmartLinks | Commit-Source (pcommit.w)  
Navigation-Source (pnavico.w and pnavlb1.w)  
TableIO-Source (pupdsav.w) |
| Instance properties dialog boxes | src/adm2/support/n-paneld.w (source code for Navigation with Icons and Navigation with Labels panels)  
gui/adm2/support/n-paneld.r (compiled code for Navigation with Icons and Navigation with Labels panels)  
src/adm2/support/u-paneld.w (source code for Update panel)  
gui/adm2/support/u-paneld.r (compiled code for Update panel) |

SmartPanel instance properties

Navigation and Update SmartPanels have instance properties dialog boxes but the Commit SmartPanel does not. This is because Commit SmartPanels do not have any instance properties.

Navigation SmartPanel instance properties

Figure 2–7 shows the instance properties dialog box for a Navigation SmartPanel.

![Navigation SmartPanel Attributes dialog box](image)

Figure 2–7: Navigation SmartPanel Attributes dialog box

You use the instance properties in this dialog box as follows:

- **Show Border** — This specifies whether to create a decorative rectangular border around the SmartPanel.

- **Edge Pixels** — If positive, this value specifies the width of the decorative rectangular border to be created with Show Border. Setting it to zero removes the border.
• 'First' on Left, ‘First’ on Right — These determine the actions associated with the navigation radio buttons. If you specify ‘First’ on Left (the default), the radio buttons perform Get-First, Get-Prev, Get-Next, and Get-Last actions in order from left to right. If you specify ‘First’ on Right, the order of these actions is reversed: the radio buttons perform Get-First, Get-Prev, Get-Next, and Get-Last actions in order from right to left.

Note: The ‘First’ on Left and ‘First’ on Right properties are enabled for the Icons version of the Navigation SmartPanel master (p-navico.w) but not the Labels version (p-navlbl.w).

Update SmartPanel instance properties

Figure 2–8 shows the instance properties dialog box for an Update SmartPanel.

![Table0 SmartPanel Attributes](image)

You use the instance properties in this dialog box as follows:

- **Show Border** — This specifies whether to create a decorative rectangular border around the SmartPanel.

- **Edge Pixels** — If positive, this specifies the width of the decorative rectangular border. Setting this value to zero removes the border.

- **Save, Update** — This specifies whether the Update SmartPanel is run in Save mode or Update mode. For descriptions of these modes, see the “Update SmartPanel” section on page 4–10.

- **Add One Record, Add Multiple Records** — This specifies whether the SmartPanel’s Add button prompts the application user for a single new record or multiple new records at run time.
SmartPanel usage notes

This section discusses special programming considerations for using SmartPanels.

Resizing SmartPanels at design time

SmartPanel instances are resizable at design time. All buttons are resized and reconfigured by the resizeObject procedure in the panel.p file. You can make SmartPanel buttons larger or smaller, rearrange them in rows, or make their display vertical instead of horizontal. Buttons you add to a SmartPanel are automatically resized with the other buttons.

SmartPanels as toolbars

In prior ADM releases, a common way to provide toolbar-type functionality in Progress applications was to use a Navigation SmartPanel and an Update SmartPanel collectively as a sort of toolbar. If the containing SmartObject is a SmartWindow, this is no longer necessary, as you can use a SmartToolbar instead: a SmartObject that provides a toolbar and/or menu interface to application commands such as running other windows or procedures or performing navigation and transaction actions in the current object. For details, see the “SmartToolbars” section on page 2–39.
SmartObjects

SmartFolders

A SmartFolder is an ADM folder-class SmartObject that provides an interface for selecting pages in a SmartContainer. You can modify a SmartFolder instance at design time by resizing it or by modifying instance properties in its instance properties dialog box.

Table 2–10 lists the SmartFolder files.

Table 2–10: SmartFolder files

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename / SmartLinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master files</td>
<td>src/adm2/folder.w (source code)</td>
</tr>
<tr>
<td>(Because SmartFolders are customized at design time, a template is not necessary.)</td>
<td>gui/adm2/folder.r (compiled code)</td>
</tr>
<tr>
<td>Primary include file</td>
<td>None</td>
</tr>
<tr>
<td>ADM/Progress Advisor-supported SmartLinks</td>
<td>Page-Source</td>
</tr>
<tr>
<td>Instance properties dialog box</td>
<td>src/adm2/support/folderd.w (source code)</td>
</tr>
<tr>
<td></td>
<td>gui/adm2/support/folderd.r (compiled code)</td>
</tr>
</tbody>
</table>

SmartFolder instance properties

Figure 2–9 shows the SmartFolder instance properties dialog box.

Figure 2–9: SmartFolder Properties dialog box
You use the instance properties in this dialog box as follows:

- **Tab Labels** — This specifies labels for the tabs in the SmartFolder. The **Label** field contains the label text; the tab numbers in the **tab#** field correspond directly to page numbers in the SmartContainer that contains the SmartFolder. To leave a blank space with no tab in the SmartFolder, insert a tab and leave its label blank.

- **Font** — This opens the standard font dialog box, in which you specify the font for the SmartFolder’s tab labels. The default font is the “default” selection in the dialog box. The tab height is adjusted according to the font you select, and the label on the **Font** button changes to use the font you specify.

- **Insert** — This adds a new tab after the selected tab.

- **Remove** — This deletes the selected tab and moves all tabs that were to its right exactly one position to the left, to fill the gap.

- **Use Fixed Tabs** — If checked, this specifies that all SmartFolder tabs have the same width and enables the **Width** in field, where you supply this value. The default is to allow dynamic sizing, which adjusts the tab widths relative to the lengths of the tab labels.

- **Width** — This specifies a fixed width for the SmartFolder tabs. It is enabled only if you check **Use Fixed Tabs**.

**SmartFolder usage notes**

This section discusses special programming considerations for using SmartFolders.

**Page 0 in SmartFolders**

It is possible to set Page 0 in a SmartFolder. Page 0 visualizes at either design time or run time. For example, **Figure 2–10** shows a run-time SmartFolder open to Page 0.

![Folder on Page 0](image)

**Figure 2–10:** SmartFolder page 0
SmartFilters

A SmartFilter is an ADM filter-class SmartObject that comprises a frame containing data fields that are used to gather user-specified selection criteria for filtering the data in a SmartDataObject. It follows the query-by-fields model and is implemented as a dynamic SmartObject; that is, the selection criteria (the filter) are specified at run time rather than at design time.

Because the SmartFilter is implemented as a dynamic SmartObject, it does not have a wizard. At design time, it opens as a frame that contains only three visual objects: buttons labeled Apply, Blank, and Reset. You then use the Link Advisor specify a SmartDataObject as a data source for the SmartFilter instance. You also specify its filter fields: the data-source fields in which the application user can specify selection criteria. You do this by modifying instance properties in its instance properties dialog box.

At run time, the user specifies selection criteria in the filter fields, using the buttons as needed:

- The **Apply** button applies the currently specified filter to the base query.
- The **Blank** button removes the currently specified filter, leaving the selection fields undefined.
- The **Reset** button resets the selection fields to the most recently applied filter.

Table 2–11 lists the SmartDataObject files.

**Table 2–11: SmartFilter files**

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename / SmartLinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master file</td>
<td>src/adm2/dynfilter.w</td>
</tr>
<tr>
<td>(Because a SmartFilter’s fields are defined at design time and its selection criteria are defined dynamically at run time, a template is not necessary.)</td>
<td></td>
</tr>
<tr>
<td>Primary include file</td>
<td>src/adm2/filter.i</td>
</tr>
<tr>
<td>ADM/Progress Advisor-supported SmartLinks</td>
<td>Filter-Source</td>
</tr>
<tr>
<td>Instance properties dialog box</td>
<td>src/adm2/support/filterd.w (source code) gui/adm2/support/filterd.r (compiled code)</td>
</tr>
</tbody>
</table>

**SmartFilter instance properties**

The main purpose of SmartFilter instance properties is to allow you to select fields from the SmartDataObject that serves as the SmartFilter’s Filter-Target and customize how application users specify filter data.
Figure 2–11 shows the SmartFilter instance properties dialog box.

![SmartFilter Properties dialog box](image)

**Figure 2–11: SmartFilter Properties dialog box**

Instance properties in this dialog box are grouped into the following panes:

- Data pane
- Field Properties pane
- Style pane
- String Operators pane
- Operator View as pane
- Size & Position pane

A final instance property, View, is not in any group.

You use these instance properties as described in the following sections.
Data pane

You use the instance properties in the **Data** pane to define the SmartDataObject that will serve as the SmartFilter’s Filter-Target and specify the SmartDataObject fields that are available to the SmartFilter:

- **Target, Browse** — If your SmartFilter already is linked to a SmartDataObject that will serve as its Filter-Target, the **Target** field lists that SmartDataObject and is disabled; you cannot specify another value.

  The **Target** field is enabled, however, if you open the instance properties dialog box without linking the SmartFilter to a Filter-Target. You might want to create a SmartFilter without a link, for example, because you placed it in a SmartContainer and intend to link it to a Filter-Target in another SmartContainer. The **Browse** button opens a window in which you can browse to and select a SmartDataObject.

  **Note:** This SmartDataObject is only design-time SmartDataObject. You must ensure it is linked at run time.

- **Fields, Edit Field List** — The **Fields** list box lists the filter fields that will appear in the SmartFilter, in the order in which they will be shown. The **Edit Field List** button opens a window in which you can select, deselect, and reorder these fields.

Field Properties pane

Because the fields that appear in the SmartFilter are generated dynamically, you cannot access most of their properties in the AppBuilder; however, the SmartFilter’s instance property provides access to a few field properties. Select a field in the **Fields** list box, and its property values appear in the **Field Properties** pane.

Changes entered into a control in the **Field Properties** pane take effect as soon as you select another field in the **Fields** list box. Pressing **OK** in the dialog box saves all field property value changes, including any that are not displayed:

- **Label, Filter Target** — The **Label** fill-in field displays the label of the selected field. By default, the AppBuilder uses the label from the data source and disables this field. To enable the **Label** field so you can enter your own label, uncheck **Filter Target**.

- **Width, Default** — The **Width** fill-in field displays the width of the selected field. By default, the AppBuilder uses the default field width and disables this field. (The default field width is data-type dependent and is specified in the **Size & Position** pane. See the “**Size & Position pane**” section on page 2–37.) To enable the **Width** field so you can enter your own value, uncheck **Default**.

- **Tooltip** — Enter the text of the tooltip for this field. There is no default.

- **Help ID** — Enter the context ID for the relevant help topic. There is no default.
• **View as range fields, Explicit operator**— These check boxes override the default filter style (see the “Style pane” section on page 2–35 for details) for the field selected in the **Fields** fill-in field:

  – The **View as range fields** check box gives the selected field the same visualization as the Range style gives to all fields.

  – The **Explicit operator** check box assigns the same operator to the selected field as the **Explicit** style gives to all fields.

You can check only one of these boxes. One or both might be disabled, depending on your choice of default filter style:

  – If the default filter style is **Inline**, neither check box is enabled.

  – If the style is **Explicit** or **Implicit**, only the **View as range fields** check box is enabled.

  – If the style is **Range**, only the **Explicit operator** check box is enabled.

**Style pane**

You use this set of radio buttons to specify the default filter style, which defines how the **AppBuilder** visualizes the fields in the SmartFilter and applies the application user’s filter criteria to the query:

• **Implicit** — This generates a single list of fill-in fields in which the application user can specify filter data at run time. Each field is assigned the appropriate data type, thus restricting the filter data entered by the user to the correct data type. You specify in the associated combo box the implicit operator to be used with the filter data.

In some cases, other SmartFilter properties might override the default behavior of the Implicit style:

  – If you check the **BEGINS** property, the AppBuilder uses the **BEGINS** operator for character fields regardless of your choice of operator. (This is the default.)

  – If you check the **CONTAINS** property, the AppBuilder uses the **CONTAINS** operator for filter fields that are mapped to word-indexed database fields, regardless of your choice of operator. In addition, these fields are visualized as editors rather than fill-in fields, with their height and word-wrap capabilities regulated by the **Number of Lines in Editors** property.

See the “String Operators pane” section on page 2–37 for details on the **BEGINS** and **CONTAINS** properties. For details on the **Number of Lines in Editors** property, see the “Size & Position pane” section on page 2–37.

The **Implicit** style supports wild cards, but only if you do not check the **BEGINS** property. There is one exception: a value that starts with a wild card always generates an expression with the Progress **MATCHES** keyword.
- **Explicit** — This shows, after a filter field, a list of selectable operators from which the application user can select at run time. The operator is visualized as either a combo box or a radio set, depending on how you set the properties in the **Operator View as pane**. For details, see the “Operator View as pane” section on page 2–37.

Selecting the **Explicit** style disables the **BEGINS** property. (This is because **BEGINS** is in the operator list for character fields.) For details, see the “String Operators pane” section on page 2–37.

- **Range** — This generates two lists of each filter field, allowing the application user to specify a range of inclusive values as filter data at run time. The left-hand set of fields support wild cards.

Selecting the **Range** style disables the **BEGINS** property. However, the **CONTAINS** string operator property might override the default behavior of the **Range** style. If you check this property, the AppBuilder uses the **CONTAINS** operator for filter fields that are mapped to word-indexed database fields, instead of a range of values. In addition, these fields are visualized as editors rather than fill-in fields, with their height and word-wrap capabilities regulated by the **Number of Lines in Editors** property.

See the “String Operators pane” section on page 2–37 for details on the **BEGINS** and **CONTAINS** properties. For details on the **Number of Lines in Editors** property, see the “Size & Position pane” section on page 2–37.

- **Inline** — This generates a single list of fields for which the application user can enter the filter data and an operator at run time. All fields are represented as character fields to allow typing the operator names. The default operator, which the AppBuilder uses if the user does not enter an operator, is **EQUALS**.

In some cases, other SmartFilter properties might override the default behavior of the **Inline** style:

- If you check the **BEGINS** property, the AppBuilder uses the **BEGINS** operator as the default operator for character fields instead of **EQUALS**.

- If you check the **CONTAINS** property, the AppBuilder uses the **CONTAINS** operator for filter fields that are mapped to word-indexed database fields, regardless of the user’s choice of operator (or the default operator, if relevant). In addition, these fields are visualized as editors rather than fill-in fields, with their height and word-wrap capabilities regulated by the **Number of Lines in Editors** property.

See the “String Operators pane” section on page 2–37 for details on the **BEGINS** and **CONTAINS** properties. For details on the **Number of Lines in Editors** property, see the “Size & Position pane” section on page 2–37.

The **Inline** style supports wild cards, but only if you do not check the **BEGINS** property. There is one exception: a value that starts with a wild card always generates an expression with the Progress **MATCHES** keyword.
String Operators pane

You use these check boxes to specify certain overrides for the default filter styles. For details, see the “Style pane” section on page 2–35:

- **BEGINS** — If checked, this specifies the following:
  - If the default filter style is **Implicit**, Progress uses the BEGINS operator for character fields instead of the implicit operator that you specify.
  - If the default filter style is **Inline**, Progress uses BEGINS as the default operator for character fields. (The default operator is used when application users do not specify an operator.)

The BEGINS property is disabled for the **Explicit** and **Range** filter styles.

- **CONTAINS** — If this is checked, and the default filter style is **Implicit**, **Range**, or **Inline**, specifies that the AppBuilder uses CONTAINS as the operator for filter fields that are mapped to word-wrapped database fields, regardless of the operator specified for that style. In addition, these fields are visualized as editors rather than fill-in fields, with their height and word-wrap capabilities regulated by the **Number of Lines in Editors** property. For details, see the “Size & Position pane” section on page 2–37.

Operator View as pane

You use this set of radio buttons to specify how the AppBuilder visualizes the operator selection provided when you select **Explicit** as the default filter style. For details, see the “Style pane” section on page 2–40:

- **Combo-box** — This visualizes the operator selection as a combo box.
- **Radio-set** — This visualizes the operator selection as a radio set.

These radio buttons are disabled if the default filter style is **Implicit** or **Inline**. They are enabled if the default filter style is **Range**, in case you choose to override the style for a specific field to **Explicit** with the **Explicit operator** property. For details, see the “Field Properties pane” section on page 2–34.

Size & Position pane

You use the instance properties in the **Size & Position** pane to define various aspects of filter fields:

- **Width of Character Fields** — This specifies the default width of character fields.
- **Width of Other Fields** — This specifies the default width of noncharacter fields.
- **Column** — this specifies the column position of all fields.

**Note:** This is the position of the actual field and not the label.
• **Number of Lines in Editors** — When you check the CONTAINS property (see the “String Operators pane” section on page 2–37), the AppBuilder visualizes certain filter fields as editors rather than fill-in fields. This property specifies the number of lines to use for these editors. If you specify 1, the editor has the same height as fill-in fields and word wrapping is disabled. If you specify 2, the editor shows two lines, word wrapping is enabled, and a vertical scroll bar appears.

**View property**

If checked (the default), the View check box makes the SmartFilter visible.

**SmartFilter usage notes**

This section discusses special programming considerations for using SmartFilters.

**Filter link as pass-through link**

The Filter link that connects a SmartFilter and its data source can be implemented as a pass-through link. This allows you to create a SmartFilter in a separate SmartWindow and wait to link it to the Filter-target until you drop the SmartFilter’s SmartWindow into the Filter-target’s SmartWindow.

You can start a SmartFilter that is in a separate SmartWindow from a SmartToolbar. Follow these general steps to do this:

1. Create a SmartFilter in a separate SmartWindow, specifying all necessary instance properties, and link it temporarily as a Filter-source to THIS-PROCEDURE (the SmartWindow).

2. Drop this SmartWindow in a SmartWindow with a potential filter—a SmartDataObject that matches the SmartFilter—and answer Yes to the Link Advisor’s prompt about adding the Filter link to the SmartDataObject.

3. Optionally, specify HideOnInit and DisableOnInit in the SmartWindow’s instance properties dialog box and start the main SmartWindow.

4. Check the Filter check box in the SmartToolbar’s instance properties dialog box.

The SmartToolbar can now view and start the SmartFilter’s SmartWindow.
SmartToolbars

A SmartToolbar is an ADM toolbar-class SmartObject that provides a toolbar and/or menu interface to application commands that perform various actions; for example, running other windows or procedures or performing navigation and transaction actions in the current object. You can contain a SmartToolbar in a SmartWindow but not in a SmartFrame or a SmartDialog, and you can put only one SmartToolbar instance in a given SmartWindow.

When a SmartToolbar instance opens, it contains a default menu and toolbar that provide access to the basic actions that application users perform. You can modify several aspects of the SmartToolbar at design time through its instance properties dialog box: whether to use both the menu and toolbar or only one of them, which actions to include, and selected presentation and behavior details. (For a note on more complex modifications, see the “SmartToolbar usage notes” section on page 2–42.)

The rules and data for the actions accessible in the SmartToolbar are defined in an action class from which the SmartToolbar inherits and are totally independent of the SmartToolbar’s layout and structure. As a result, all SmartToolbar actions have the same text, image, and interface throughout the application regardless of the SmartToolbar layout.

Table 2–12 lists the SmartDataObject files.

Table 2–12:  SmartToolbar files

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename / SmartLinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master file</td>
<td>src/adm2/dyntoolbar.w</td>
</tr>
<tr>
<td>(Because SmartToolbars are</td>
<td></td>
</tr>
<tr>
<td>customized at design time, a</td>
<td></td>
</tr>
<tr>
<td>template is not necessary.)</td>
<td></td>
</tr>
<tr>
<td>Primary include file</td>
<td>src/adm2/toolbar.i</td>
</tr>
<tr>
<td>ADM/Progress Advisor-supported</td>
<td>Navigation-Source TableIO-Source</td>
</tr>
<tr>
<td>SmartLinks</td>
<td></td>
</tr>
<tr>
<td><strong>Note:</strong> These links are supported</td>
<td></td>
</tr>
<tr>
<td>by default. The SmartToolbar also can</td>
<td></td>
</tr>
<tr>
<td>also support the Commit-Source</td>
<td></td>
</tr>
<tr>
<td>SmartLink depending on how you set</td>
<td></td>
</tr>
<tr>
<td>the Commit instance property.</td>
<td></td>
</tr>
<tr>
<td>Instance properties dialog</td>
<td>src/adm2/support/toold.w (source code)</td>
</tr>
<tr>
<td></td>
<td>gui/adm2/support/toold.r (compiled code)</td>
</tr>
</tbody>
</table>
SmartObjects

SmartToolbar instance properties

Figure 2–12 shows the SmartToolbar instance properties dialog box:

![SmartToolbar Properties dialog box]

Figure 2–12: SmartToolbar Properties dialog box

The instance properties in this dialog box are grouped into two panes containing Style and Content. Figure 2–12 shows the default properties; the available properties might change if you add or remove actions from the SmartToolbar or the action class (on which the SmartToolbar is based). You use these instance properties as described in the following sections.

Style pane

You use the instance properties in the Style pane to specify how your SmartToolbar instance provides access to toolbar actions. You can specify a menu bar, a toolbar, or a menu bar and a toolbar. You cannot deselect both the menu bar and the toolbar. The Style pane contains the following items:

- **Menu** — If checked (the default), this specifies that the SmartToolbar instance provides a menu bar with the two menus (File and Navigation) shown in Figure 2–13.

![SmartToolbar menus]

Figure 2–13: SmartToolbar menus

Figure 2–13 shows how the menus look when all possible commands are enabled. Various commands are present or absent from the menu bar depending on how you set the instance properties in the Contents pane at design time.
• **Toolbar** — If checked (the default), this specifies that the SmartToolbar instance provides a toolbar, shown in Figure 2–14.

![SmartToolbar toolbar](image)

**Figure 2–14: SmartToolbar toolbar**

Figure 2–14 shows how the toolbar looks when all possible icons are displayed. Various icons are present or absent depending on how you set the instance properties in the Contents pane at design time.

• **Show Border** — If checked, this adds a graphic border that surrounds the toolbar and divides the icon groups. This property is enabled only if the **Toolbar** property is checked.

**Contents pane**

You use the instance properties in the **Contents** pane to specify the actions that application users can perform from the SmartToolbar. Each property represents several actions (as defined by the action class):

• **Navigation** — If checked (the default), enables navigation actions: first record, previous record, next record, and last record. These actions are accessible from the **First**, **Prev**, **Next**, and **Last** commands on the Navigation menu and from the third group of icons (the arrow icons) on the toolbar.

• **TableIO** — If checked (the default), enables TableIO actions: add, update, copy, delete, and save records; reset; and cancel. These actions are accessible from the **Add record**, **Update record**, **Copy record**, **Delete record**, **Save record**, **Reset**, and **Cancel** commands on the **File** menu and the first group of icons on the toolbar.

The **Save** and **Update** radio buttons associated with the **TableIO** check box enable slightly different sets of TableIO actions:

– Clicking the **Update** button includes the **File** menu’s **Update** record and the corresponding icon on the toolbar.

– Clicking the **Save** button omits the **Update** record command and icon.

• **Commit** — If checked, this enables commit actions for updateable objects: undo and commit. These actions are accessible from the **File → Undo** and **File → Commit** commands and from the second group of icons on the toolbar. The default is unchecked.

*Note:* This property is enabled only if you link the SmartToolbar as a Commit-Source to a Commit-Target.

• **Filter** — If checked, this enables calling a SmartFilter that is the SmartToolbar’s Navigation-Target. This functionality is available through the **File → Filter** command and the last icon on the toolbar. For details, see the “Filter link as pass-through link” section on page 2–38.
SmartToolbar usage notes

This section discusses special programming considerations for using SmartToolbars.

SmartPanels as toolbars

In previous ADM releases, a common way to provide toolbar-type functionality in Progress applications was to instance both a Navigation SmartPanel and an Update SmartPanel and use them collectively as a sort of toolbar. If the containing SmartObject is a SmartWindow, this is no longer necessary, as you can now use a SmartToolbar instead.

Complex SmartToolbar modifications

More complex SmartToolbar modifications (for example, the order and structure of icons and menus) involve modifying the SmartToolbar object itself. You manage the layout and availability of actions in a toolbar in the functions initializeMenu( ) and initializeToolbar( ). These functions are not defined in the SmartToolbar super procedure as a default, but they are called from initializeObject and always must exist in the toolbar. Using these functions to customize the SmartToolbar ensures that its structure and layout are defined at the right moment in the initializeObject( ) function.

You can override the default behavior of the SmartToolbar in either a local SmartToolbar master or a local SmartToolbar super procedure:

- **Local SmartToolbar master** — Create a local SmartToolbar master by copying the master (adm2/dyntoolbar.w), then customize the initializeMenu( ) and initializeToolbar( ) functions as required.

  Note that if your application requires two or more versions of a SmartToolbar, you can create local masters for each (the local masters must have different names), then update the .cst files accordingly.

- **Local SmartToolbar super procedure** — Create a local SmartToolbar super procedure by copying the super procedure (adm2/custom/toolbarcustom.p), then customizing its code as required. The code in adm2/dyntoolbar.w's initializeMenu( ) and initializeToolbar( ) functions calls SUPER( ) with the no-error option. If it does not find a super procedure or returns false, the default code is executed.

  You can use this technique to implement a data-driven SmartToolbar that checks properties (for example, user and object) to retrieve the correct SmartToolbar specification data. (If no data is found, return FALSE to trigger the default behavior.)
SmartSelects

A SmartSelect is a special type of SmartDataField that performs a lookup on the associated SmartDataObject field; that is, it retrieves and displays a predefined set of values from which an application user can choose a single value. Inherited SmartDataField behavior ensures that the SmartSelect automatically displays the current field values and that modified SmartSelect values are passed back to the associated SmartDataObject field. Like the basic SmartDataField, the SmartSelect is intended for inclusion only in a SmartDataViewer.

Table 2–13 lists the SmartSelect files.

Table 2–13: SmartSelect files

<table>
<thead>
<tr>
<th>File type</th>
<th>Filename / SmartLinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master file</td>
<td>src/adm2/dynselect.w</td>
</tr>
<tr>
<td>(Because SmartSelects are customized at</td>
<td></td>
</tr>
<tr>
<td>design time, a template is not necessary.)</td>
<td></td>
</tr>
<tr>
<td>Primary include file</td>
<td>src/adm2/select.i</td>
</tr>
<tr>
<td>ADM/Progress Advisor-supported SmartLinks</td>
<td>– (None)</td>
</tr>
<tr>
<td>Instance property dialog</td>
<td>src/adm2/support/selectd.w (source code)</td>
</tr>
<tr>
<td></td>
<td>gui/adm2/support/selectd.r (compiled code)</td>
</tr>
</tbody>
</table>

SmartSelect instance properties

Figure 2–15 shows the default SmartSelect instance properties dialog box (different properties are enabled or disabled if you reset the View as property).
The instance properties in this dialog box are grouped into the following panes:

- **Data Source pane.**
- **Data pane.**
- **Properties pane.**
- **Optional pane.**
- **Visualization pane.**
- **Field Attributes pane.**
- **Geometry pane.**
- **Define Any Key Trigger on Field pane.**

You use these instance properties as described in the following sections.

**Data Source pane**

You use the instance properties in the **Data Source** pane to get and specify information about the SmartDataObject that is the data source for the SmartSelect:

- **SmartDataObject** — This shows the name of the SmartDataObject that serves as the SmartSelect’s data source. This field is populated by dropping a SmartDataObject into the associated SmartDataViewer and linking it to the SmartSelect with a Data SmartLink.

- **Filter** — This specifies an optional filter for the SmartDataObject; for example, an expression such as the following:

```
CreditLimit > 10000
```

Note that you can specify the filter only as a hard-coded value; you cannot specify variables.

- **Instance Properties** — This opens the instance properties dialog box for the SmartDataObject identified by the SmartDataObject property.

**Data pane**

You use the instance properties in the **Data** pane to define the field that the SmartSelect represents:

- **External Field** — This is a non-updateable field in the SmartDataViewer that displays the SmartDataObject field that the SmartSelect represents. This field is populated when you drop the SmartSelect onto a SmartDataViewer’s field.

- **Key Field** — This combo box is for specifying the SmartDataObject field that maps to the SmartDataViewer field that the SmartSelect represents. When the SmartSelect’s instance properties dialog box first appears, this property automatically displays any SmartDataObject field whose name matches the name of the SmartDataViewer field.

- **Display Field** — This combo box is for specifying the SmartDataObject field to display. All the SmartDataObject’s fields are available in the combo box.
• **Event on change** — Enter an event to PUBLISH if the value of the displayed field changes. For more detailed information on using this property (including an example), see the “Event on change instance property” section on page 2–48.

• **Edit Browse Columns** — Enter the fields to display in a browse if the View as property is set to Browser. (See the “Visualization pane” section on page 2–46.) If View as is set to Browser and you do not specify any browse columns, the browser displays the key field and the display field. This button is enabled only when the View as property is set to Browser.

**Properties pane**

You use the instance properties in the Properties pane to specify whether the SmartSelect is enabled and how it displays data:

• **Enable** — If checked (the default), this enables the SmartSelect for use when the containing SmartDataViewer is also enabled for use.

• **Display** — If this is checked (the default), the SmartSelect displays data when the containing SmartDataViewer is displayed.

• **Sort** — If checked (the default), this specifies that the SmartSelect data is sorted on the displayed column.

• **Exit Browser on Default Action** — This field is enabled only when View as is set to Browser. It controls whether the SmartSelect browser exits (closes) when the application user selects a row. If this is unchecked (the default), the browser stays open until it is explicitly closed; otherwise, it exits as soon as the application user selects a row.

• **Cancel on Exit of Browse** — This field is enabled only when View as is set to Browser. It controls whether exiting (closing) the browser is considered a selection. If this is unchecked (the default), exiting behaves as a selection; otherwise, it does not.

• **Reposition Data Source** — This controls whether the Data-Source linked to the SmartDataViewer containing this SmartSelect repositions itself to the record that corresponds with the selected value whenever the value in the SmartSelect changes. If unchecked (the default), the Data-Source does not reposition itself, otherwise it does reposition itself. The default behavior is the implicit behavior for a browser, so this field is disabled when View as is set to Browser.

**Note:** Leaving this option unchecked minimizes performance overhead. (The amount of performance overhead varies depending on how much data there is and where it is.) In some circumstances, however, your application will work correctly only if you check this option; for example, if the Data-Source linked to the containing SmartDataViewer is linked to other objects that must reflect changes in the SmartSelect.
Optional pane

You use the instance properties in the Optional pane to manage the display of no value in the SmartSelect:

- **Optional** — If checked, this indicates that displaying a value in the SmartSelect is optional. This allows the SmartSelect to display no value (represented by the unknown value).

- **No Value** — Enter a value to be displayed when no value (the unknown value) is selected. This field is enabled only when Optional is checked.

Visualization pane

You use the instance properties in the Visualization pane to manage the visualization of the SmartSelect:

- **View as** — A combo box in which to select the type of control used to visualize the SmartSelect. The choices are **Browse**, **Combo-box**, **Selection-list**, and **Radio-set**.

  - If you select **Browse**, the containing SmartDataViewer displays the SmartSelect as a field with a button that, when pressed by the application user, starts a separate window with a browser that displays the SmartSelect data. This browser shows the number of lines specified by the **Inner Lines** property and presents the application user with a search field in which to search on the field specified by the **Display Field** property. (See the “Data pane” section on page 2–44.) The application user selects a row either by double-clicking or by pressing the **RETURN** key or its equivalent.

  - If you select **Combo-box**, a radio-set appears from which you can select a combo box type: **Drop-down-list**, **Drop-down**, or **Simple** (the three types of Progress combo boxes).

  - If you select **Radio-set**, a radio set appears from which you can select an orientation: **horizontal** or **vertical**.

- **Inner Lines** — This specifies the number of lines used to display data in the SmartSelect. This option is disabled if you select **Radio-set**.

Field Attributes pane

You use the instance properties in the Field Attributes pane to specify labeling and formatting information for the SmartSelect field:

- **Label, Data Source** — The Label fill-in field displays the label of the SmartSelect field. By default, the AppBuilder uses the label from the data source and disables this field. To enable the Label field so that you can enter your own label, uncheck the associated Data Source check box.

- **Browse Label** — Enter a title for the browse window if the View as property is set to **Browser**. (See the “Visualization pane” section on page 2–46.) This field is enabled only when the View as property is set to **Browser**.

- **Data Type** — This is the data type of the SmartSelect key field (this field is read only).
• **Format, Data Source** — The **Format** fill-in field displays the format of the SmartSelect key field. By default, the AppBuilder uses the format from the data source and disables this field. To enable the **Format** field so you can enter your own format, uncheck the associated **Data Source** check box. This field is enabled only if you select **Combo-box** as the value of **View as**.

• **Tooltip** — Enter the text of the tooltip for this field. There is no default.

• **Help ID** — Enter the context ID for the relevant help topic. There is no default.

**Geometry pane**

You use the instance properties in the **Geometry** pane to alter the SmartSelect’s size and position. You also can modify the SmartSelect’s geometry by directly manipulating the SmartSelect in the SmartDataViewer Container:

• **Width** — This is the width of the SmartSelect.

• **Height** — This is the height of the SmartSelect. This field is disabled for object types that have a fixed height.

• **Column** — This is the width of the SmartSelect’s field.

• **Row** — This is the height of the SmartSelect’s field.

**Define Any Key Trigger on Field pane**

This pane contains the following items:

• **Define Any Key Trigger on Field** — This field is enabled only when **View as** is set to **Browser**. It controls whether a persistent trigger is defined on the ANY-KEY event of the field. If it is checked (the default), the trigger is defined.

  The main use for a persistent trigger on the ANY-KEY event is to start the SmartSelect’s browser as specified in the **Browse Keys** instance property’s fill-in field. The procedure that runs on this event is the anykey procedure; however, you can have other keys perform actions in the data-source by overriding the anykey procedure in a customized SmartSelect super procedure (adm2/custom/selectcustom.p). For example, you do this to implement combo-box behavior such as navigation on cursor keys.

  See the “Custom class files” section on page 8–12 and the “Writing super procedures” section on page 8–15 for a description of custom files and information on customizing super procedures, respectively.

• **Browse Keys** — Specifies the key label or key function that starts the SmartSelect’s browser from the field. You can supply a comma-separated list.
• **Key** — A button that opens the Keyboard Event dialog box, shown in Figure 2–16.

![Keyboard Event dialog box](image)

**Figure 2–16: Keyboard Event dialog box**

This dialog box traps a key function or key label and returns the value to the fill-in field for the Browse Keys instance property. You can append the selected value to the current value by inserting a comma after the existing value before you press the Key button. Note this dialog box cannot distinguish between key functions and key labels.

**SmartSelect usage notes**

This section discusses special programming considerations for using SmartSelects.

**Event on change instance property**

The Event on change property allows you to specify an event to PUBLISH if the value of a displayed SmartSelect field changes. Doing this, however, entails more than simply specifying an event name; you also must subscribe the SmartSelect’s data source to your new event, define the event, and so on. The following example illustrates how to do this. It assumes you started the AppBuilder and connected to a copy of the Progress sports2000 database.

To specify an event to PUBLISH if the value of a displayed SmartSelect field changes:

1. Create the following SmartObjects:
   - A SmartDataObject named dcust.w against the customer table
   - A SmartDataObject named ds1srep.w against salesrep table
   - A SmartDataViewer for the customer SmartDataObject. Include the Custnum, Name, and SalesRep fields
2. Drop the salesrep SmartDataObject onto the SmartDataViewer.
3. Drop a SmartSelect onto the salesrep field of the SmartDataViewer.
4. Set the SmartSelect instance properties as follows:
   - **SmartDataObject**: ds1srep.w (the salesrep SmartDataObject)
   - **Key Field**: SalesRep
   - **Displayed Field**: SalesRep
   - **Event on change**: SalesRepSelected (you will define this event in a later step)
• View as: Browser
• Label: Sales Rep
• Browse Title: Sales Representatives
• Tooltip: Select Sales Rep for the Order

For all other instance properties, accept the defaults.

5. In the Section Editor, create the following:

• An override procedure for the SmartDataViewer that uses the initializeObject procedure. Note that whether this procedure runs before or after the SmartDataViewer’s standard behavior depends on where you put the code, as the example code illustrates.

• A SUBSCRIBE statement for your event. (Note that h_dynselect is the SmartSelect object name.)

```/*--------------------*/
Purpose: Super Override
Parameters:
Notes:
--------------------*/

/* Code placed here executes PRIOR to the standard behavior. */
RUN SUPER.

/* Code placed here executes AFTER the standard behavior. */
SUBSCRIBE "SalesRepSelected" IN h_dynselect.
END PROCEDURE.
```

6. Using the Section Editor, create an internal procedure that defines your event; for example:

```/*--------------------*/
Purpose: <none>
Parameters: <none>
Notes:
--------------------*/

DEFINE INPUT PARAMETER cNewValue AS CHARACTER NO-UNDO.
MESSAGE "VALUE IS: " cNewValue VIEW-AS ALERT-BOX.
END PROCEDURE.
```

7. Save this procedure with Name = SalesRepSelected.

8. Save the SmartDataViewer object as vcustsls.w.
9. Create a new SmartWindow, then add the following:
   - A customer SmartDataObject (dcust.w).
   - Your modified SmartDataViewer (vcustsls.w).
   - For navigation purposes, a SmartToolbar (accept the defaults).

10. Save this application as the object wcustsls.w, then run it.

11. Choose the Browse icon to view a list of sales-representative names.

12. Select the name of a sales representative. A dialog box appears and displays the value that you selected. This dialog box is based on the SalesRepSelected procedure you defined.
Simple SmartObjects

The template for the simple SmartObject is a starting point for creating new visual SmartObjects. Figure 2–17 shows how an unmodified simple SmartObject appears.

![Figure 2–17: Simple SmartObject](image)

The simple SmartObject is intended primarily as a container for basic objects (Progress 4GL widgets) and 4GL code; however, you also can build a new SmartObject from its template instead of basing it on another SmartObject template (for example, the SmartDataViewer template).

The simple SmartObject template includes a single primary include file, `src/adm2/smart.i`. This is the library common to all SmartObjects. It specifies no Advisor-supported SmartLinks. No instance properties are associated with the simple SmartObject template. You add include files, supported SmartLinks, and instance properties as needed when you create a new SmartObject template based on the basic SmartObject.
SmartLinks

SmartLinks are the mechanism, or pathway, by which SmartObjects communicate with each other.

This chapter discusses SmartLinks in detail in the following sections:

- SmartLink overview
- Source and target objects
- SmartLink events
- Invoking behavior in other linked objects
- Using Data SmartLinks
- Defining SmartLinks in Progress applications
SmartLink overview

SmartLinks are an important element of the ADM architecture. A SmartLink is a bidirectional association of two SmartObjects. It establishes how one SmartObject relates to another and what sort of behavior each can expect from the other. SmartLinks also are part of the overall ADM messaging scheme (communication between SmartObjects).

A SmartObject associated with another by a SmartLink serves as either the link’s source object or its target object. Data flows across the SmartLink primarily from the source to the target, but in some circumstances, from the target to the source.

SmartLinks are created and maintained by each SmartObject’s container. Each link is a property in each SmartObject that supports that type of link. The smart.p super procedure has procedures and functions for link management. Each link is mapped to one or more events that use the PUBLISH and SUBSCRIBE statements. These event lists also are SmartObject properties. For more information on PUBLISH and SUBSCRIBE, see the PUBLISH and SUBSCRIBE entries in OpenEdge Development: ABL Reference.

SmartLink types

The SmartLink type determines the kind of relationship that is established between two objects. Table 3–1 describes the relationships that these SmartLinks establish and gives examples (in parentheses) of SmartObjects that support each SmartLink type.

Table 3–1: ADM-recognized SmartLink types

<table>
<thead>
<tr>
<th>SmartLink type</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td>Links a containing object (SmartContainer) to an object that it contains. One SmartObject contains another if it creates the contained SmartObject, regardless of whether the containing object visually parents the contained object. The ADM automatically establishes all Container SmartLinks at run time. This is why Container SmartLinks do not appear in the AppBuilder.</td>
</tr>
<tr>
<td>Navigation</td>
<td>Links an object that provides a navigation interface (SmartPanel or SmartToolbar) to an object that supplies a query through which to navigate (SmartDataObject).</td>
</tr>
<tr>
<td>TableIO</td>
<td>Links an object that provides a record-modification interface (SmartPanel or SmartToolbar) to an object through which to enter changes (SmartDataViewer or SmartDataBrowser).</td>
</tr>
<tr>
<td>Page</td>
<td>Links an object that provides a page-selection interface (SmartFolder) to an object that manages the hiding and viewing of pages (SmartContainer).</td>
</tr>
<tr>
<td>PageN</td>
<td>Links an object that manages the hiding and viewing of a numbered set of pages (SmartContainer) to the SmartObjects on a specific page (holds the object handle and the page that it is on). Note: Application developers do not normally use this link type.</td>
</tr>
</tbody>
</table>
You are not restricted to using these ADM-recognized SmartLink types. You also can define new SmartLink types and configure SmartObjects to support them.

<table>
<thead>
<tr>
<th>SmartLink type</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>GroupAssign</td>
<td>Links an object that manages a record (SmartDataViewer, considered the master SmartDataViewer) to an object that manages a related record or another view of the same record (SmartDataViewer), to assure that all fields are updated in a single transaction. This is an advanced task. For details, see the “Using GroupAssign SmartLinks” section on page 6–11.</td>
</tr>
<tr>
<td>Data</td>
<td>Links an object that manages records (SmartDataObject) to a visualization object (SmartDataViewer or SmartDataBrowser) or to another object that manages records (SmartDataObject).</td>
</tr>
<tr>
<td>Update</td>
<td>Links a visualization object (SmartDataBrowser or SmartDataViewer) to a SmartDataObject for the purpose of passing updated data values from the visualization object to the SmartDataObject. This is a separate link from the Data link. This is because the data source might not be a SmartDataObject, but all updates must be done through a SmartDataObject. Normally, the Update link associates the same two SmartObjects as a Data link, but in the opposite direction; however, this is not the case if the visualization object gets records from itself or from another source.</td>
</tr>
<tr>
<td>Commit</td>
<td>Links a Commit SmartPanel to a SmartDataObject for committing changes to sets of database records when it is necessary to perform more than one update in a single transaction.</td>
</tr>
<tr>
<td>Filter</td>
<td>Links a SmartFilter to a SmartDataObject.</td>
</tr>
</tbody>
</table>
Source and target objects

A SmartLink connects source SmartObjects and target SmartObjects. A source object can have one or more active target objects, but a target object usually has only one active source object. For example, a single SmartContainer (Container-Source) might contain a SmartDataViewer, a SmartDataBrowser, and a SmartPanel, each of which is a Container-Target.

Sometimes, a single object can be on either end of a SmartLink type. For example:

- A SmartContainer might contain other SmartContainers.
- A SmartDataObject might be the Data-Source for a SmartDataViewer and a Data-Target for another SmartDataObject.

As you design an application screen, you determine how SmartObjects link together; however, different SmartObjects have rules about which SmartLinks they allow. For details on which SmartLinks are allowed for each SmartObject, see Chapter 2, “SmartObjects.”
SmartLink events

For each SmartLink type, the source and target objects have specific responsibilities to each other. These responsibilities are fulfilled by the methods contained in the SmartObject; therefore, the type of links a particular SmartObject can support is determined by the methods it contains.

The specific responsibilities of a pair of linked SmartObjects to each other depend only on the type of SmartLink that connects them, not on their object types. For example, in a TableIO relationship, the target object always has the same responsibilities, regardless of whether it is a SmartDataBrowser or a SmartDataViewer. Specifically:

- A source object Publishes named events to which its target object must subscribe and for which the target object must implement event procedures that execute when published events occur.

- The target object Publishes named events to which its source object must subscribe and for which the source object must implement event procedures that execute when published events occur.

The source and target objects, therefore, must subscribe to the correct set of events, or they cannot interact as expected. These events evaluate to the internal procedures in the source and target objects.

Subscription occurs when you add the SmartLink: the addLink procedure (in smart.p) subscribes the target procedure to the SourceEvents for the link and subscribes the source procedure to the TargetEvents, as shown in the following code fragment:

```abl
/* SUBSCRIBE to all the appropriate events on each side of the link. First SUBSCRIBE the target to all the events it says it wants from its source. */
cEvents = DYNAMIC-FUNCTION("get":U + pcLink + "SourceEvents":U IN phTarget) NO-ERROR.
IF cEvents NE ? THEN
   DO iEvent = 1 TO NUM-ENTRIES(cEvents):
      SUBSCRIBE PROCEDURE phTarget TO ENTRY(iEvent, cEvents) IN phSource.
   END.
/* Then SUBSCRIBE the source to all the events (if any) that it wants from its target. */
cEvents = DYNAMIC-FUNCTION("get":U + pcLink + "TargetEvents":U IN phSource) NO-ERROR.
IF cEvents NE ? THEN
   DO iEvent = 1 TO NUM-ENTRIES(cEvents):
      SUBSCRIBE PROCEDURE phSource TO ENTRY(iEvent, cEvents) IN phTarget.
   END.
RETURN.
END PROCEDURE.
```

This behavior happens automatically, in the addLink procedure.

For more information on Publish and Subscribe, see *OpenEdge Development: ABL Reference*. For a description of dynamic-function, see *OpenEdge Development: ABL Reference*.

The following sections describe the responsibilities of the source and target SmartObjects for each ADM-recognized SmartLink type. For each SmartLink type, these sections also list the subscribed events/internal procedures required by the source and target objects.
Container SmartLink

The Container SmartLink’s source object can create and destroy its target objects. During creation, the source can tell its targets to set their positions relative to the source (if the targets have visualizations).

The source also can tell its targets to parent themselves to the source. If the source is hidden, viewed, or deleted, all its targets also are hidden, viewed, or deleted; this is standard Progress behavior.

To maintain a Container SmartLink, the ADM requires the source and target objects to SUBSCRIBE to and implement the events shown in Table 3–2. Specifically:

- The Container-Target SUBSCRIBEs to the source events in the Container-Source procedure handle and must implement the corresponding internal procedure names.
- The Container-Source SUBSCRIBEs to target events in the Container-Target procedure handle and must implement the corresponding internal procedures.

Table 3–2: Container events

<table>
<thead>
<tr>
<th>ContainerSourceEvents</th>
<th>ContainerTargetEvents</th>
</tr>
</thead>
<tbody>
<tr>
<td>initializeObject</td>
<td>exitObject</td>
</tr>
<tr>
<td>hideObject</td>
<td></td>
</tr>
<tr>
<td>viewObject</td>
<td></td>
</tr>
<tr>
<td>destroyObject</td>
<td></td>
</tr>
<tr>
<td>enableObject</td>
<td></td>
</tr>
<tr>
<td>confirmExit</td>
<td></td>
</tr>
</tbody>
</table>

A Container SmartLink typically connects the following types of SmartObjects:

- **Source** — SmartWindow, SmartFrame, SmartDialog
- **Target** — Any SmartObject

Navigation SmartLink

The Navigation SmartLink’s source object can send navigation requests to its target objects. The source can also manage an interface for navigation (such as **First**, **Next**, **Prev**, and **Last** buttons).
To maintain a Navigation SmartLink, the ADM requires the source and target objects to subscribe to and implement the events shown in Table 3–3. Specifically:

- The Navigation-Target subscribes to the source events in the Navigation-Source procedure handle and must implement the corresponding internal procedure names.
- The Navigation-Source subscribes to target events in the Navigation-Target procedure handle and must implement the corresponding internal procedures.

Table 3–3: Navigation events

<table>
<thead>
<tr>
<th>NavigationSourceEvents</th>
<th>NavigationTargetEvents</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetchFirst</td>
<td>queryPosition</td>
</tr>
<tr>
<td>fetchNext</td>
<td>updateState</td>
</tr>
<tr>
<td>fetchPrev</td>
<td>linkState</td>
</tr>
<tr>
<td>fetchLast</td>
<td>filterState</td>
</tr>
</tbody>
</table>

The Navigation SmartLink typically connects the following types of SmartObjects:

- **Source** — Navigation SmartPanel, SmartToolbar
- **Target** — SmartDataObject

Page SmartLink

The Page SmartLink’s source object can inform its target object that a new page has been selected. The source also can manage an interface for selecting pages (such as tab folders).

To maintain a Page SmartLink, the ADM requires the source object to subscribe to and implement the events shown in Table 3–4. Specifically, the Page-Source subscribes to target events in the Page-Target procedure handle, and must implement the corresponding internal procedures.

Table 3–4: Page events

<table>
<thead>
<tr>
<th>PageSourceEvents</th>
<th>PageTargetEvents</th>
</tr>
</thead>
<tbody>
<tr>
<td>– (None)</td>
<td>changeFolderPage</td>
</tr>
<tr>
<td></td>
<td>deleteFolderPage</td>
</tr>
</tbody>
</table>

The Page relationship typically connects the following types of SmartObjects:

- **Source** — SmartFolder
- **Target** — SmartContainer, SmartWindow, SmartFrame, SmartDialog
**PageN SmartLink**

The PageN SmartLink’s source object can manage a set of pages. The target object is expected to have hide and view capabilities.

The PageN SmartLink has no PageN events—neither source events nor target events.

**Note:** Application developers do not normally use the PageN link type. Instead, the ADM maintains this link type, to identify which objects are on which page of a container for the purposes of hiding and viewing objects, and so on.

---

**TableIO SmartLink**

The TableIO SmartLink’s source object can tell its target objects to perform record modifications.

To maintain a TableIO SmartLink, the ADM requires the source and target objects to **subscribe** to and implement the events shown in Table 3–5. Specifically:

- The TableIO-Target subscribes to the source events in the TableIO-Source procedure handle, and must implement the corresponding internal procedure names.
- The TableIO-Source subscribes to target events in the TableIO-Target procedure handle, and must implement the corresponding internal procedures.

<table>
<thead>
<tr>
<th>TableIO-SourceEvents</th>
<th>TableIO-TargetEvents</th>
</tr>
</thead>
<tbody>
<tr>
<td>addRecord</td>
<td>queryPosition</td>
</tr>
<tr>
<td>updateRecord</td>
<td>updateState</td>
</tr>
<tr>
<td>copyRecord</td>
<td>linkState</td>
</tr>
<tr>
<td>deleteRecord</td>
<td></td>
</tr>
<tr>
<td>resetRecord</td>
<td></td>
</tr>
<tr>
<td>cancelRecord</td>
<td></td>
</tr>
<tr>
<td>updateMode</td>
<td></td>
</tr>
</tbody>
</table>

The TableIO SmartLink typically connects the following types of SmartObjects:

- **Source** — Update SmartPanel, SmartToolbar
- **Target** — SmartDataViewer, SmartDataBrowser
**GroupAssign SmartLink**

The GroupAssign SmartLink’s source object can tell its target objects to either commit or not commit record modifications as part of a single transaction.

To maintain a GroupAssign SmartLink, the ADM requires the source and target objects to **SUBSCRIBE** to and implement the events shown in **Table 3–6**. Specifically:

- The GroupAssign-Target **SUBSCRIBE**s to the source events in the GroupAssign-Source procedure handle, and must implement the corresponding internal procedure names.
- The GroupAssign-Source **SUBSCRIBE**s to target events in the GroupAssign-Target procedure handle, and must implement the corresponding internal procedures.

**Table 3–6: GroupAssign events**

<table>
<thead>
<tr>
<th>GroupAssignSourceEvents</th>
<th>GroupAssignTargetEvents</th>
</tr>
</thead>
<tbody>
<tr>
<td>addRecord</td>
<td>updateState</td>
</tr>
<tr>
<td>copyRecord</td>
<td></td>
</tr>
<tr>
<td>updateRecord</td>
<td></td>
</tr>
<tr>
<td>resetRecord</td>
<td></td>
</tr>
<tr>
<td>cancelRecord</td>
<td></td>
</tr>
<tr>
<td>enableFields</td>
<td></td>
</tr>
<tr>
<td>disableFields</td>
<td></td>
</tr>
<tr>
<td>collectChanges</td>
<td></td>
</tr>
</tbody>
</table>

The GroupAssign SmartLink typically connects the following types of SmartObjects:

- **Source** — SmartDataViewer
- **Target** — SmartDataViewer

**Data SmartLink**

The Data SmartLink’s source object can tell its target objects that a record is waiting in the data source.

To maintain a Data SmartLink, the ADM requires the source and target objects to **SUBSCRIBE** to and implement the events shown in **Table 3–7**. Specifically:

- The Data-Target **SUBSCRIBE**s to the source events in the Data-Source procedure handle, and must implement the corresponding internal procedure names.
- The Data-Source **SUBSCRIBE**s to target events in the Data-Target procedure handle, and must implement the corresponding internal procedures.

**Table 3–7: Data events**

<table>
<thead>
<tr>
<th>DataSourceEvents</th>
<th>DataTargetEvents</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataAvailable</td>
<td>updateState</td>
</tr>
<tr>
<td>queryPosition</td>
<td></td>
</tr>
<tr>
<td>deleteComplete</td>
<td></td>
</tr>
<tr>
<td>fetchDataSet</td>
<td></td>
</tr>
</tbody>
</table>
The Data SmartLink typically connects the following types of SmartObjects:

- **Source** — SmartDataObject
- **Target** — SmartDataObject, SmartDataViewer, SmartDataBrowser

**Update SmartLink**

The Update SmartLink’s source object can tell its target objects which fields have changed and supply the fields’ new values.

The ADM does not require the source and target objects to **SUBSCRIBE** to and implement any events. Instead, the Update relationship is maintained by the Update-Source directly running functions such as submitRow in the Update-Target.

The Update relationship typically connects the following types of SmartObjects:

- **Source** — SmartDataViewer, SmartDataBrowser
- **Target** — SmartDataObject

**Commit SmartLink**

The Commit SmartLink’s source object can tell its target objects to commit or undo the current transaction.

To maintain a Commit SmartLink, the ADM requires the source and target objects to **SUBSCRIBE** to and implement the events shown in Table 3–8. Specifically:

- The Commit-Target **SUBSCRIBE**s to the source events in the Commit-Source procedure handle, and must implement the corresponding internal procedure names.
- The Commit-Source **SUBSCRIBE**s to target events in the Commit-Target procedure handle, and must implement the corresponding internal procedures.

**Table 3–8: Commit Events**

<table>
<thead>
<tr>
<th>CommitSourceEvents</th>
<th>CommitTargetEvents</th>
</tr>
</thead>
<tbody>
<tr>
<td>commitTransaction</td>
<td>rowObjectState</td>
</tr>
<tr>
<td>undoTransaction</td>
<td></td>
</tr>
</tbody>
</table>

The Commit SmartLink typically connects the following types of SmartObjects:

- **Source** — Commit SmartPanel, SmartToolbar
- **Target** — SmartDataObject
Filter SmartLink

The Filter SmartLink’s source object passes its filter data (operators and values) to the target object and removes filter data (for any field and operator combination that this source potentially can apply) in the target. When it finishes, the source opens its query. It reads the current operators and values from its target when it starts and whenever it is reset.

The ADM does not require the source and target objects to SUBSCRIBE to and implement any events. Only the source object knows about a filter relationship. The target object has no knowledge of the source object or why its activities are occurring.

The Filter relationship typically connects the following types of SmartObjects:

- **Source** — SmartFilter
- **Target** — SmartDataObject
Invoking behavior in other linked objects

A SmartObject can execute internal procedures or functions in any SmartObject to which it is linked. This often is necessary to access special behavior or to facilitate communication between the SmartObjects. Accessing these internal procedures or functions does not require the SmartObjects to know each others’ procedure handles.

The ADM accomplishes this using the Progress PUBLISH/SUBSCRIBE syntax. Each SmartLink represents a set of named events. During the creation of a SmartLink, its source automatically subscribes to the named events in the target procedure, and vice versa. For example, the Navigation-Target subscribes to the named events fetchFirst, fetchNext, fetchPrev, and fetchLast in a SmartPanel Navigation-Source. When the application runs and a user presses the Next button, the SmartPanel executes the following statement, which sends a message to its Navigation-Target:

```
PUBLISH 'fetchNext'.
```

This syntax does not require the SmartPanel to keep track of which objects are its Navigation-Targets. Instead, the Progress interpreter tracks this using the SUBSCRIBE statements that the addLink procedure executes on the Navigation-Target’s behalf. (The addLink procedure is executed when the SmartObjects are initialized.)

The standard ADM code contains many such PUBLISH statements. Custom application code can also use PUBLISH statements such as these to communicate between SmartObjects. For example, suppose that you use the addLink procedure as follows to add a SmartLink that is not recognized as one of the object's supported links:

```
RUN addLink(h1, 'newLink', h2)
```

The addLink procedure now executes the following statement:

```
SUBSCRIBE PROCEDURE h2 TO 'newlink' IN h1.
```

The h1 procedure can now execute the PUBLISH statement whenever it wants to invoke the named event in h2:

```
PUBLISH 'newlink'.
```

Each SmartObject includes properties for each link type. These properties, which have names composed of the link type followed by an event name, define the events to which an object is automatically subscribed by the addLink procedure. (See Table 3–2 through Table 3–7 for lists of source and target events.) You can modify these properties for your custom link types.
Pass-through links

Applications invariably contain SmartObjects that are subordinate to SmartContainers, because either they are contained by them (a SmartFrame inside a SmartWindow or another SmartFrame) or they are lower in the SmartContainer hierarchy (a SmartWindow defined as the logical child of another). In some cases, you must be able to connect a SmartObject outside the containing or parenting SmartContainer with its subordinate SmartObject, even though you have direct access only to the SmartContainer. A SmartObject inside a SmartContainer cannot link directly to a SmartObject external to the SmartContainer at design time; however, you can create a two-step link called a **pass-through link** that allows the two objects to communicate. In a pass-through link, the outer SmartObject is the source for the containing or parenting SmartContainer and the containing or parenting SmartContainer is the virtual source for the contained SmartObject.

Figure 3–1 illustrates the two-step pass-through link.

![Figure 3–1: Pass-through link](image)

**Example of setting up a pass-through link**

Suppose you build a special SmartFrame, called an Order SmartFrame, containing these SmartObjects:

- A SmartDataObject for Orders
- A SmartDataBrowser that browses this SmartDataObject’s query
- A SmartDataViewer that displays field values for the current Order
- An Update SmartPanel that allows changes to the current Order record

You create the Order SmartFrame, as a package; that is, as a single reusable object. You want to use an instance of this SmartFrame to display and update Orders for a particular Customer, an operation that requires passing Customer keys (the Cust-Num fields) from a SmartObject outside the SmartFrame to the SmartDataObject it contains. Your application will have as its basis a SmartWindow that will contain an instance of your special SmartFrame and a SmartDataObject and SmartDataBrowser for Customers. You will link the SmartObjects in the SmartWindow, so each time an application user selects a Customer, the SmartDataObject inside the Order SmartFrame receives the Cust-Num value and reopens its query for Orders of that Customer; that is, the Order SmartDataObject will be the Data-Target of the Customer SmartDataObject. Because the containing SmartFrame is between these SmartDataObjects, you cannot link them directly and must establish a pass-through link. In one part of this link, the Order SmartFrame is the virtual Data-Source for its Order SmartDataObject. In the other part of this link, the Customer SmartDataObject is the Data-Source, and the Order SmartFrame is its Data-Target.
You must build the your application in a specific order:

1. Build the inner connection, between the SmartFrame and the contained SmartDataObject.
2. Create the SmartWindow and define its contents.
3. Build the outer connection, between the outer SmartDataObject and the SmartFrame.

At run time, the ADM combines the two separate connections into a single link.

When you build your Order SmartFrame, you define the SmartFrame as the virtual Data-Source for the SmartDataObject that it contains in the Link Editor. To do this, you select THIS-PROCEDURE (which indicates the SmartFrame container you are building) as the Data-Source and the Order SmartDataObject as the Data-Target. You then tell the Order SmartDataObject to expect a Cust-Num key from its Data-Source at run time. The Link Advisor in the AppBuilder cannot do this automatically, because the objects are not in a single SmartContainer; therefore, you must set the ForeignFields property of the Order SmartDataObject so it expects Customer keys. To do this, define a createObjects procedure in the SmartFrame with this code:

```
RUN SUPER.
/* h_order is a handle to the Order SmartDataObject */
DYNAMIC-FUNCTION('setForeignFields' IN h_dorder,"Order.Cust-num, Cust-Num").
```

The RUN SUPER statement runs the standard behavior for createObjects. This runs the AppBuilder-generated adm-create-objects procedure, which creates all the objects in the SmartFrame and establishes the necessary links. After the standard createObjects procedure executes, the setForeignFields function tells the SmartDataObject to expect to receive the Cust-Num field from its Data-Source and to use it to match the Order.Cust-Num field in its own query.

Figure 3–2 illustrates the established inner connection.

![Figure 3–2: Pass-through link: inner connection](image)
You now create a SmartWindow into which you place a Customer SmartDataObject and SmartDataBrowser. Next, drop your Order SmartFrame into the SmartWindow. You want to pass Customer keys to that SmartFrame, so they get to the Order SmartDataObject inside it. You also do this in the Link Editor, by choosing the Customer SmartDataObject as the Data-Source and the SmartFrame as its Data-Target. Theoretically, when you run this SmartWindow, you have two Data links, one from the Customer SmartDataObject to the SmartFrame and one from the SmartFrame to the Order SmartDataObject, as shown in Figure 3–3.

![Figure 3–3: Pass-through link: inner and outer connections](image)

The SmartFrame, however, is not prepared to deal with the Customer keys; the two Data links must be combined into one, so the Customer SmartDataObject and the Order SmartDataObject can communicate directly with each other. To accomplish this, the addLink procedure, which establishes links, notes that the SmartFrame container is both a Data-Target and a Data-Source (this makes the Data link a potential pass-through link) and modifies the inner link by changing its Data-Source from the SmartFrame to the Customer SmartDataObject. There is now a link from the Customer SmartDataObject directly to the Order SmartDataObject, and the two objects can function as if they are in the same container. With a pass-through link, it is the link rather than the data that is passed through.

The outer link, from the Customer SmartDataObject to the SmartFrame, is maintained. This enables the code in the SmartFrame to use Customer keys passed from the Customer SmartDataObject. There is no standard code in the SmartFrame object or its super procedure container p to handle the data. If desired, you can write application code to respond to events from the Customer SmartDataObject.

### Getting records from an additional source

As with most other link types, it is valid for a Data-Target SmartObject to have only one Data-Source at a time. Sometimes, however, you must be able to pass a record down a different path than the normal Data link path. For example, suppose you have a SmartDataViewer that receives Customer values from a SmartDataObject. When the SmartDataObject is notified of the dataAvailable event, it gets the Customer values for the current row in the Customer SmartDataObject. This sequence is repeated whenever the SmartDataObject retrieves a new Customer record. Occasionally, however, the SmartDataViewer might need a different record. For example, it might need the User record on startup, to perform security checks. This User record might not be available through its normal Data-Source.
You can deal with this scenario in several ways. In addition to the default automatic execution of the `colValues` function in the Data-Source from `dataAvailable`, you can program a SmartObject to run `colValues` in a SmartObject other than its regular Data-Source. This approach provides an ADM-standard way to get an extra set of values from one SmartObject to another. For example, you could define an `initializeUser` link from the source of the User ID to the SmartDataViewer that needs it. When you create a nonstandard link, Progress performs a single `SUBSCRIBE` in the target for the link name. For example, if the `initializeUser` procedure is defined in the SmartDataViewer, it will be executed whenever the source `PUBLISHes` the `initializeUser` event. The User ID can be passed as an argument or queried from target to source.
Using Data SmartLinks

Data in the form of field values is passed from one SmartObject to another using a Data SmartLink. A SmartDataObject can be a Data-Source for another object. It also can be a Data-Target, receiving a key value from another query object to use in opening a query. Visual objects such as SmartDataBrowsers and SmartDataViewers also can be Data-Targets. A SmartDataViewer as a Data-Target receives formatted column values from a SmartDataObject for display. A SmartDataBrowser as a Data-Target browses its Data-Source’s query and displays column values.

At design time, when you drop a SmartObject into an application, the AppBuilder’s Link Advisor assesses each SmartObject already in the application to determine whether it is suitable for a Data link connection with the new SmartObject and, if it is, it recommends the connection. It makes its assessment by examining the properties of each object to determine what columns it defines: a query object can be a Data-Source for another object only if it can supply one or more columns that the object could use as a key to open its own query. In particular:

- A query object can be a Data-Source for a visual object such as a SmartDataViewer if all fields displayed by the SmartDataViewer are in the query object and all fields enabled in the SmartDataViewer can be updated in the Data-source.

- A SmartDataBrowser can be a Data-Target only for a SmartDataObject with the same signature. For more information on signatures, see the “Compatibility checking” section on page 4–4.

**Note:** Because of the flexibility of SmartObject combinations and the dynamic nature of queries, the Link Advisor might suggest links that are not desirable for your application. Always examine the Link Advisor’s recommendations before accepting them. One way to avoid problems is to drop Data-Sources onto the design window before Data-Targets. When two objects can be linked in either direction, the Link Advisor assumes the first object dropped into the application is the Data-Source.

When you are linking two query objects, you can use the SmartDataObject’s ForeignFields property at design time to designate which of its fields to use as foreign key fields in SmartDataObjects that are subordinate to it. A corresponding property called ForeignValues holds the values of those fields for the current row. For details, see the “Initializing the foreign key fields” section on page 4–3.
Defining SmartLinks in Progress applications

You are not restricted to using only the SmartLinks installed with the Progress product in your application. You can define additional SmartLinks for the ADM, either static SmartLinks, which use a standard event support structure and whose run-time behavior is predefined, or dynamic SmartLinks, whose run-time behavior varies depending on application-user choices. The following sections provide information on defining both types.

Defining static SmartLinks

When you define a new, custom, static SmartLink, you must provide the following support structure for standard events:

- The template for a SmartObject that will use the new SmartLink must define a preprocessor value called \texttt{ADM-SUPPORTED-LINKS}. This value is a comma-separated list of links this object supports. This list becomes the initial value of the \texttt{SupportedLinks} property when the object is initialized; however, you can modify the associated instance property value at run time (before the object is initialized) if an application condition changes the links that the SmartObject must support.

\textbf{Note:} The Container and PageN links are supported implicitly and do not need to be on this list.

- The source object for the link must define a property named \texttt{linkTarget}, where \texttt{link} is the base link name such as Data, and the target object must define a property \texttt{linkSource}, along with \texttt{getpropname} and \texttt{setpropname} functions to get and set the values for those properties. The data type of the property indicates whether the link supports multiple objects on either end:

  - A \texttt{HANDLE} data type means only one object is allowed; its procedure handle is stored in its native data type. This is the typical case for the source for the standard ADM links, which generally support only one source SmartObject.

  - A \texttt{CHARACTER} data type indicates multiple objects can be at that end of the link; their handles are stored in string format as a comma-separated list. This is typical for the target for the standard ADM links, which generally support multiple target SmartObjects.

For example, a Data-Source can have multiple Data-Targets, but a Data-Target can have only one Data-Source. The \texttt{addLink} procedure uses these properties to store the handles of related objects and determine whether one object or multiple objects are permitted.
Defining SmartLinks in Progress applications

- Any events associated with the link must be defined in a property in the source called
  \texttt{LinkTargetEvents}, where \texttt{Link} is the base link name such as \texttt{Data}, and/or in the target in
  a property called \texttt{LinkSourceEvents}. This property is a comma-separated list of events to
  subscribe to. For example, a SmartContainer (which can be a Container-Source) has the
  following:

  - A \texttt{ContainerTarget} property of type \texttt{CHARACTER}, where the \texttt{addLink} procedure
    stores the handles of its Container-Target procedures at run time.
  - A \texttt{ContainerTargetEvents} property, which is initialized in the property include file
    \texttt{cntnprop.i} to \texttt{exitObject}, the one event a Container must subscribe to in its target.

Likewise, all SmartObjects that can be Container-Targets have the following:

  - A \texttt{ContainerSource} property of type \texttt{HANDLE} (defined in \texttt{smrtprop.i}, because it
    applies to all SmartObjects), where \texttt{addLink} stores the handle of its
    Container-Source at run time.
  - A \texttt{ContainerSourceEvents} property, which is initialized in the property include file
    to a list of all the events to which the Container-Target must subscribe in its
    Container-Source and for which it must implement event procedures
    ("initializeObject,destroyObject ...").

These property values also can be modified at run time, before the object is initialized, to
change the list of events to which the object will subscribe in that particular instance. The
\texttt{addLink} procedure then subscribes to each event in the list using the source and target
handles. Normally, the \texttt{getpropname} and \texttt{setpropname} property functions and the event
procedures are implemented in the super procedure for the object (\texttt{containr.p} for links
defined in \texttt{cntnprop.i}, \texttt{smart.p} for links defined in \texttt{smrtprop.i}, and so on).

Once you account for all these requirements, the ADM takes care of all necessary mechanics of
handling events between SmartObjects automatically.

You also can define standard Supported Links programmatically, by writing calls to \texttt{addLink}
in an application. Likewise, you can remove them by running the \texttt{removeLink} procedure.

### Defining dynamic SmartLinks

Sometimes a SmartObject application requires a link that passes messages between
SmartObjects more dynamically; that is, the particular message to be passed (and any resulting
processing) is determined by an application user’s run-time choice. This type of link is a
dynamic SmartLink.

When you define a dynamic SmartLink, you do not need to provide the support structure for
standard events required for static SmartLinks (see the previous section for details). If, at design
time, you define between two SmartObjects a SmartLink not in the SupportedLinks list for
either object, the \texttt{addLink} procedure registers the link as a subscription to an event of the same
name as the links. (You also can do this programmatically, by running \texttt{addLink} in application
code.)

The following example illustrates how to define a dynamic SmartLink. This SmartLink, called
\texttt{ProcessCalc}, links a simple SmartObject and a SmartDataObject, passing messages that invoke
processing in the SmartDataObject. The particular message sent, and the processing invoked,
are determined at run time by the choice of the application user.
To define the ProcessCalc dynamic SmartLink, in the AppBuilder:

1. Create a SmartDataObject that contains an internal procedure named ProcessCalc that has the following code:

```
DEFINE INPUT PARAMETER pcCalcType AS CHARACTER NO-UNDO.
CASE pcCalcType:
  WHEN "Totals":U THEN RUN calcTotals.
  WHEN "Discounts":U THEN RUN calcDisocunts.
END CASE.
```

2. Create the code for the calcTotals and calcDiscounts procedures.

3. Create a simple SmartObject called sCalcPanel.w that contains two buttons named Calculate Totals and Calculate Discounts, and add the following trigger code to these buttons:

```
ON CHOOSE OF btnCalcTotals DO:
  PUBLISH "ProcessCalc":U (INPUT "Totals":U).
END.

ON CHOOSE OF btnCalcDiscounts DO:
  PUBLISH "ProcessCalc":U (INPUT "Discounts":U).
END.
```

4. Create a SmartWindow.

5. Drop the SmartDataObject onto the SmartWindow.

6. Drop the sCalcPanel.w simple SmartObject onto the SmartWindow.

7. Add a SmartLink as follows:
   a. Invoke the Link Advisor.
   b. Choose Add. The Add a SmartLink dialog box appears.
   c. Choose h_sCalcPanel as the source.
   d. Choose New... as the Link Type.
   e. Enter ProcessCalc as the New Link Type.
   f. Choose the SmartDataObject as the target.
   g. Click OK.
When you run this application:

- Choosing the **Calculate Totals** button invokes `ProcessCalc` in the SmartDataObject with the `Totals` input parameter, which runs the `calcTotals` procedure.

- Choosing the **Calculate Discounts** button invokes `ProcessCalc` in the SmartDataObject with the `Discounts` input parameter, which runs the `calcDiscounts` procedure.

Thus, the `ProcessCalc` SmartLink is dynamic, in the sense that the link name is not bound to a set of event procedure names that are statically defined in the SmartObjects.
The main purpose of most Progress applications is data management: allowing application users to browse, maintain, add, and delete records in databases. Thus, most SmartObjects are related to data management:

- SmartDataObjects work with database records.
- SmartDataViewers and SmartDataBrowsers provide display and browse capabilities for data sets.
- SmartFilters allow the user-specified filtering of queries.
- SmartPanels and SmartToolbars provide screen controls for moving through and updating records managed by the other SmartObjects.

Each of these SmartObjects represents an individual aspect of data-management functions. This chapter describes how they work together to allow you to build applications that manage database records in a variety of ways.

The chapter contains the following sections:

- Displaying and browsing data sets
- Filtering records
- Using SmartPanels and the SmartToolbar
- Transactions and record locking
Displaying and browsing data sets

The SmartDataObject accesses data. SmartDataViewers and SmartDataBrowsers provide various display and browse capabilities for the data sets accessed by the SmartDataObjects to which they are linked:

- A SmartDataObject manages a database query on one or more tables that can be distributed across the AppServer. It defines and retrieves a set of database records.

- A SmartDataViewer views column values it receives from another object, like a SmartDataObject, that is accessing the database. A SmartDataObject and SmartDataViewer, together with a SmartPanel, can retrieve, display, navigate, and update records.

- A SmartDataBrowser allows an application user to browse column values it receives from another object, like a SmartDataObject, that is accessing the database. A SmartDataBrowser also may allow updates to data.

A SmartDataObject has its own database query. A SmartDataObject can pass a key value from one query object to a subordinate object, so the second object can open a query dependent on that key value; for example, Orders of a particular Customer. For more information, see the “Initializing the foreign key fields” section on page 4–3.

All database queries can be modified dynamically so a given SmartDataObject can be used in various application contexts. For example, a SmartDataObject whose query definition is FOR EACH Order might be used in one application context to display and manage all Orders, but in another context, it might be made into a dependent query object that receives a Cust-Num key from another Customer SmartDataObject and opens its query as FOR EACH Order WHERE Cust-Num = key value.

SmartDataObjects

The SmartDataObject accesses a database directly. It transfers rows, as defined by its database query, to a temp-table called RowObject, one batch at a time. (Fields from one or more tables can be joined into a single result table.) For a discussion of the RowObject temp-table, see the “SmartDataObject query and update operations” section on page 7–3.

Once the RowObject temp-table is built, the SmartDataObject presents its rows to SmartDataViewers, SmartDataBrowsers, and other client objects. The SmartDataObject thus insulates the client objects from the database specifics.

When a SmartDataObject reads a data set into a RowObject temp-table, it provides essentially a snapshot of the current state of the data set, often called a view. The RowObject temp-table does not necessarily reflect updates to the data set that are made by other users in another session; however, any code that executes the openQuery procedure on the data set automatically refreshes the RowObject temp-table with the latest data. Alternately, you can use the refreshRow function on the current row to get the latest data.

You can customize various field-related properties of a SmartDataObject to meet your own application needs:

- Select which fields will be a part of the data set presented to other client objects.
- Change the field names.
- Determine which fields the SmartDataObject allows client objects to update.
• Change formats and other field attributes.

• Use calculated fields to express a calculation based on other fields in the RowObject temp-table or as a placeholder for information to be passed to or from client objects that use the SmartDataObject.

SmartDataObjects support update validation logic beyond that defined in the Data Dictionary or in database triggers. For more information, see Chapter 7, “Developing Your Application’s Business Logic.”

**SmartDataViewers**

A SmartDataViewer does not define or open a query; therefore, it always relies on another SmartObject for data. SmartDataViewers never find a database record, and no record buffers are passed to it. This helps separate the user interface from the data management and does not constrain SmartDataViewers to run in a session with a database connection. A SmartDataViewer never has any database records at run time; it displays data from a query defined elsewhere. As with the SmartDataBrowser, all updates must be performed through a SmartDataObject.

**SmartDataBrowsers**

You build a SmartDataBrowser against a SmartDataObject, selecting fields from that SmartDataObject. The SmartDataBrowser browses the SmartDataObject query at run time. In the same way as a SmartDataViewer, the SmartDataBrowser does not actually have any database records at run time; it is browsing a query defined elsewhere. Also, as with the SmartDataViewer, updates to a SmartDataBrowser simply cause modified values to be passed to the SmartDataObject for updating. This method of building a SmartDataBrowser maintains independence of the user interface from the database, since the SmartDataBrowser browses a temp-table query maintained in another object.

Recall there are two types of SmartDataBrowsers: dynamic and static. The dynamic SmartDataBrowser is more versatile; it can be configured to display and update the query of any SmartDataObject, unlike a static SmartDataBrowser, which is specific to a particular SmartDataObject. The dynamic SmartDataBrowser displays and updates records from a set of fields determined at run time (rather than at design time, as is the case for the static SmartDataBrowser) and takes its query at run time from the SmartDataObject to which it is linked.

**Initializing the foreign key fields**

ForeignField1ds is a SmartDataObject property that is set at design time to designate which of its fields to map to foreign key fields in a SmartDataObject that is its parent. A corresponding property called ForeignValues holds the values of those fields for the current row. When ForeignField1ds is set for a particular SmartDataObject, performing the Add or Copy operation in that SmartDataObject assigns the values of the key fields to the corresponding fields in the newly added row.
Suppose you have two SmartDataObjects, CUST and ORDER, that are set up as follows:

- CUST parents ORDER.
- Both these SmartDataObjects have a cust-num field.
- The ForeignFields property in ORDER is set to the cust-num field.
- The current value of cust-num (and, therefore, the ForeignValues property in the ORDERCUST SmartDataObject) is 27.

Figure 4–1 illustrates this setup.

![Diagram](image)

**Figure 4–1: Inheriting foreign key field values**

As the application user adds orders, the value of the ORDER SmartDataObject’s cust-num field is automatically set to the current value of the CUST SmartDataObject’s cust-num field; that is, 27. This is done in the addRow function of data.p to display the ForeignField value as the initial value for the new row. If the application user does not change that value (and such fields normally cannot be updated), the initial value is assigned to the new row in the procedure submitForeignField, called from the function submitRow during the Save operation. See the online help for descriptions of these functions.

**Compatibility checking**

When you instantiate a static SmartDataBrowser or a SmartDataViewer in a Progress application, the AppBuilder must be able to check whether the fields to be browsed or displayed in the SmartObject match (are compatible with) those of the SmartDataObject to which you will link it. Depending on the SmartObject type, the AppBuilder uses either signatures or field lists to perform this check. (Compatibility checking is not required for dynamic SmartDataBrowsers.)

**Signature and field-list checking at design time**

The AppBuilder performs a compatibility check in the following circumstances:

- When you instantiate a SmartDataBrowser or SmartDataViewer in a SmartContainer. The AppBuilder checks all SmartDataObjects in the SmartContainer, to see whether their signatures and field lists match those of the new SmartDataBrowser or SmartDataViewer, respectively. If any matches occur, the Progress Advisor asks which, if any, SmartDataObjects to link to the SmartDataBrowser or SmartDataViewer. If there are no matches, the Advisor does not appear.
• When you instantiate a SmartDataObject in a SmartContainer. The AppBuilder checks all existing SmartDataBrowsers and SmartDataViewers in the SmartContainer to see whether their signatures and field lists, respectively, match those of the new SmartDataObject. If any matches occur, the Progress Advisor asks which, if any, SmartDataBrowsers or SmartDataViewers to link to the SmartDataObject. If there are no matches, the Advisor does not appear.

• When you modify a SmartDataObject. The AppBuilder checks all objects linked to that SmartDataObject for compatibility, and the Progress Advisor notes any incompatibilities.

• When you modify a SmartDataBrowser or SmartDataViewer that is linked to a SmartDataObject. The AppBuilder checks for compatibility, and the Progress Advisor notes any incompatibilities.

You can verify the validity of the signatures of linked objects by clicking on the **Check Links** button in the **SmartLinks** window. (See *OpenEdge Development: AppBuilder* for details.)

### Checking SmartDataBrowser compatibility

Compatibility checking is not required for dynamic SmartDataBrowsers. A dynamic SmartDataBrowser simply attaches to the query of its SmartDataObject and generates a column list at run time, thus avoiding the necessity for a compatibility check.

Compatibility checking for static SmartDataBrowsers uses signatures. As in the case of a SmartDataObject, the temp-table definition of a SmartDataBrowser gives it a unique signature that identifies the order, number, and data types of its fields. The exact use of the signature differs depending on what action is attempted and against which objects the action occurs.

In a compatibility check, the `getDataSignature` function for both the SmartDataObject and static SmartDataBrowser returns a character string with a representation of the object’s signature. The signature of the SmartDataObject is compared to the signature of the static SmartDataBrowser to see whether they are compatible. If so, the static SmartDataBrowser can browse the SmartDataObject’s query when the objects are linked at run time.

### Checking SmartDataViewer compatibility

The field-matching relationship of a SmartDataViewer with a SmartDataObject against which it is built is different from that of a SmartDataBrowser with a SmartDataObject. The AppBuilder uses the SmartDataViewer’s displayed field list, rather than a signature, to determine compatibility with the SmartDataObject. This displayed field list is defined by the SmartDataViewer’s displayed fields and accessed with the `getDisplayedFields` function. If a SmartDataViewer’s displayed fields are the same as or a subset of a SmartDataObject’s fields, the objects are considered to have a matching displayed field signature. A SmartDataViewer can successfully retrieve data from a SmartDataObject only if the SmartDataObject can provide all of the fields that the SmartDataViewer needs.

Similarly, a SmartDataViewer has an enabled field list that is defined by its enabled fields and accessed with the `getEnabledFields` function. If a SmartDataViewer’s enabled fields are the same as or a subset of a SmartDataObject’s updateable fields, the objects are considered to match, according to this enabled field list. A SmartDataViewer can successfully send data to a SmartDataObject only if the SmartDataObject can update all fields the SmartDataViewer provides.
Example: SmartDataObject and SmartDataViewer. The SmartDataObject dcust.w generates the following include file:

**dcust.i**

```plaintext
FIELD Name LIKE Customer.Name VALIDATE~
FIELD Address LIKE Customer.Address VALIDATE~
FIELD City LIKE Customer.City VALIDATE~
FIELD State LIKE Customer.State VALIDATE~
FIELD PostalCode LIKE Customer.PostalCode VALIDATE~
...
```

The updateable fields in this file are Name, Address, and City.

Suppose a SmartDataViewer built against dcust.w enables the Name and Address fields. In this case:

- The SmartDataViewer has a matching displayed field list with the SmartDataObject.
- The SmartDataViewer has a matching enabled field list with the SmartDataObject.

Now, suppose the SmartDataViewer built against dcust.w enables the Name and Postal-Code fields. In this case:

- The SmartDataViewer has a matching displayed field list with the SmartDataObject.
- The SmartDataViewer does not have a matching enabled field list with the SmartDataObject. This is because the SmartDataObject’s PostalCode field is not updateable.

Example: SmartDataObjects with Contrasting Signatures. The SmartDataObject dcust.w generates the following include file:

**dcust.i**

```plaintext
FIELD Name LIKE Customer.Name VALIDATE~
FIELD Address LIKE Customer.Address VALIDATE~
FIELD City LIKE Customer.City VALIDATE~
FIELD State LIKE Customer.State VALIDATE~
FIELD PostalCode LIKE Customer.PostalCode VALIDATE~
...
```

In contrast, the SmartDataObject d-custreverse.w generates the following include file:

**dcustreverse.i**

```plaintext
FIELD PostalCode LIKE Customer.PostalCode VALID
FIELD State LIKE Customer.State VALIDATE~
FIELD City LIKE Customer.City VALIDATE~
FIELD Address LIKE Customer.Address VALIDATE~
FIELD Name LIKE Customer.Name VALIDATE~
...
```
In this example:

- A SmartDataBrowser built against the dcust.w SmartDataObject will not match the d-custreverse.w SmartDataObject. This is because the temp-table signature is used to determine compatibility and the order of the fields is different.

- A SmartDataViewer built against the dcust.w SmartDataObject will match the d-custreverse.w SmartDataObject. This is because the displayed field list is used to determine compatibility and the order of the fields does not matter, only their names and data types.

**Using Data SmartLinks**

Data in the form of field values is passed from one SmartObject to another using a Data SmartLink. A SmartDataObject can be a Data-Source for another object. It also can be a Data-Target, receiving a key value from another query object to use in opening a query. Visual objects such as SmartDataViewers and SmartDataBrowsers also can be Data-Targets. A SmartDataViewer as a Data-Target receives formatted column values from a SmartDataObject for display. A SmartDataBrowser as a Data-Target browses its Data-Source’s query and displays column values in its browse.

At design time, when you drop a SmartObject into an application, the AppBuilder’s Link Advisor assesses each SmartObject already in the application to determine whether it is suitable for a Data link connection with the new SmartObject and, if it is, it recommends the connection. It makes its assessment by examining the properties of each object to determine what columns it defines: a query object can be a Data-Source for another object only if it can supply one or more columns that the object could use as a key to open its own query.

In particular:

- A query object can be a Data-Source for a visual object such as a SmartDataViewer if all fields displayed by the SmartDataViewer are in the query object and all fields enabled in the SmartDataViewer are updateable in the Data-Source.

- A SmartDataBrowser can be a Data-Target only for a SmartDataObject with the same signature.

**Note:** Because of the flexibility of SmartObject combinations and the dynamic nature of queries, the Link Advisor might suggest possible links that are not desirable for your application. Always examine Link Advisor recommendations before accepting them. One way to avoid problems is to drop Data-Sources onto the design window before Data-Targets. When two objects can be linked in either direction, the Link Advisor assumes that the first object dropped into the application should be the Data-Source.
Filtering records

You use the SmartFilter to allow your application users to filter records with user-specified selection criteria rather than built-in criteria. The SmartFilter follows the query-by-form model and is implemented as a dynamic SmartObject; that is, the selection criteria are specified at run time rather than being built into the object.

Progress provides the prebuilt master `dynfilter.w` for the SmartFilter. When you build a SmartFilter, you must define a Filter link to connect the SmartFilter to the SmartDataObject that will serve as its data source, optionally defining that SmartDataObject at this time. (You also can define it later in the instance properties dialog box.) The Filter link can be implemented as a pass-through link (see the “Pass-through links” section on page 3–13.) This allows you to create a SmartFilter in a separate SmartWindow and wait to link it to the Filter-target until you drop the SmartFilter’s SmartWindow into the Filter-Target’s SmartWindow.

Because the SmartFilter is implemented as a dynamic SmartObject, it does not have a wizard, and you modify it at design time through its instance properties. You can specify both the SmartDataObject that will serve as the data source for the records to be filtered and the fields the application user can filter. (See the “Customizing SmartFilters” section on page 4–8.) At run time, the user specifies selection criteria in the filter fields.

The SmartToolbar contains a Filter button that can start a SmartFilter. If the SmartFilter is in the same window as the SmartToolbar, it simply uses a Filter SmartLink. For instructions on linking the SmartToolbar if the SmartFilter is in a separate SmartWindow, see the “Connecting to a SmartFilter in another SmartWindow” section on page 4–9.

Customizing SmartFilters

You customize the SmartFilter object through its instance properties. The SmartFilter has an instance properties dialog box (source program: `src/adm2/support/filterd.w`) that you can use to specify various filter properties. Unless you linked to a data source before you opened the instance properties dialog box, you must at least customize your SmartFilter by providing it with a SmartDataObject that will serve as its data source. By default, all fields are selected, but you can add or delete fields and change the order in which they appear. Because SmartFilter fields are generated dynamically, you cannot access their properties in the instance properties dialog box; however, you can access the most important field properties for each field: its label, width, tooltip, and HelpID.

You also can specify some additional properties of the filter itself:

- Its style: **Implicit**, **Explicit**, **Range**, or **Inline**.
- Whether filter data in a character field is evaluated with **BEGINS**.
- Whether an Explicit operator is visualized using a combo box or a radio set.

See Chapter 2, “SmartObjects,” for descriptions of SmartFilter instance properties.
Connecting to a SmartFilter in another SmartWindow

To link the Filter button in a SmartToolbar to a SmartFilter in a separate SmartWindow:

1. Create a SmartFilter in a separate SmartWindow, specifying all necessary instance properties, and link it temporarily as a Filter-Source to THIS-PROCEDURE (the SmartWindow).

2. Drop this SmartWindow in a SmartWindow with a potential filter—a SmartDataObject that matches the SmartFilter—and answer Yes to the Link Advisor’s prompt about adding the Filter link to the SmartDataObject.

3. Optionally, specify HideOnInit and DisableOnInit in the SmartWindow’s instance properties dialog box, and start the main SmartWindow.

4. Check the Filter check box in the SmartToolbar’s instance properties dialog box.

The SmartToolbar can now view and start the SmartFilter’s SmartWindow.
Using SmartPanels and the SmartToolbar

A SmartPanel is a group (or panel) of buttons that send messages to other SmartObjects. There are several types of SmartPanels, which you can use to manage records in various ways: navigate through them, update them, or perform transaction processing. The SmartToolbar provides a toolbar and/or menu interface for performing one or more of these operations, as well as some additional operations not discussed in this chapter.

Using SmartPanels

Progress provides prebuilt masters for three SmartPanel types: Navigation, Update, and Commit. You can modify these masters to create new SmartPanels with specialized functions, so the ADM does not include a SmartPanel template.

In prior ADM releases, a common use for the Navigation and Update SmartPanels was to place both a Navigation SmartPanel and an Update SmartPanel in a Progress application and use them collectively as a sort of toolbar. If the containing SmartObject is a SmartWindow, this is no longer necessary, as you can provide this combined functionality with a SmartToolbar. See the “Using SmartToolbars” section on page 4–13.

Navigation SmartPanels

Two prebuilt SmartPanels are designed to navigate through a set of records; they support the Navigation link type. These panels contain buttons that represent the functions First, Previous, Next, and Last. They differ only in how they represent these buttons:

- The master file src/adm2/pnavlbl.w has buttons with labels.
- The master file src/adm2/pnavico.w has buttons that resemble VCR icons.

Choosing these buttons publishes the following ADM events (implemented in the procedure src/adm2/query.p), which are received in the SmartPanel’s Navigation-Target:

- **First** invokes fetchFirst.
- **Prev** invokes fetchPrev.
- **Next** invokes fetchNext.
- **Last** invokes fetchLast.

The standard SmartObject designed to be a Navigation-Target is the SmartDataObject; however, if you want your users to be able to use buttons to navigate through records, use a SmartDataBrowser as a visualization for a SmartDataObject. In this case, moving through the rows of the SmartDataBrowser navigates through the SmartDataObject query (although the SmartDataBrowser is not actually the Navigation-Target). The SmartPanels know nothing about the data through which they help to navigate. A custom SmartObject with these four ADM procedures could use them to navigate through any kind of data, including pages in a report, segments of a video, and so on.

Update SmartPanel

Another prebuilt SmartPanel contains buttons for modifying records. This SmartPanel, which uses the TableIO link type, can serve as a TableIO-Source for a SmartDataViewer, SmartDataBrowser, or other SmartObject that is a TableIO-Target.
The master file src/adm2/pupdsav.w is called the Standard Update SmartPanel. You should use this SmartPanel to modify database records. It has six buttons and runs in one of two modes, or styles, that you can set in the SmartPanel’s instance properties dialog box:

- **Save mode** is the default mode. In Save mode, the updateable fields of the SmartPanel’s TableIO-Target are enabled at all times. The label on the first button is **Save**. A user can make changes to the current record and choose the **Save** button to write them back to the associated SmartDataObject.

- **Update mode** is the nondefault mode. Initially, the label on the first button is **Update**, and fields in the TableIO-Target SmartDataViewer or SmartDataObject are disabled. When the user locates a record to change, choosing the **Update** button enables the updateable fields, and the label on this button becomes **Save**. Choosing the **Save** button sends the changes to the SmartDataObject, disables the fields again, and resets the label to **Update**.

Choosing the buttons of the Standard Update SmartPanel in **Save** mode invokes the following methods, found in the super procedure src/adm2/datavis.p, as well as in the SmartDataBrowser- and SmartDataViewer-specific super procedures src/adm2/browser.p and src/adm2/viewer.p:

- **Save** invokes `updateRecord`
- **Add** invokes `addRecord`
- **Reset** invokes `resetRecord`
- **Copy** invokes `copyRecord`
- **Delete** invokes `deleteRecord`
- **Cancel** invokes `cancelRecord`

In Update mode, choosing the **Save**, **Add**, **Reset**, **Copy**, **Delete**, and **Cancel** buttons invoke the same methods as in Save mode. In addition, choosing the **Update** button invokes `enableFields`.

See the online help for more information on the functions of these procedures and functions.

The Standard Update SmartPanel has no direct role in managing database transactions. It merely invokes the procedures and functions named. All actual data access, locking, and transaction management are handled by the super procedures for the SmartDataObjects themselves.

**Commit SmartPanel**

The Commit Panel SmartObject includes two buttons: **Commit** and **Undo**. The **Commit** button PUBLISHes `commitTransaction` and the **Undo** button PUBLISHes `undoTransaction`.

The Commit SmartPanel, `pcommit.w`, defines the CommitTarget property and CommitTarget event. The panel subscribes to the RowObjectState in its target. The RowObjectState procedure takes either NoUpdates or RowUpdated for parameters. These states disable or enable the panel buttons.

When you use a Commit Panel, the SmartDataObject is made a Commit-Target, and the panel is a Commit-Source. The SmartDataObject sets the AutoCommit property to No if it has a Commit-Source.
The data.p super procedure includes the `commitTransaction` and `undoTransaction` event procedures. See the online help for more information on these procedures.

**Note:** Use the Commit SmartPanel only when your application must specifically allow users to batch multiple updates to be committed at the same time; for example, to allow a set of order-line records for an order to be added or updated together. In other cases, using the Commit Panel is unnecessary and can complicate the update process.

SmartDataObjects have an `AutoCommit` property that indicates whether, on update (Add, Save, Copy, or Delete), changes should be written back to the database as each individual record is saved (or deleted). By default, the `AutoCommit` property is set to YES. When this is the case, the following happens each time the Save or Delete button is pressed (or an equivalent `updateRecord` or `deleteRecord` operation occurs in some other way):

1. The before and after image of the single row that is being updated is written into an additional temp-table called RowObjUpd. For a discussion of this temp-table, see the “SmartDataObject query and update operations” section on page 7–3.

2. If the SmartDataObject is divided between client and AppServer, the row is passed back to the AppServer.

3. The change is made to the database.

However, if a SmartDataObject has a Commit-Source (normally the Commit SmartPanel), the `AutoCommit` property is automatically set to NO. In this case, each time the Save or Delete button is pressed, the relevant RowObject row is written to the RowObjUpd temp-table, but the changes are not written back to the database. To write the updates to the database, the application user presses the Commit button, which executes the `commitTransaction` procedure in the SmartDataObject. This procedure sends the RowObjUpd table back to the server, then opens a transaction block and makes all the modifications to the database in a single transaction. If errors occur, the updates continue (if possible), to identify all errors with the update, the transaction is undone, and the errors are returned to the client for correction. If no errors occur, all the updates are written to the database in a single transaction block.

Pressing the Undo button, which executes the `undoTransaction` procedure, has the effect of reversing the current Commit operation. This procedure cancels all updates that were not committed, empties the RowObjUpd temp-table, and returns the RowObject temp-table to its state before the first of these updates was made.
Customizing SmartPanels

You can customize the SmartPanel objects in several ways:

- **Instance properties** — The SmartPanels have an instance properties dialog box (source program `src/adm2/support/n-pane1d.w` for Navigation panels and `src/adm2/support/u-pane1d.w` for Update panels) that you can use to specify properties in a run-time instance of a SmartPanel, without modifying the master object itself. For all SmartPanels, you can choose to show a decorative border rectangle around the buttons. The default border is a rectangle with two edge-pixels (giving a chiseled look in Microsoft Windows). You can modify the number of edge-pixels or suppress the rectangle altogether by setting the value of the `EdgePixels` instance property to 0 (zero).

For the Navigation SmartPanel with icons, you can choose to make the VCR icon on the left the First button (the default) or the button on the right (for countries where text is read from right to left). This corresponds to the instance property `RightToLeft`.

For descriptions of the SmartPanel instance properties, see Chapter 2, “SmartObjects.”

- **Resizing SmartPanels** — You can resize a SmartPanel at design time by grabbing its resize handles and reshaping the panel. The `resizeObject` procedure in the super procedure `src/adm2/panel.p` (used by all SmartPanels) resizes and rearranges the buttons (in a column or in multiple rows if necessary) to fit the available space.

- **Creating custom SmartPanels** — You can create a custom SmartPanel by deleting one or more buttons from a SmartPanel and saving the result as a new SmartPanel. (Code in the SmartPanel master files verifies a button is present before enabling or disabling it.) For example, you can remove any buttons from the Navigation SmartPanels to create a smaller custom Navigation panel. You also can remove buttons from the Update SmartPanels.

  **Caution:** Do not remove from Update SmartPanels any buttons needed to complete update operations. You cannot remove the Save/Update button. If Add or Copy is present or a panel based on the Standard Update panel is in Update mode, do not remove the Cancel button.

You also can add buttons with user-specific functionality to a custom SmartPanel. See Chapter 6, “Advanced ADM Topics,” for more information on building custom SmartObjects.

Using SmartToolbars

A SmartToolbar is a SmartObject that provides a toolbar and/or menu interface to application commands that perform various actions; for example, performing navigation and transaction actions in the current object. You can contain a SmartToolbar in a SmartWindow but not in a SmartFrame or a SmartDialog, and you can put only one SmartToolbar instance in a given SmartWindow.

Progress provides the prebuilt master `toolmenu.w` for the SmartToolbar. You can modify this master to create a new SmartToolbar with specialized functions, so the ADM does not include a SmartToolbar template.

By default, a SmartToolbar instance includes both a toolbar and a menu. See the “Customizing SmartToolbars” section on page 4–14 for information on changing this.
The SmartToolbar provides the application user with the ability to perform the following groups of actions:

- **Navigation operations** — First, Prev, Next, Last. For information on these actions, see the “Navigation SmartPanels” section on page 4–10.

- **TableIO operations** — Either of the following:
  - TableIO Save operations — Add record, Copy record, Delete record, Save record, Reset, Cancel.
  - TableIO Update operations — Add record, Update record, Copy record, Delete record, Save record, Reset, Cancel.

  For information on these actions, see the “Update SmartPanel” section on page 4–10.

- **Commit operations** — Commit, Undo. For information on these actions, see the “Commit SmartPanel” section on page 4–11.

- **Filter operation** — For information on this action, see the “Filtering records” section on page 4–8.

By default, only the Navigation and TableIO Save operations are enabled. You can specify that the user can perform all or a subset of these actions. See the “Customizing SmartToolbars” section on page 4–14. Which action groups you select determine whether the SmartToolbar functions as a Navigation-Source or a TableIO-Source or both. The wizard prompts you with possible connections.

The rules and data for the actions accessible in the SmartToolbar are defined in an action class from which the SmartToolbar inherits and are totally independent of the SmartToolbar’s layout and structure. As a result, all SmartToolbar actions have the same text, image, and interface throughout the application regardless of the SmartToolbar layout.

### Customizing SmartToolbars

The simplest way to customize the SmartToolbar object is through its instance properties. The SmartToolbar has an instance properties dialog box (source program: src/adm2/support/toold.w) that you can use to specify properties in a run-time instance of a SmartPanel without modifying the master object itself:

- You can choose to show either a menu or a toolbar or both.

- You can choose to show a decorative border rectangle around the toolbar; the border includes dividers between icon groups.

- You can specify which actions the application user can perform from the SmartToolbar.

For descriptions of the SmartToolbar instance properties, see Chapter 2, “SmartObjects.”

More complex SmartToolbar modifications (for example, the order and structure of icons and menus) involve modifying the SmartToolbar object itself. For basic information, see the “Complex SmartToolbar modifications” section on page 2–42.
Transactions and record locking

The transaction logic built into the standard SmartObjects (in particular, the SmartDataObject and its supporting files data.i and the super procedure data.p) uses optimistic locking. No database records are locked when they are initially read and transferred into the RowObject temp-table. When an update is committed, the corresponding database records are re-read with an EXCLUSIVE-LOCK. If a lock conflict occurs, the commit is aborted and this information is reported to the user. In the case of an add or copy operation, the record CREATE for the new record is performed inside of this transaction block.

Next, by default, the SmartDataObject verifies that the database records were not modified by another user since they were first read. If you want to skip this check, set the checkCurrentModified property in the SmartDataObject to NO in the instance properties dialog box for the SmartDataObject when assembling an application. Alternately, you can set it to NO in the initialization code for the SmartDataObject.

Finally, only those fields actually modified by the application are ASSIGNED back to the database records. Because the code uses the ChangedFields field in the update temp-table (RowObjUpd) to keep track of which fields were changed, it is necessary to update this field by hand if user-written application code is used; for example, if the RowObjectValidate procedure in a SmartDataObject modified fields beyond those changed in a SmartDataViewer or SmartDataBrowser.

When the update is complete, the database records are re-read to capture any field values that might have been assigned by the CREATE, WRITE, or ASSIGN triggers, and those values are passed back to the client objects for display. A transaction is never held open while data values are examined or entered.
For a given application screen to work smoothly, the SmartObjects on the screen must continually interact. In particular, they must know how to communicate, how to get and set ADM properties, and how to pass data. As a programmer, you need to know how the ADM manages these different forms of interaction.

This chapter contains the following sections:

- General structure of a SmartObject
- RUN protocol
- ADM properties

The protocols provide a reliable and flexible way for independent encapsulated modules (SmartObjects) to communicate and pass data.
General structure of a SmartObject

This section discusses the following topics:

- Directory and file structure
- Support include files
- Super procedures
- Property and prototype include files

Directory and file structure

The ADM directory structure includes:

- Primary include files for SmartObjects and their super procedures, located in
  %DLC%/src/adm2 (Windows) or $DLC/src/adm2 (UNIX)

- Templates, located in %DLC%/src/adm2/template (Windows) or
  $DLC/src/adm2/template (UNIX)

- Support procedures, located in %DLC%/src/adm2/support and compiled into
  %DLC%/gui/adm2/support (Windows), or located in $DLC/src/adm2/support and
  compiled into $DLC/gui/adm2/support (UNIX)

- Super procedures and prebuilt objects such as the SmartPanels and SmartFolders, also
  located in %DLC%/src/adm2 and compiled into %DLC%/gui/adm2 (Windows), or located in
  $DLC/src/adm2/support and compiled into $DLC/gui/adm2/support (UNIX)

Many SmartObjects are constructed from template (.w) files. These templates contain a
minimum of executable code to minimize maintenance problems if general changes need to be
made to a class of objects after many specific masters are already built from a template. Some
SmartObjects do not have template files; they have only master files that can be customized at
design time.

Support include files

As noted in Chapter 1, “Overview,” a SmartObject template generally includes a single support
include file: the primary include file for the class on which the SmartObject is based. This
include file has the same name as the template (but a different extension). For example, the
SmartPointer template viewer.w includes the primary include file viewer.i. A primary
include file such as viewer.i does not directly contain most of the code needed to support the
SmartObject. It contains only those program elements required to compile a master built from
that template:

- Any variable or other definitions needed by all masters of that type

- A reference to the next support include file up the chain (for example, a SmartDataViewer
  is a visual object that displays data, so it includes datavis.i; if there is no other
  intermediate object above this type, it includes smart.i), and its own property include file,
  which defines those properties stored in the ADMProps temp-table for the object
• Initialization code in the main block for this object type’s properties and any other generic object startup code

• Any routine that requires compile-time resolution of preprocessors or for some other reason must be part of the master’s compilation unit (.r file)

Super procedures

All support code for an object type, other than the code in its primary include file, is in a separately compiled super procedure built as a structured .p file. For example, the super procedure for SmartDataViewers is viewer.p.

During startup, the .p file is run and established as a super procedure for each of its SmartObjects running in that session. Its routines use TARGET-PROCEDURE to get the procedure handle of the appropriate SmartObject and can then get property values from that procedure to get handles or other data needed to act on behalf of the SmartObject. These SmartObjects can run any internal procedure or function implemented in one of the object’s super procedures as if it were implemented in the SmartObject itself. This allows these separately compiled procedure objects to do the following:

• Support standard behavior for all SmartObjects of a given type running in a Progress session.

• Enable the customization or overriding of these procedures and functions in individual SmartObjects.

For an overview of super procedures, see OpenEdge Getting Started: ABL Essentials. For descriptions of super procedures and TARGET-PROCEDURE, see OpenEdge Development: ABL Reference.

Figure 5–1 illustrates this relationship.

Figure 5–1: Super procedures and master interactions

The smart.p code uses TARGET-PROCEDURE to reference the two masters. For master1.w, the xyz procedure in the smart.p super procedure is run because master1.w does not have its own xyz procedure. For master2.w, its own xyz procedure is run and then, because it executes RUN SUPER, the xyz procedure in smart.p also is run.
As this example illustrates, using super procedures to provide most of the support code for a SmartObject provides these advantages:

- Encapsulation of routines into a super procedure where they are accessible from all SmartObjects associated with that super procedure.
- The ability to use the same procedure or function name in multiple layers when defining the behavior for a particular SmartObject.

Each super procedure must define a `get propname` function for each object property that can be retrieved from objects of that type and a `set propname` function for each settable property. For more information on these properties, see the “Get and set functions for object properties” section on page 5–16.

Super procedures are not SmartObjects in themselves and are not directly seen or manipulated in the AppBuilder during application assembly. They do not have SmartLinks, nor do they have properties of their own.

The super procedures for SmartObjects are designed to be shared and stateless. This means a given super procedure such as `viewer.p` is run only once in a session. No matter how many SmartDataViewers there are, each SmartDataViewer designates the same instance of `viewer.p` as its super procedure. This means whenever a routine is invoked in `viewer.p` as a super procedure, it queries whatever properties of its TARGET-PROCEDURE are needed for it to act. The next routine invoked in `viewer.p` is not assumed to come from the same TARGET-PROCEDURE.

### Property and prototype include files

The property include file for a SmartObject defines its basic properties, references the its prototype file, and, if necessary, starts its super procedure. It also can define instance properties and the instance properties dialog box and may reference certain custom class files as needed. For more information on properties, see the “ADM properties” section on page 5–11.

### Prototype include files

The prototype include file for a SmartObject defines prototypes for all internal entries (functions and internal procedures) of the super procedure for the class.
SmartObject file relationships

Within the overall SmartObject class hierarchy, the class files described in the “General structure of a SmartObject” section on page 5–2 relate to other files of the same type in certain very specific ways. These relationships establish what characteristics and properties a specific SmartObject type inherits. Figure 5–2, which is based on the SmartDataViewer, illustrates these relationships.

![Figure 5–2: Include file structure for SmartDataViewer](image)

In Figure 5–2, note the following:

- The `smrtprop.i` file (lower left) defines an ADMProps temp-table that in turn defines the basic properties that apply to all SmartObjects. Each include file above the `smrtprop.i` file adds properties specific to the class with which it is associated. For details, see the “ADMProps temp-table and object properties” section on page 5–11.

- The `viewer.w` file includes only `viewer.i`.

- The `viewer.i` file includes `datavis.i`, because a SmartDataViewer is a data visualization object. It also includes `viewprop.i`, which includes the prototype include file `viewprto.i`.

- The `datavis.i` file includes `visual.i`, because a SmartDataViewer is a visual object. It also includes `dvisprop.i`, which includes `dvisprto.i`.

- The `visual.i` file includes `smart.i`, because a visual object is a SmartObject. The `visual.i` file also includes `visprop.i`, which includes `visprto.i`.

- The `smart.i` file includes only `smrtprop.i` (which includes `smrtprto.i`).
When SmartObject code is preprocessed, the preprocessors limit the full expansion of the include files to exactly one branch of the class hierarchy, to avoid duplication of the SmartObject code.

Each SmartObject type uses a different part of the class hierarchy (different branches), but the tree for any SmartObject type always ends with smart.i.

Figure 5–3 expands on Figure 5–2 to illustrate how super procedures are started.

![Super procedure schematic for SmartDataViewer](image)

**Figure 5–3:** Super procedure schematic for SmartDataViewer

In Figure 5–3, note the following:

- The smart.i file starts smart.p (if it is not already running) and adds it as a super procedure for the SmartDataViewer. The smart.p file should be the top-level super procedure for all SmartObjects.
- The visual.i file starts visual.p, if it is not already running, and adds it as a super procedure for the SmartDataViewer.
- The datavis.i file then starts datavis.p (if it is not already running) and adds it as a super procedure for the SmartDataViewer.
- The viewer.i file starts viewer.p (if it is not already running) and adds it as a super procedure.

**Note:** The object’s own super procedure always must be added last. This is because super procedures are searched in last-in, first-out (LIFO) order, so the interpreter searches in viewer.p first for needed routines and in smart.p last. This ensures that more general behavior is executed last and more object-specific behavior is executed first.
RUN protocol

All SmartObjects can invoke the behavior of other SmartObjects in themselves by running internal procedures and invoking functions and by publishing events to which other objects subscribe.

RUN statement

Many aspects of SmartObject behavior are invoked by running internal procedures and invoking functions. Much of this behavior is implemented in super procedures that support the SmartObjects. The behavior in super procedures can be localized, and the Progress interpreter will locate and execute the correct behavior. This is possible because super procedures are initialized and added to each SmartObject as each SmartObject starts up.

For example, an application can use the addLink procedure to add a link from one SmartObject to another during execution:

```
RUN addLink ( hSourceProc, 'navigation', hTargetProc ).
```

In this code, hSourceProc is the HANDLE of the source procedure and hTargetProc is the HANDLE of the target procedure.

The RUN statement for addLink is written as if addLink were an internal procedure within the calling procedure. In fact, it is contained in the super procedure smart.p. The super procedure mechanism makes the contents of smart.p available to every SmartObject. Standard behavior of this kind includes the following procedures (any many others):

- **initializeObject** — Runs when each object is started, to initialize that object
- **destroyObject** — Runs when an object is terminated, to destroy that object
- **addMessage** — Adds an error message to a message log

Similarly, the contents of other SmartObject super procedures are available to SmartObjects that require their support. For example, all visible objects use the super procedure visual.p and all SmartObjects that can contain other SmartObjects use containr.p. (See the AppBuilder online help for a detailed list of super procedures and their contents.) Using this architecture locates the majority of SmartObject behavior in a set of independently built procedures that can serve all SmartObjects that are running in a Progress session. This helps to organize behavior into reusable classes and keeps the size of individual SmartObjects to a minimum.

Just as a SmartObject can RUN procedures implemented in its super procedures, a SmartObject can invoke behavior in another SmartObject simply by knowing its procedure handle and running internal procedures or functions in that SmartObject. For example, one SmartObject can destroy another by executing the following statement:

```
RUN destroyObject IN hOtherObject.
```
Localizing standard SmartObject behavior

Another advantage of using super procedures to implement classes of behavior is that you can localize each of the internal procedures and functions that compose a super procedure in any individual SmartObject by writing a procedure or function of the same name. You can do this either to replace the standard behavior or to augment the standard behavior as required by the SmartObject. The Section Editor supports this functionality by identifying the internal procedures and functions that you can override based on the type of object you are creating.

To create a new displayFields procedure for a SmartDataViewer in the AppBuilder:

1. In the Object Palette, right-click on the SmartDataViewer button.
2. Choose New SmartDataViewer.
3. Complete the steps in the SmartDataViewer wizard.
4. On the AppBuilder toolbar, choose the Edit code button.
5. Change the Section to Procedures.
6. Click the New button. The New Procedure dialog box appears.
7. Choose the Override Type radio button. The Name fill-in field becomes a combo box:
8. Choose the `displayFields` procedure and click OK. The Section Editor generates a skeleton of the local procedure or function. This contains the necessary parameter definitions and, by default, invokes the standard behavior by executing the `RUN SUPER` or `RUN SUPER()` statement:
9. You can change this code as required for your application. For example, to change the color of the fields in the SmartDataViewer, change the code as shown:

```plaintext
Note: This code augments the standard displayFields behavior and affects all instances created from this master.
```
ADM properties

A basic requirement of SmartObjects is that each SmartObject can expose its properties to other objects, either Progress procedures or non-Progress application objects communicating through the Open4GL interface. Within a Progress session, it is not practical to use \texttt{SHARED} variables or buffers to share values between SmartObjects. SmartObjects are run as persistent procedures, which means they run as peers of one another and have no fixed execution hierarchy; therefore, the \texttt{SHARED} mechanism does not work well. Also, there is no way to use \texttt{SHARED} values between a client and non-Progress application components.

The ADM supports properties through these mechanisms:

- The ADMProps temp-table for object property definitions and storage
- Property-specific \texttt{get} and \texttt{set} functions
- The \texttt{setUserProperty} and \texttt{getUserProperty} functions for dynamic properties

The first two mechanisms, which establish static properties, share a common syntax and can be used interchangeably. They return property values with the native data type, so their use is recommended when you need properties in custom super procedures that will become part of an object type. Dynamic properties, in contrast, return property values as strings that might need conversion to other data types, so this mechanism is most appropriate for properties that will be used only once in your application. The rest of this chapter describes these mechanisms, as well as related topics.

Some special property considerations apply when you are building new SmartObject classes and or extending existing classes. For details, see Chapter 8, “Developing ADM Extensions.”

ADMProps temp-table and object properties

One mechanism that supports getting and setting many SmartObject properties is the \textit{ADMProps temp-table}. An ADMProps temp-table is defined for each object; it defines all standard object properties for that object type.

Each SmartObject super procedure uses a Progress include file that defines its own set of properties, and each SmartObject type that uses that super procedure also includes the same list. For example, the super procedure \texttt{smart.p} defines functions and procedures that all SmartObjects use and is at the top of the hierarchy of super procedures. This procedure includes \texttt{smrtprop.i}, which defines the basic properties that apply to all SmartObjects. It defines a preprocessor constant for each basic property and also puts a \texttt{FIELD} definition for that property in the ADMProps temp-table. In addition, all SmartObjects use \texttt{smart.p} as a super procedure and include the supporting include file \texttt{smart.i}. The \texttt{smart.i} include file also includes \texttt{smrtprop.i}, as well as adding \texttt{smart.p} as a super procedure.
The smrtpop. file defines these properties:

```plaintext
/* These preprocessors let the get and set methods know at compile time which
 property values are located in the temp-table and which must be accessed
 through the property functions. */

&GLOB xpObjectVersion
&GLOB xpObjectType
&GLOB xpContainerType
&GLOB xpPropertyDialog
&GLOB xpQueryObject
&GLOB xpContainerHandle
&GLOB xpInstanceProperties
&GLOB xpSupportedLinks
&GLOB xpContainerHidden
&GLOB xpObjectInitialized
&GLOB xpObjectHidden
&GLOB xpContainerSource
&GLOB xpContainerSourceEvents
&GLOB xpDataSource
&GLOB xpDataSourceEvents
&GLOB xpTranslatableProperties
&GLOB xpObjectPage
&GLOB xpDBAware
```
If a preprocessor has a name of the form `xp`propname, you can access the associated property value directly from its property temp-table FIELD. The `{get}` and `{set}` pseudo-syntax uses this mechanism to optimize references to properties, primarily in super procedures. If it is necessary for either `get` or `set` to invoke the corresponding property function (because of some other action it performs), the preprocessor constant is not defined for that property, and references always go through the `get` and `set` functions. For more information, see the "{get} and {set} pseudo-syntax for object properties" section on page 5–17.
Now consider a SmartObject defined as a subclass of smart.p. For example, a SmartDataObject adds smart.p as its first super procedure and then adds query.p and data.p as the super procedures that define behavior specific to SmartDataObjects. The data.p procedure includes the property include file dataprop.i, which appends more basic values to the list already started by smrtprop.i and continued by qryprop.i. Each property include file adds FIELD definitions to the ADMProps temp-table definition.

The dataprop.i file defines these additional basic properties:

```plaintext
-dataprop.i

/* Preprocessor definitions which tell at compile time which
 properties can be retrieved directly from the property temp-table. */
&GLOB xpEnabledTables
&GLOB xpAutoCommit
&GLOB xpDataHandle
&GLOB xpCurrentRowid
&GLOB xpAppService
&GLOB xpASUsePrompt
&GLOB xpASInfo
&GLOB xpASHandle
&GLOB xpASDivision
&GLOB xpUpdateSource
&GLOB xpCommitSource
&GLOB xpCommitSourceEvents
&GLOB xpCommitTarget
&GLOB xpCommitTargetEvents
&GLOB xpDataModified
&GLOB xpRowsToBatch
&GLOB xpCheckCurrentChanged

{src/adm2/qryprop.i}
```
&IF 
 &ADMSuper)"&U = "&U &THEN
    ghADMProps:ADD-NEW-FIELD('RowObject':U, 'HANDLE':U, 0, ?, ?).
    ghADMProps:ADD-NEW-FIELD('RowObjUpd':U, 'HANDLE':U, 0, ?, ?).
    ghADMProps:ADD-NEW-FIELD('FirstRowNum':U, 'INT':U, 0, ?, ?).
    ghADMProps:ADD-NEW-FIELD('LastRowNum':U, 'INT':U, 0, ?, ?).
    ghADMProps:ADD-NEW-FIELD('DataHandle':U, 'HANDLE':U).
    ghADMProps:ADD-NEW-FIELD('ASUsePrompt':U, 'LOGICAL':U, 0, ?, ?).
    ghADMProps:ADD-NEW-FIELD('ASDivision':U, 'CHAR':U, 0, ?, ''&U).
    ghADMProps:ADD-NEW-FIELD('CommitTarget':U, 'CHAR':U, 0, ?, ''&U).
    ghADMProps:ADD-NEW-FIELD('DataModified':U, 'LOGICAL':U, 0, ?, no).
    ghADMProps:ADD-NEW-FIELD('RowsToBatch':U, 'INT':U, 0, ?, 200). /* Rows per
AppServer xfer */
    ghADMProps:ADD-NEW-FIELD('CheckCurrentChanged':U, 'LOGICAL':U, 0, ?, yes).
    ghADMProps:ADD-NEW-FIELD('NextBatchRow':U, 'CHAR':U, 0, ?, ?). /* Next row to return in a batch */
    ghADMProps:ADD-NEW-FIELD('FirstResultRow':U, 'CHAR':U, 0, ?, ?).
    ghADMProps:ADD-NEW-FIELD('LastResultRow':U, 'CHAR':U, 0, ?, ?).
    ghADMProps:ADD-NEW-FIELD('RowObjectState':U, 'CHAR':U, 0, ?, 'NoUpdates':U).
    ghADMProps:ADD-NEW-FIELD('ServerOperatingMode':U, 'CHAR':U, 0, ?, ?).
    ghADMProps:ADD-NEW-FIELD('StatelessSavedProperties':U, 'CHAR':U, 0, ?, 'CheckCurrentChanged,RowObjectState,LastResultRow,FirstResultRow,'
QueryRowIdent':U).
&ENDIF
As this example shows, each SmartObject type appends more basic properties to the end of the existing list. The process of appending basic properties to the existing list can be nested to any number of levels of object definition. In this case, both the data.p super procedure for SmartDataObjects, and individual SmartObjects themselves, include dataprop.i. The dataprop.i include file in turn includes qryprop.i, which includes smrtprop.i. Basic properties are then initialized in each corresponding object include file (such as smart.i for all SmartObjects and data.i for SmartDataObjects). This initialization both allocates FIELDS for each basic property and, where appropriate, assigns specific initial values. Because these preprocessor values are included in both the super procedures and in master objects of the type, both have access to them. This allows a super procedure to retrieve a property value directly from the ADMProps temp-table of its TARGET-PROCEDURE, which is extremely fast, instead of using function calls.

**Get and set functions for object properties**

A second mechanism that supports retrieving and setting property values is a naming convention that defines two groups of Progress functions called *get* and *set* functions. These functions are available for all properties that can be read or written by other objects:

- The *get* functions, which have the format `get propname`, retrieve the values of properties. They take no input parameters and each returns the associated property value with the native data type. If a property is write only, no *get* function is available.

- The *set* functions, which have the format `set propname`, set property values. They take as their only INPUT parameter the value of the associated property (which can be of any Progress data type) and each returns the type LOGICAL: TRUE or FALSE depending on whether the *set* operation was successful. If a property is read only, no *set* function is available.

If no special processing is needed to get or set a property, the *get* and *set* functions simply use the special include file syntax defined in the “ADMProps temp-table and object properties” section on page 5–11 to access the appropriate field in the properties temp-table record for the SmartObject. For example:

```progress
FUNCTION getMyProp RETURNS CHARACTER:
   DEFINE VARIABLE cProp AS CHARACTER NO-UNDO.
   {get MyProp cProp}.
   RETURN cProp.
END.

FUNCTION setMyProp RETURNS LOGICAL (pcMyValue AS CHARACTER):
   {set MyProp pcMyValue}.
   RETURN TRUE.
END.
```
The get and set functions are not restricted to such simple operations; they can perform whatever actions you need to set and get values, such as verifying the validity of values. For example, you could use the following pair of functions to get and set the value of the BackGround Color property (BGColor) of the default Progress Frame in an object such as a SmartDataViewer:

```plaintext
FUNCTION getBGColor RETURNS INTEGER:
  RETURN {&FRAME-NAME}:BGCOLOR.
END.

FUNCTION setBGColor RETURNS LOGICAL (piBGC AS INTEGER):
  IF piBGC <= SomeMaxBGColorValue THEN DO:
    {&FRAME-NAME}:BGCOLOR = piBGC.
    RETURN TRUE.
  END.
  ELSE RETURN FALSE.
END.
```

### {get} and {set} pseudo-syntax for object properties

Progress supplies a {get} and {set} pseudo-syntax in the ADM that can be used to access property values. It is implemented using Progress include files named get and set, located in the gui and tty directories. This pseudo-syntax makes accessing property values as simple as possible, and it provides transparency to whether a property can be retrieved or set directly, or must be accessed through its get and set functions. (The normal .i extension is removed to make these references look as much like standard syntax as possible.)

#### Get include file logic

The get include files use the following logic:

```
{get propname target-variable [ object-handle ]}.
```

The get syntax first checks to see whether the preprocessor xppropname exists. If it does, the get syntax returns the value in the TARGET-PROCEDURE’s ADMProps temp-table into `target-variable`. If the optional `object-handle` is specified, the get syntax searches that procedure’s ADMProps temp-table instead of the TARGET-PROCEDURE.

If there is no `xppropname` preprocessor constant, the `getpropname` function is executed in the TARGET-PROCEDURE (or in the `object-handle`, if specified).

This syntax is appropriate as an optimization for super procedures. In your application code, you should invoke the real functions.

#### Set include file logic

The set include files operate similarly, as follows:

```
{set propname value [ object-handle ]}.
```
If \texttt{xp\_propname} is defined, that entry in the \texttt{TARGET-PROCEDURE}'s or the \texttt{object-handle}'s ADMProps temp-table is set to \texttt{value}. If \texttt{xp\_propname} is not defined, the \texttt{setpropname} function is executed and passed the \texttt{value} (in its native data type) as an input parameter. The function returns TRUE or FALSE, depending on whether the operation was successful.

The following example illustrates interpreting the \{get\} logic:

\[
\begin{align*}
\text{\{get DataColumns cColumns\}.}
\end{align*}
\]

How this translates depends on whether \texttt{DataColumns} has an xp preprocessor defined for it. If it does, the statement translates into the actual 4GL code:

\[
\begin{align*}
\text{ASSIGN} \\
\text{ghProp} &= \text{WIDGET-HANDLE(ENTRY(1, TARGET-PROCEDURE:ADM-DATA, CHR(1)))} \\
\text{ghProp} &= \text{ghProp:BUFFER-FIELD('DataColumns')} \\
\text{cColumns} &= \text{ghProp:BUFFER-VALUE}.
\end{align*}
\]

This retrieves the handle of the ADMProps temp-table buffer for the \texttt{TARGET-PROCEDURE} (stored in the procedure’s \texttt{ADM-DATA} attribute) and then retrieves the field value using the dynamic \texttt{BUFFER-FIELD} attribute.

If the preprocessor is not defined for \texttt{DataColumns}, the statement translates to the following:

\[
\begin{align*}
\text{cColumns} &= \text{DYNAMIC-FUNCTION("getDataColumns":U IN TARGET-PROCEDURE).}
\end{align*}
\]

The \texttt{ghProp} variable is defined in \texttt{smart.i} for use by these include files. The buffer handle of the ADMProps property temp-table record is stored in the \texttt{ADM-DATA} procedure property of each SmartObject, so it can be located by these include files. These include files resolve during compilation to a single executable Progress statement, making them extremely fast.

\textbf{Using the get and set include files in other Progress code}

You can use the \texttt{get} and \texttt{set} include files from any Progress code. When you use them either from a SmartObject of a given type or one of its super procedures, the syntax locates and uses the basic properties in the ADMProps property temp-table. If you use the syntax in other Progress application code (for example, in an object of a different type that does not share that property), it will not have a value for the preprocessor index and so will always use the \texttt{get} and \texttt{set} property functions.

The \texttt{get} and \texttt{set} include files are not actual 4GL syntax and represent a fairly small optimization, so you should restrict their use to super procedures and use the standard \texttt{get} and \texttt{set} function invocations in other Progress application code. Further, any non-Progress code that uses the Open 4GL interfaces will be able to access the property only by invoking the appropriate \texttt{get} or \texttt{set} functions directly. For this reason, you should always define \texttt{getpropname} and \texttt{setpropname} functions for each basic property intended to be read and written from outside the object’s class. These property functions can simply access the ADMProps value directly.
Special properties

In some cases, it is desirable to store a property value in the ADMProps temp-table, to allow direct access to it from its super procedure and elsewhere, but also to have an action of some type take place when the value is set. For example, the QueryPosition property holds information about the cursor position of a query (FirstRecord, LastRecord, and so on). Whenever its value is set, a message must be also published so other objects can react. In cases of this type, you should not define an \texttt{xpclassname} preprocessor constant, so the get and set functions that have the additional behavior will run instead of getting and setting the value directly in the ADMProps temp-table.

Instance properties

Each object type has certain properties that are appropriate to set as part of the object’s initialization and can be assigned values when SmartWindows and other containers using that object are being assembled. For example, the properties HideOnInit, DisableOnInit, and ObjectLayout can be set for any visual object to indicate whether this particular instance of the object should be hidden when it is first realized or disabled when it is first realized, or which of multiple visual layout names should be used for the object. These properties are referred to as \textit{instance properties}. These can be differentiated from properties such as ObjectInitialized, ObjectHidden, QueryPosition, and many others that are set during the course of application execution but have no meaningful initial value that would specialize how that object is used in a particular case.

For each object include file, there is defined a list of instance properties using the xcInstanceProperties preprocessor value. These properties can be passed down to subclasses of a class. For example, all visual objects (including SmartDataViewers, SmartPanels, etc.) have the three Instance Properties defined in \texttt{visual.i}, and can append more to that list.

These properties are supported by instance properties dialog procedures for each object type. Any class can define an instance properties dialog procedure by defining the preprocessor \texttt{ADM-PROPERTY-DIALOG} to be the name of the procedure file that contains the instance properties dialog box. This initializes the PropertyDialog property at object startup time. When the object is dropped into a container at application assembly time, selecting the Instance Properties choice from the object’s pop-up menu runs this dialog box procedure and sets these property values in that instance of the object. The values also are specified in AppBuilder-generated code in \texttt{adm-create-objects}, in calls to the \texttt{constructObject} procedure.

Translatable properties

In the ADM, Progress maintains a list of properties that are translatable; that is, those properties whose literal values should not be tagged with \texttt{":U"} when specified in \texttt{adm-create-objects} code generated by the AppBuilder. Most properties are not translatable, but a few should be; for example, the FolderLabels property for a Folder. The list of these properties is stored in the TranslatableProperties property.
Functions for accessing properties

The following property functions are useful for accessing SmartObject properties from application code:

- The `assignLinkProperty` function sets a property value in one or more objects at the other end of the specified link. It is defined as follows:

  ```
  assignLinkProperty RETURNS LOGICAL
  (pcLink AS CHARACTER, pcPropName AS CHARACTER, pcPropValue AS CHARACTER):
  ```

  For example, the following code in a SmartDataObject uses `assignLinkProperty` to set the `DataModified` property to `yes` in all its Data-Targets:

  ```
  DYNAMIC-FUNCTION('assignLinkProperty':U,
    INPUT "DATA-TARGET":U,
    INPUT "DataModified":U,
    INPUT "yes":U).
  ```

- The `linkProperty` function returns the value of the specified property in the single object at the other end of the specified link. It is defined as follows:

  ```
  linkProperty RETURNS CHARACTER
  (pcLink AS CHARACTER, pcPropName AS CHARACTER):
  ```

  For example, the following code in a visual object (a SmartDataViewer or a SmartDataBrowser) uses `linkProperty` to get the current page of the object’s container:

  ```
  iCurrentPage = DYNAMIC-FUNCTION('linkProperty':U,
  ```

- The `propertyType` function returns the data type of the specified property. It is defined as follows:

  ```
  propertyType RETURNS CHARACTER
  (pcPropName AS CHARACTER):
  ```

  For example, the following code in a SmartDataObject gets the data type of its `DataQueryBrowsed` property:

  ```
  cDataType = DYNAMIC-FUNCTION('propertyType':U IN h_dcustomer,
    INPUT "DataQueryBrowsed":U).
  ```

  This function returns data types only for properties for which a `setpropname` function is defined.
Dynamic properties

Dynamic properties are object properties that are defined on an as-needed basis. Two functions, setUserProperty and getUserProperty, support dynamic properties in SmartObjects. They allow you to define dynamic properties without using either the set and get functions or a temp-table FIELD.

The setUserProperty function sets a named property. It is defined in an application as follows:

```plaintext
setUserProperty RETURNS LOGICAL
   ( pcPropName AS CHARACTER, pcPropValue AS CHARACTER ) :
```

The getUserProperty function returns the value of a specified property that was previously set using the setUserProperty function. The getUserProperty function is defined as follows:

```plaintext
getUserProperty RETURNS CHARACTER
   ( pcPropName AS CHARACTER ) :
```

Because the property values managed by getUserProperty and setUserProperty are stored as part of the ADM-DATA procedure attribute for the SmartObject, they always are stored and returned as CHARACTER strings.

This example illustrates how to set up a dynamic property called NewLimit. It assumes that you have a SmartDataViewer, and a SmartWindow that contains a button that sets a NewLimit attribute in the SmartDataViewer.

To set up the NewLimit dynamic property:

1. Add the following to the SmartWindow button trigger code, to set the property:

   ```plaintext
   DYNAMIC-FUNCTION('setUserProperty':U IN h_v-customer,
   ```

2. Add the following code to the SmartDataViewer, to get the property:

   ```plaintext
   DECIMAL(DYNAMIC-FUNCTION('getUserProperty':U,
                    INPUT "NewLimit").
   ```
Advanced ADM Topics

This chapter discusses several advanced ADM topics:

- Parameterizing SmartObject instances as variables
- Writing local event procedures and functions
- Monitoring SmartObject applications
- AppBuilder requirements for SmartObjects
- Using GroupAssign SmartLinks
Parameterizing SmartObject instances as variables

When you insert a SmartObject instance into a SmartContainer, you can specify a variable name for it by accessing its property sheet and activating the **Parameterize As Variable** toggle box. When you do this, you are prompted for a variable name to associate with the SmartObject. You can supply either a local **CHARACTER** variable or a parameter that you define in the SmartContainer that you are building; this variable or parameter must be assigned the filename of the SmartObject procedure file to be run in place of the SmartObject. The SmartContainer uses the specified variable name at run time to get the name of the master procedure file. The SmartContainer uses this filename to instantiate the SmartObject instance; by programmatically changing the value of the variable, you can have it specify different SmartObject master filenames. When the SmartContainer instantiates the SmartObject, it uses the setting of the variable to choose a particular master procedure file.

With this technique, you can use the same SmartContainer to create many different application screens. Keep in mind that inside the SmartContainer you must define the variable and set it appropriately.

The following examples illustrate how to use Parameterize As Variable. Each example uses **placeholder** SmartObjects: SmartObjects that you place in a SmartWindow to allow linking them at design time in the AppBuilder. Specifically, each example uses a placeholder SmartDataObject and SmartDataViewer for the customer table, which you can replace at run time with any SmartDataObject and SmartDataViewer that can be linked with Data and Update SmartLinks. To do this, you set the variables to the object names for the replacement SmartDataObject and SmartDataViewer.

### Using Parameterize As Variable with a Variable.

To use Parameterize As Variable with a variable defined in a SmartContainer:

1. Create a SmartWindow.
2. Drop a placeholder SmartDataObject and a placeholder SmartDataViewer into the SmartWindow and link them with Data and Update SmartLinks.
3. Select the SmartDataObject and open its property sheet (**Tool** → **Property Sheet**).
4. In the **Variable** fill-in field that appears, check the **Parameterize As Variable** option and enter `cSDONAME`.
5. Select the SmartDataViewer and open its property sheet (**Tool** → **Property Sheet**).
6. In the **Variable** fill-in field that appears, check the **Parameterize As Variable** option and enter `cSDVName`.
7. Save the SmartWindow to a filename of your choice.
8. In the **Definitions** section of the saved SmartWindow file, define variables for the SmartDataObject and SmartDataViewer that the SmartWindow will contain:

```
DEFINE VARIABLE cSDONAME AS CHARACTER NO-UNDO.
DEFINE VARIABLE cSDVName AS CHARACTER NO-UNDO.
```
9. In the SmartWindow’s main block, assign to these variables values that specify which SmartDataObject and SmartDataViewer are displayed in this SmartWindow at run time:

```
ASSIGN
  cSDOName = "dorder.w"
  cSDVName = "vorder.w".
```

You must assign values to these variables in the main block, so they are set before the AppBuilder runs `adm-create-objects` to construct the SmartObjects they specify.

**Using Parameterize As Variable with Input Parameters.**

To use Parameterize As Variable with input parameters defined in a SmartContainer:

1. Create a SmartWindow.
2. Drop a placeholder SmartDataObject and a placeholder SmartDataViewer into the SmartWindow and link them with Data and Update SmartLinks.
3. Select the SmartDataObject and open its property sheet (Tool→Property Sheet).
4. In the Variable fill-in field that appears, check the Parameterize As Variable option and enter `pcSDOName`.
5. Select the SmartDataViewer and open its property sheet (Tool→Property Sheet).
6. In the Variable fill-in field that appears, check the Parameterize As Variable option and enter `pcSDVName`.
7. Save the SmartWindow to a filename of your choice.
8. In the Definitions section of the saved SmartWindow file, define variables for the SmartDataObject and SmartDataViewer that the SmartWindow will contain:

```
DEFINE INPUT PARAMETER pcSDOName AS CHARACTER.
DEFINE INPUT PARAMETER pcSDVName AS CHARACTER.
```

9. When the SmartWindow is run, pass the SmartDataObject and SmartDataViewer names to the SmartWindow as parameters:

```
RUN wparamasvar.w (INPUT "dorder.w", INPUT "vorder.w").
```
Writing local event procedures and functions

An event procedure is a procedure that executes when a particular event occurs. This is sometimes accomplished through the PUBLISH/SUBSCRIBE functionality, as follows:

1. SmartObjects that need to know about an event SUBSCRIBE to the event. (The addLink procedure does this at startup.)
2. When an event occurs, a SmartObject PUBLISHes the event.
3. This action notifies all subscribing SmartObjects that the event occurred.

Sometimes, however, this happens simply when a SmartObject RUNs an internal procedure or invokes a user-defined function. In either case, the procedure or function being executed normally is implemented in a separate super procedure. See the “Super procedures” section on page 5–3.

The super procedure functionality allows you to write a local version of an ADM internal procedure or function in a SmartObject master procedure file that augments or overrides default ADM events. When the ADM runs an internal procedure or function, Progress 4GL looks at the SmartObject and its super procedures, starting with a local version if there is one, and runs the first one it finds. If this super procedure performs a RUN SUPER or SUPER( ), the 4GL looks further up the super procedure chain to find the next version to run. A local version of an ADM procedure can execute code before, after, or instead of the more standard behavior of the internal procedures and functions in ADM super procedures.

Writing a local event procedure

To write a local event procedure in the AppBuilder:

1. Invoke the Code Section Editor, go to the Procedures code section, and choose the New button. The New Procedure dialog box appears:
2. Activate the **Override** radio button. The AppBuilder populates the **Name** combo box list with the ADM event procedures available in the super procedures for the SmartObjects:

![New Procedure dialog](image)

3. Select the ADM event procedure for which you want to write a local version, then choose **OK**. The AppBuilder returns you to the Code Section Editor and creates a procedure with the same name. If this procedure already exists, the AppBuilder displays the existing code section.

The default code for a local event procedure always executes the standard version of the procedure, located in one of the SmartObject's super procedures. For example, for `addLink`, the following line appears in the local version:

```
RUN SUPER ( INPUT phSource, INPUT pcLink, INPUT phTarget).
```

This allows you to put code before or after the standard event procedure call or not run the code at all (by commenting it out).

**Note:** Do not suppress the standard event procedure by removing the `RUN SUPER` statement unless you are sure that you understand what it does and that you replace it correctly.

### Writing a local event function

With one exception, you write a local event function in the same way you write a local event procedure, as described in the “Writing a local event procedure” section on page 6–4. The exception is that in Step 1, you go to the list of functions in the Code Section Editor instead of the **Procedures** code section.
Monitoring SmartObject applications

ADM provides several OpenEdge PRO*Tools for monitoring SmartObject applications:

- Procedure Object Viewer
- The Application Debugger
- Pro*Spy Plus

Procedure Object Viewer

The Procedure Object Viewer keeps track of all procedures you run in your Progress session and displays information about procedure objects. You can use the information and options that this tool provides to manage procedures when you work with SmartObject applications. You can also make use its information when you are using the Debugger.

Starting the Procedure Object Viewer

The Procedure Object Viewer is accessible from the AppBuilder's PRO*Tools palette. To open the PRO*Tools palette, choose Tools → PRO*Tools from the AppBuilder main window. Figure 6–1 shows the PRO*Tools palette and identifies the Procedures Object Viewer icon.

![Figure 6–1: Procedure Object Viewer icon on PRO*Tools palette](image1.png)

Clicking this icon opens the Procedure Object Viewer, shown in Figure 6–2.

![Figure 6–2: Procedure Object Viewer](image2.png)
**Monitoring SmartObject applications**

**Viewing procedure objects**

The Procedure Object Viewer lets you see immediately what internal procedures are available for a SmartObject and what their parameters are. As shown in Figure 6–3, its Procedure Objects selection list displays a list of all procedures you run in your Progress session. When you select a procedure from this selection list, the **Internal Entries** selection list displays all internal procedures available in the external procedure. Similarly, when you select an entry from the **Internal Entries** selection list, the **Parameters** selection list displays a list of the entry’s parameters.

![Procedure Object Viewer](Image)

**Figure 6–3:** Selecting procedure objects and internal entries

The Procedure Object Viewer application does not merely display information. It also lets you:

- Run an internal procedure inside of a procedure object to see its effect, provided that the procedure does not take parameters.
- Look at a SmartObject’s SmartInfo dialog box and view the SmartObject’s properties.

By default, the Procedure Object Viewer views all procedures running persistently in a Progress session except those used by the Progress tools and those SmartObjects that are in design mode. If you want to see SmartObjects in design mode, choose View→Hide UIB-Mode Objects.

**The Application Debugger**

The Debugger enables you to move automatically from object to super procedure, and from event publisher to subscriber. You can use this tool to follow the flow of control among the persistent procedures that make up an application.

**To begin a debugging session for a SmartObject application:**

1. Choose a starting point for a debugging session by picking the event procedure where you want to start tracking the program flow. You can find this procedure in the Procedure Object Viewer.
2. Create a local version of that event procedure or function in the SmartObject where you want to begin tracing behavior.
3. In that procedure or function, insert the following statements before the RUN SUPER statement:

```plaintext
DEBUGGER:INITIATE( )
DEBUGGER:SET-BREAK( )
```

4. Save this object and run the SmartWindow that contains it. The Debugger starts up the first time that procedure or function executes.

5. When the Debugger starts up, use the STEP command, which causes the Debugger to step into the super procedure version of that same procedure. This effectively puts a breakpoint at the start of an internal procedure or function in a super procedure without the necessity of editing the super procedure itself.

Alternatively, if you run your Progress session with the `-debugalert` startup option, you can simply code a message statement in Step 3, instead of explicitly invoking the Debugger. For example:

```plaintext
MESSAGE "proc-name".
```

Then, when you run your application and the message alert box appears, press the HELP button to see a stack trace of the Progress 4GL code and determine where you are in the application. If you now press the Debug button on the stack trace window and the OK button on the original message alert box, Progress invokes the Debugger to allow you to move through the application from that point. (If you do not want to enter the Debugger, do not press the Debug button.)

**Debugger tips**

Here are some useful notes for using the Debugger:

- When the Debugger is at a RUN SUPER statement, a STEP command steps into the code in the next super procedure up the chain.
- When the Debugger is at a PUBLISH statement, a STEP command steps into the event procedure in the first subscriber for that event. Continuing to STEP or NEXT through the code walks through all subscribers to that event.
- To display what procedure is executing, use the THIS-PROCEDURE:FILE-NAME expression.
- To display the name of the SmartObject on whose behalf a super procedure routine is executing, use the TARGET-PROCEDURE:FILE-NAME expression.
- To display the name of the calling procedure, use the SOURCE-PROCEDURE:FILE-NAME expression.
- To ensure the Debugger has access to all procedures you want to walk through, add the %DLC%/src (Windows) or $DLC/src (UNIX) directory to the PROPATH after the gui directory.
Pro*Spy Plus

Pro*Spy Plus is an OpenEdge PRO*Tool whose primary function is to serve as a training and debugging aid for use when developing SmartObject applications. Specifically, this tool logs execution flow information for the application and provides it to the developer. This section provides a brief overview of Pro*Spy Plus. For complete details on its use, see the online help.

Pro*Spy Plus supports the:

- Hierarchical display of program call data in the Pro*Spy Plus window. This tree view presentation allows you to drill down, backtrack into an application’s call flow, and trace application execution. This functionality allows you to quickly investigate and determine where processing might have gone astray. The Pro*Spy Plus window also supports a find feature that is associated with its browser pane.

- Definition of a criteria by which you can search through, or filter, a trace you can initiate on a call stack. Using the Filter dialog box, you build filter criteria to inspect specific elements in a call stack and display the criteria’s results in this window’s browse.

Pro*Spy Plus is accessible from the AppBuilder’s PRO*Tools palette. To open the PRO*Tools palette, choose Tools→PRO*Tools from the AppBuilder main window. Figure 6–4 shows the PRO*Tools palette and identifies the PRO*Spy Plus icon.

Figure 6–4: Pro*Spy Plus icon on PRO*Tools palette

Clicking this icon opens the Pro*Spy Plus window, shown in Figure 6–5.

Figure 6–5: Pro*Spy Plus window

This window includes a menu bar and a toolbar. In the menu bar, you can (among other things) open an existing log file to view it in the hierarchical tree view section. Using buttons on the toolbar, you enable and disable the logging of call stack information. The rest of the window displays the results of the current recording.

For more information, see the Pro*Spy Plus online help.
AppBuilder requirements for SmartObjects

If you want to design your own SmartObject procedure file, make sure the src/adm2/smart.i class include file is included, directly or indirectly. You must be aware of the following points about how the AppBuilder interacts with SmartObject procedure files:

- A master procedure file must be runnable by itself:
  - It must use no shared variables.
  - It must receive no parameters (from other external procedures).
  - It must compile successfully.

- If the procedure file is designed to have instance properties, the AppBuilder must be able to execute the editInstanceProperties event procedure for the procedure file. If the AppBuilder cannot locate this event procedure, the AppBuilder disables the Instance Properties option on the Instance menu.

- The AppBuilder determines which SmartLinks it will allow for the procedure file by checking the SupportedLinks property (using the getSupportedLinks function). The AppBuilder also checks the SupportedLinks property to see if the procedure file can be a Page-Target. If so, the AppBuilder enables the Pages button on the Procedure Settings dialog box and the Page number area in the status bar on the AppBuilder main window.

  The initial value of the SupportedLinks property is the ADM-SUPPORTED-LINKS preprocessor name, which is usually defined in the object template.

- The procedure file must contain the following ADM event procedures:
  - initializeObject
  - destroyObject

**Note:** The AppBuilder also dispatches the hideObject and viewObject event procedures, if they are available.

- If the procedure file is for a SmartObject that is meant to be visualized, it must return a value for the getContainerHandle function. If the RETURN value is not the handle of a frame, the AppBuilder does not initialize the SmartObject; instead, the AppBuilder creates a visualization using the SmartObject icon. For example, the SmartDataObject has no visualization, so the AppBuilder visualizes it using the SmartDataObject icon.

- If the procedure file is for a SmartObject that is designed to be visualized with a frame, the AppBuilder runs the repositionObject and resizeObject procedures, if they are available. These procedures determine whether the visualization is movable and/or resizable, respectively, at application assembly time. For the AppBuilder to visualize a procedure file with a frame, it must be able to set the parent of the frame by invoking the setObjectParent function.
Using GroupAssign SmartLinks

GroupAssign SmartLinks are intended for grouping updates in multiple SmartDataViewers into one transaction. To do this, you link a master SmartDataViewer to the SmartDataObject with an Update SmartLink, use GroupAssigns to link this SmartDataViewer to the other SmartDataViewers that will be grouped with it. When the updateRecord procedure executes, the collectChanges procedure runs for the master SmartDataViewer, to collect changed field values from all its GroupAssign-Targets. The collected changes serve as the input to the submitRow procedure in the data source that updates the database. Figure 6–6 illustrates this process. The arrows in the figure indicate SmartLinks.

Figure 6–6: Grouping updates with GroupAssign SmartLinks

Note the SmartLinks in Figure 6–6:

- The Customer SmartDataObject is linked to each Customer SmartDataViewer with a Data SmartLink.
- The master SmartDataViewer is linked to the SmartDataObject with an Update link.
- The master SmartDataViewer is linked to each of the other SmartDataViewers, in a star pattern, with a GroupAssign SmartLink.
- The Update SmartPanel is linked to the master SmartDataViewer with a TableIO SmartLink.
The following procedure outlines how to use the Customer table in the Progress sample database, links various SmartDataViewers with GroupAssign SmartLinks so that the application can update a single customer record across multiple SmartDataViewers.

To update a single customer record across multiple SmartDataViewers:

1. Create a SmartDataObject for the Customer table.

2. Create a SmartDataViewer for the Customer table that contains the following fields:
   - CustNum
   - Name
   - Contact
   - Phone
   - Fax
   - EmailAddress

   Name this SmartDataViewer vcustcontact.w. This is the master SmartDataViewer.

3. Create a second SmartDataViewer for Customer table that contains the following fields:
   - Address
   - Address2
   - City
   - State
   - PostalCode
   - Country

   Name this SmartDataViewer vcustaddress.w.

4. Create a SmartWindow and on it create a SmartFolder with two tabs, labeled Contact and Address.

5. Drop the SmartDataObject onto the window.

6. Drop the vcustcontact.w SmartDataViewer (the master SmartDataViewer) onto page 1 of the SmartWindow and link it to the SmartDataObject with Data and Update links.
7. Drop the vcustaddress.w SmartDataViewer onto page 2 of the SmartWindow and add SmartLinks as follows:
   – A Data link from the SmartDataObject to this SmartDataViewer
   – A GroupAssign link from the vcustcontact.w (master) SmartDataViewer to this SmartDataViewer

8. Drop an Update SmartPanel onto page 0 of the window and link it to the vcustcontact.w (master) SmartDataViewer.

When this application is run, it can make updates to a single record through both SmartDataViewers. These updates are saved together, updating the database with all changes to a single customer record in one transaction.
Developing Your Application’s Business Logic

The AppBuilder supports the development and deployment of applications in heterogeneous environments (single or combined client/server, n-tier, or Web-enabled environments). The business logic for your application determines how you implement it. Developing and deploying a distributed SmartObject application requires a more detailed understanding of how a SmartDataObject operates, which this chapter provides. It contains the following sections:

- SmartDataObject features and operating characteristics
- SmartDataObject query and update operations
- ADM event procedures and functions in the SmartDataObject
- SmartDataObjects and SmartLinks
- Validation procedures
- Running SmartDataObjects in a distributed environment
- Managing SmartDataObjects in distributed mode
- AppServer-related SmartDataObject properties
- Running SmartDataObjects from a non-Progress client
- Java applications and SmartDataObjects
**SmartDataObject features and operating characteristics**

Progress applications require an object that can manage a database query and the associated data update logic. The SmartDataObject serves this purpose. It is designed to meet several key ADM goals:

- **User-interface-independence** — SmartObjects that manage the user interface do not manage data and, conversely, SmartDataObjects that manage data do not manage the user interface.

- **Open interface** — SmartDataObjects can function in an Open4GL environment with non-Progress objects and applications.

- **Distributed** — SmartDataObjects can function appropriately and efficiently “close to the user interface” or “close to the data”; that is, they can function competently at any location within a distributed application.

The SmartDataObject provides the following features:

- A basic database query definition that can be refined at run time

- Query management logic such as open query, repositions, key finds, and sending rows or batches of rows between client and server

- Formatting and other logic in support of one visualization or many separately defined visualizations

- Validation and other data-checking code and business rules

- Database update logic

You can associate business rules or any other required logic with the SmartDataObject.

The operating characteristics of the SmartDataObject make it suitable for heterogeneous environments:

- The SmartDataObject is the only SmartObject that connects directly to the database.

- The SmartDataObject takes a database query and turns it into a temp-table representation used by all other SmartObjects.

- The SmartDataObject can be transparently and efficiently distributed using the AppServer.

- The SmartDataObject handles batching rows, performing updates, and reporting errors.
SmartDataObject query and update operations

SmartDataViewers and SmartDataBrowsers provide the display and browse capabilities that make the data retrieved by a database query accessible to an application user; however, these SmartObjects never directly access a database. The query and update operations that do access the database are performed, instead, by a SmartDataObject to which the SmartDataViewer or SmartDataBrowser is linked. The SmartDataObject manages query and update operations using two automatically created Progress temp-tables, RowObject and RowObjUpd, which serve as data repositories within the application. Figure 7–1 illustrates data flow among a database, a SmartDataObject and its temp-tables, and a client object of the SmartDataObject, in this case a SmartDataViewer.

Figure 7–1: The SmartDataObject and its temp-tables

The following sections discuss how the SmartDataObject uses the RowObject and RowObjUpd temp-tables in query and update operations. A subsequent section describes special fields in the RowObject and RowObjUpd temp-tables.

Query operations and the RowObject temp-table

The SmartDataObject accesses a database using its own database query. Progress transfers the result set retrieved through the query into the SmartDataObject’s RowObject temp-table, which contains the columns defined for the SmartDataObject. The data in the temp-table is the access mechanism for all visualizations or other objects that use the SmartDataObject. Client objects see only the columns in RowObject, receive column values only from RowObject, and return updated values and new rows only to RowObject.

The SmartDataObject uses the RowObject temp-table to manage data in both non-distributed and distributed mode. In distributed mode, the use of this temp-table allows the SmartDataObject to operate in any of these configurations:

- Run in a Progress client session with a direct connection to the database
- Run split between a client object and a server object running on an AppServer
- Run entirely on an AppServer
A SmartDataObject that operates as part of a client-side SmartDataObject application is a Navigation-Target, so it responds to repositioning commands such as fetchNext. It also can be a Data-Source, passing values from its RowObject temp-table to other objects for display and update. In addition, it can be a Data-Target, so it can receive key field values from other SmartDataObjects and use them in defining its own query.

**Update operations and the RowObjUpd temp-table**

SmartDataObject update operations require an additional Progress temp-table called RowObjUpd. This temp-table, a copy of the RowObject temp-table, is used for handling all types of changes to data rows: updating existing rows, adding rows, and deleting rows:

- When an existing row is updated, Progress creates in RowObjUpd a *before image* of the row; that is, the row with its original row values, as it appeared before the update. When the update is committed, the updated version of the row also is written to the RowObjUpd table. The server-side code uses the before image to verify that the row’s data was not changed by another user since it was read, then uses the new values to update the database.

- When a row is added and the add is committed, the new row is written to the RowObjUpd table.

- When a row is deleted, the deleted row is removed from the RowObject table and written to the RowObjUpd table to be passed to the server.

The SmartDataObject transfers updated and newly added rows back to the database when it receives a Commit request. Depending on how the SmartDataObject’s AutoCommit property is set, the commit can be performed either automatically when an update is received (the default behavior) or when a specific Commit request is received after a set of updates are made. The default behavior is the typical usage; however, to change to the nondefault behavior, set the SmartDataObject’s AutoCommit property to NO. For details, see the “Commit SmartPanel” section on page 4–11.

When a change is committed, the RowObjUpd temp-table is passed back to the server (if the SmartDataObject is divided between client and AppServer), and the code in serverCommit reads the table and makes the appropriate changes to the database. For details, see the field description of RowMod in the “Nondata fields in the RowObject and RowObjUpd temp-tables” section on page 7–4.

**Nondata fields in the RowObject and RowObjUpd temp-tables**

A RowObject temp-table contains more than just fields selected from database tables. It also contains the following information:

- A temp-table definition (the first line of the table)
- Field definitions for the temp-table data (corresponds to the database data)
- Field definitions for the temp-table itself
- Index definitions
For example:

```
DEFINE TEMP-TABLE RowObject RCODE-INFORMATION
  FIELD CustNum LIKE Customer.CustNum VALIDATE ~
  FIELD Name LIKE Customer.Name VALIDATE ~
  FIELD Address LIKE Customer.Address VALIDATE ~
  FIELD City LIKE Customer.City VALIDATE ~
  FIELD State LIKE Customer.State VALIDATE ~
  FIELD PostalCode LIKE Customer.PostalCode VALIDATE ~
  FIELD RowNum   AS INTEGER
  FIELD RowIdent AS CHARACTER
  FIELD RowMod   AS CHARACTER

INDEX RowNum   IS PRIMARY RowNum
INDEX RowMod   RowMod
INDEX RowIdent RowIdent.
```

The RowObject temp-table contains three special fields, RowNum, RowIdent, and RowMod. These fields, which are maintained by the SmartDataObject support code, are used to manage queries and updates as follows:

- **RowNum** — An integer value that represents the sequence number of the row within the query’s result set. A unique value is assigned to each row as it is added to the table; basically it is used to keep the rows in a consistent sort order. The values begin arbitrarily with 900,000, which generally assures that the values of this field remain positive whether the temp-table is built forward or backward.

- **RowIdent** — A character field that contains a comma-separated list of the ROWIDs of the database records from which the RowObject row is derived. The database ROWIDs are used to re-retrieve the records from the database when performing an update. If the records in the RowObject temp-table result from a join, the ROWIDs in this field are a combination of the ROWIDs for the joined tables.

- **RowMod** — A character field that contains a one-character code that indicates the kind of update operation performed on a row:
  - For a newly added row, RowMod is set to A if the row results from an Add operation or C if it is results from a Copy operation.
  - For a row that is to be deleted, RowMod is set to D.
  - An Update operation returns two rows: a copy of the row as it was received, for which RowMod is set to “” , and the row with its changes, for which RowMod is set to U. The copy of the row as received is used to verify that the corresponding database records were not changed since the user accessed the row, and the row with changes is used to update the database.
The RowObjUpd temp-table contains the same fields as the RowObject temp-table plus the ChangedFields field, which is used only for updates:

- **ChangedFields** (RowObjUpd temp-table only) — A character field that contains a comma-separated list of the names of all SmartDataObject columns in the current row that were modified. This list is used to ensure that only those fields that were not changed previously are written back to the database, to minimize interference between different users of an application who might be changing different fields in the same record. This field is maintained automatically by the SmartDataObject support code, regardless of how each field’s value may have been changed.
ADM event procedures and functions in the SmartDataObject

This section documents the internal procedures and functions of the SmartDataObject. For comprehensive details about these procedures and functions, see *OpenEdge Development: ADM Reference* and the online help.

The SmartDataObject is supported by two super procedures in addition to `smart.p`: the `query.p` procedure contains all logic relative to managing the database query, and the `data.p` procedure (with its include file `data.i`) provides the remaining support.

### Opening a database query

The `openQuery( )` function opens a database query. Typically it is called when the SmartObject is initialized, however, you can override this by setting the `OpenOnInit` property to `FALSE`. (The default is `TRUE`.)

### Dynamic query manipulation

The ADM provides a set of functions that allow you to modify a SmartDataObject’s database query dynamically. These functions modify the query’s `WHERE` clause in various ways:

- `assignQuerySelection( )` and `addQueryWhere( )` append new criteria to existing criteria.
- `removeQuerySelection( )` and `columnQuerySelection( )` work with field expressions that you add with `assignQuerySelection( )`.
- `setQuerySort( )` changes the sort criteria.
- `setQueryWhere( )` sets the `WHERE` clause criteria.

The manipulated query is stored in a property and the actual `QUERY-PREPARE` and `QUERY-OPEN` do not take place until `openQuery( )` is called.

You also can write code to obtain the `QueryHandle` property and perform your own `QUERY-PREPARE`. If the SmartDataObject is divided between client and AppServer, such code must be executed on the AppServer for the `QueryHandle` property to be valid.

The remainder of this section provides more detailed information on the ADM query-manipulation functions. For more information about the `WHERE`, `QUERY-PREPARE`, and `QUERY-OPEN` keywords, see *OpenEdge Development: ABL Reference*.

### assignQuerySelection( )

The `assignQuerySelection( )` function adds field comparison expressions to the query. It supports the majority of standard database queries. With this function:

- The criteria are added to the existing `WHERE` clause.
- Fields, values, and operators have separate input parameters.
- Each field expression is distributed to the correct table’s `WHERE` clause.
• A field can be referenced several times, while a field and operator is considered unique. Thus, a field/operator value pair overwrites the value previously added to the \textit{WHERE} clause with this method.

• An implicit AND is used to add new expressions to existing expressions. (This function does not support OR.)

• An implicit AND is used between field expressions. (This function does not support OR.)

\textbf{addQueryWhere( )}

The \texttt{addQueryWhere( )} function allows you to add more complex criteria to a query. It:

• Adds criteria to the existing \textit{WHERE} clause.

• Adds the criteria to \texttt{ONE} of the tables in the queries. (It defaults to the first table reference in the expression.) If you must add criteria to multiple tables, use the function once for each table.

• Accepts an optional parameter to specify which table.

• Accepts an optional parameter to specify to specify OR.

\textbf{removeQuerySelection( )}

This function works with field expressions that are added with \texttt{assignQuerySelection}. Specifically, it allows you to remove criteria from the \textit{WHERE} clause. Fields and operators have separate input parameters.

\textbf{columnQuerySelection( )}

This function also works with field expressions that are added with \texttt{assignQuerySelection}. Specifically, it allows you to inspect all current criteria for a field, and returns all values added for a particular field as a CH\texttt{R(1)}-separated list of operators and values.

\textbf{setQuerySort( )}

This function changes the \textit{SORT} phrase.

\textbf{setQueryWhere( )}

This function sets the criteria. It also can be used to add one expression to the query. With it:

• You can pass a complete query prepare string (beginning with the \texttt{FOR} keyword).

• A blank input parameter resets the query to the design expression.

• The criteria is added to \texttt{ONE} of the tables in the queries. (It defaults to the first table reference in the expression.)

\textbf{Caution:} \texttt{setQuerySort( )} wipes out any other dynamically added criteria from the \textit{WHERE} clause, including \texttt{ForeignFields}.
Data transfer in queries

Rows are transferred from a database query to a SmartDataObject temp-table, called RowObject, when the first fetch operation takes place. Additional rows are transferred as needed when either of the following operations is performed:

- A fetchNext operation that moves beyond the end of the list of rows currently in the RowObject table
- A fetchLast operation

The transfer of rows is performed through the sendRows procedure, which has both a client and a server component to support transparently a SmartDataObject that is divided between client and AppServer. The number of rows transferred at a time is determined by the RowsToBatch SmartDataObject property. (The initial value is 200.)

By default, the RowObject data set is built up contiguously starting at the beginning, thus the fetchLast operation causes the retrieval of successive batches of rows until it reaches the end of the data set. For a large data set, this can be a very slow process. Progress Software Corporation recommends that for potentially large data sets, you modify this default behavior by changing the Rebuild On Reposition toggle box in the SmartDataObject instance properties dialog box to YES. (This sets the Rebuild On Reposition SmartDataObject property.) The data set is now rebuilt whenever the query is repositioned to a row outside of the current client-side data set.

You use the fetchFirst, fetchNext, fetchPrev, and fetchLast procedures to position within the RowObject table. Once you are positioned on a row, you use the colValues( ) function to return a list of formatted values for the current row. (See OpenEdge Development: ADM Reference or the online help for details on the format of the values returned.) Whenever the cursor position in the RowObject query changes, the dataAvailable event occurs. (Objects such as SmartDataViewers subscribe to this event.) Additional procedures and functions allow many useful operations, including but not limited to the following:

- Repositioning to a particular database ROWID (fetchRowIdent)
- Identifying the database ROWID of a record that satisfies a where-clause expression (rowidWhere)
- Obtaining the latest database values for the current row (refreshRow)

The following example illustrates how to reposition to a particular record in a SmartDataObject’s query. In this application, the application user can search a database for a customer name. If the application finds the name, it displays the customer information in the SmartDataViewer.

The SmartWindow in this example contains the following SmartObjects:

- A SmartDataObject for the customer table
- A SmartDataViewer whose data source is the SmartDataObject
- A name-search field and a button to start the search
The application user enters a customer name into the name-search field, then presses the search button. The search button contains the following trigger code:

```
ON CHOOSE OF BtnSearch DO:
  DEFINE VARIABLE cRowIdent AS CHARACTER NO-UNDO.
  DEFINE VARIABLE cSearch   AS CHARACTER NO-UNDO.
  ASSIGN
    cSearch  = "Name BEGINS '" + NameSrch:SCREEN-VALUE + "'",
    cRowIdent = DYNAMIC-FUNCTION('rowidWhere':U IN h_dcustomers, cSearch).
  IF cRowIdent NE ? THEN
    DYNAMIC-FUNCTION('fetchRowIdent':U IN h_dcustomers, cRowIdent, '':U).
  END.
```

The `rowidWhere` function in this code returns the ROWID of the first database query row that satisfies the where clause specified in `cSearch`. The `fetchRowIdent` function accepts a comma-separated list of database record ROWIDs for a SmartDataObject row that corresponds to the RowObject’s RowIdent field. If the row is currently in the SmartDataObject’s temp-table, the SmartDataObject repositions to that record, otherwise the database query repositions to that row and rebuilds the temp-table.

After the application repositions the SmartDataObject’s query to the specified customer, it refreshes the SmartDataViewer, displaying the customer data in the SmartDataViewer.

**Error handling**

Errors that occur during the transfer of data rows between the database and a visualization object can be detected either on the client side or the server side. If they are detected on the client side, error messages can go directly back to the visualization object. Errors detected on the server side, however, must go back to the visualization object through the SmartDataObject handling the row transfer. For more information on error reporting, see the “Validation procedures” section on page 7–13.

**Initiating update operations**

The event procedures in visual objects (SmartDataViewers and SmartDataBrowsers) that initiate update operations are as follows:

- `updateRecord`
- `addRecord`
- `copyRecord`
- `deleteRecord`
- `cancelRecord`
- `resetRecord`
These procedures typically are invoked from the buttons of an Update SmartPanel or a SmartToolbar. They perform only the parts of their operations relevant to the visualization, such as enabling Frame fields or displaying initial values. The actual data update side of the operation occurs in the SmartDataObject, using the following functions:

- `submitRow()`
- `addRow()`
- `copyRow()`
- `deleteRow()`
- `cancelRow()`

These functions cause changes to the RowObject table. If the SmartDataObject’s `AutoCommit` property is set to Yes (the default if no Commit Panel is attached), each row’s changes automatically are written to the database through the `commit()` function, otherwise they remain in the RowObject table until an explicit commit is performed.

**Getting and setting properties**

In addition to the standard `set` and `get` functions for properties, `data.p` has several support functions to set and get properties of individual columns in the RowObject; for example, `Format` and `DataType`. To differentiate these functions from the normal `get/set` functions for object properties, they are named `assignColumnProperty(column, value)` to assign a value (for those properties that are writable) and `columnProperty(column)` to retrieve the value.

**Signaling data availability**

The SmartDataObject indicates that data is available by publishing `dataAvailable`, and the Data-Target that has subscribed to this event responds appropriately. This happens when the SmartDataObject opens its query and when the cursor position in the query changes. This behavior makes sense for a configuration in which the SmartDataObject and its visualization are part of the same Progress client.

However, if the user interface either is not Progress or is in a different process than the SmartDataObject, the Data SmartLink is not present and the `dataAvailable` event is not sent; you must design your user interface objects to deal with this. For example, a visualization could execute the `openQuery` function in the SmartDataObject by knowing its handle rather than by using an ADM link, and then immediately start requesting data using other functions. If no other outside source is controlling the SmartDataObject, its visualization does not need the Data link or the dataAvailable event; that is, ADM links and event procedures are supported by SmartDataObjects but are not required by other new objects with which they communicate.

Generally, signaling data availability without the Data SmartLink and the `dataAvailable` event is of interest when you are using SmartDataObjects from a non-Progress environment. Within a 4GL SmartObject application, the ADM manages communication between objects using the link and event, and the code in the ADM super procedures handles this automatically.

Special considerations apply when you access a SmartDataObject from a Java application. For details, see the “Java applications and SmartDataObjects” section on page 7–29.
SmartDataObjects and SmartLinks

The ADM provides separately defined Data and Update SmartLinks to allow maximum flexibility in how you build and assemble SmartDataObjects:

- The Data link passes values from one object to another for display.
- The Update link passes changed values back to a SmartDataObject to be validated and saved in the database.

A SmartDataObject that is the Data-Source for a particular SmartObject can be the Update-Target for the same SmartObject. For example, a SmartDataObject that is the Data-Source for (and passes value to) a SmartDataViewer can be the Update-Target for that SmartDataViewer. Similarly, if a SmartDataBrowser allows updates, it must pass the updates to a SmartDataObject, because that is where the update logic resides. In this case, the SmartDataObject is the Update-Target for the SmartDataBrowser.

The SmartDataObject is designed to participate in a peer-to-peer SmartObject client relationship. However, it is also designed to operate as a server object that receives requests from other client objects, satisfies them strictly through OUTPUT parameters to those requests, and does not need links or callbacks to function.
Validation procedures

The SmartDataObject supports the definition of validation procedures for the following:

- RowObject temp-table fields
- The RowObject temp-table record as a whole
- Update transaction on the server side

This allows you to define several distinct levels of data validation, as discussed in this section.

Data Dictionary validation

As you might expect, Data Dictionary validation is simply the validation expression (and accompanying validation message) defined in the Data Dictionary. The temp-table columns inherit this validation for each column that maps to a single database field. Client objects can query it using the `columnValExp` and `columnValMsg` functions; for example, you might do this at design time to include Data Dictionary validation in the generated code for a non-Progress visualization.

SmartDataBrowsers and SmartDataViewers include the same field definition list as the SmartDataObject. In this way, these objects automatically inherit the format, validation expression, and other field attributes of each field in the RowObject, as defined in the SmartDataObject. This field validation can be performed in the visualization even without a database connection available at run time (in the event that the application is distributed using AppServer), as long as the field validation does not require database lookups (CAN-FIND and so on). Because of this restriction, the column definition dialog box (which appears when you build a SmartDataObject) includes a toggle box that you can turn off to prevent the RowObject field from inheriting the field validation for the database field from which that column is derived. You should normally turn off this toggle box if the Data Dictionary validation requires a database lookup and the client might be run without a database connection of its own to satisfy that lookup.

The Data Dictionary validation is not automatically executed in the SmartDataObject itself. It normally occurs in the SmartDataViewer or SmartDataBrowser when saving changes to an entire row.

Field-level validation procedures

You can define field validation procedures (also called column validation procedures) in the SmartDataObject for each updateable field in the temp-table. To do this, you define an internal procedure named `columnValidate` that takes the column value as an input parameter. Each time the SmartDataObject receives an updated field from a client object, the `submitRow` function in the super procedure `data.p` runs the field validation procedure (if any). The procedure receives the field’s value as an INPUT parameter. If the validation fails, the procedure can signal an error and log a message by RETURNing the message text. This message is displayed to the user as the error text, and in the visualization, focus is applied to the invalid field.
The field-level validation procedure allows more extensive value checking than the logical ValExp (schema validation expression) permits. Because the schema validation expression might not always be executed in the client visualization, it is a good idea to duplicate any important checks in the SmartDataObject field validation procedures. The field validation procedures are executed only if a field value changes.

**Note:** To work around the restriction that field-level validation executes only when a field value changes, use record-level validation instead of field-level validation. This works because the record-level validation procedure can access any field in the row object by name. For details, see the “Record-level validation procedures” section on page 7–15.

The following example, which utilizes the Progress sample database, describes a SmartObject application that implements a field-level validation procedure in a SmartDataObject. Specifically, it creates field-level validation that ensures that application users enter OrderLine quantities that are greater than zero.

To implement a field-level validation procedure in a SmartDataObject in the AppBuilder:

1. Create a SmartWindow.
2. Create a SmartDataObject for the OrderLine table in the sports2000 sample database.
3. Create a SmartDataViewer for the OrderLine table, using the SmartDataObject created in Step 2 as its Data-Source.
4. Drop the SmartDataObject and SmartDataViewer onto the SmartWindow and link them with Data and Update links.
5. Drop a Navigation SmartPanel onto the SmartWindow and link it to the SmartDataObject with a Navigation link.
6. Drop an Update SmartPanel onto the SmartWindow and link it to the SmartDataViewer with a TableIO link.
7. Create in the SmartDataObject an internal procedure called QtyValidate that has the following code:

   ```
   DEFINE INPUT PARAMETER valqty AS CHARACTER NO-UNDO.
   IF INTEGER(valqty) = 0 THEN
      RETURN "Order Line quantity must be greater than zero".
   END PROCEDURE.
   ```

With this field-level validation procedure in place, an application user who enters 0 (zero) for the Order Line quantity receives the specified error message, and focus is applied to that field in the SmartDataViewer. Note, however, that if the initial value of the Order Line quantity is 0 and the application user does not modify the field in the visualization, the validation does not run. This is because field validation procedures are executed only if a field value changes.
Record-level validation procedures

You can define a validation procedure called RowObjectValidate in the SmartDataObject for the temp-table record as a whole. This record-level validation procedure (also called a row validation procedure) can perform cross-field validation and other kinds of checks for a set of updates to a single record. The submitRow function in the super procedure data.p executes the record-level validation procedure (if defined) whenever one or more field values in a record are updated, after the field validation procedures run. The procedure can access any field in the RowObject row by name and, like the field-level validation procedure, it signals error by RETURNing an error message text to the caller.

Note: Because the RowObjectValidate procedure can access any field in the row object by name, you can use record-level validation instead of field-level validation to work around the restriction that field-level validation executes only when a field value changes.

The fact that RowObjectValidate procedures in a SmartDataObject run whenever one or more field values in a record are updated has a side effect you must be aware of. If your RowObjectValidate procedure validates a field that is not included in a visualization that updates the table and the procedure returns an error, the application user cannot access the field or fix the error.

Caution: RowObjectValidate procedures run for every visualization that can add or update the table. If you add a RowObjectValidate procedure to a SmartDataObject, make sure any visualization that might fire the procedure includes the appropriate fields so application users can add or correct data as necessary to pass the validation check.

Suppose, for example, a customer SmartDataObject includes a RowObjectValidate procedure that ensures at least one of the following fields—phone, fax, email address—is populated for a customer record. If the application that uses this SmartDataObject includes a SmartDataViewer that can add customer records but does not include any of these fields, the validation procedure will not allow the application user to add a record and will return an error noting the user must enter either a phone, fax, or e-mail address. However, the user cannot access these fields and will not be able to enter any of these values.

The following example, which uses the Progress sample database, describes a SmartObject application that implements a record-level validation procedure in a SmartDataObject. Specifically, it creates record-level validation that ensures that application users enter credit limits greater than zero.

To implement a record-level validation procedure in a SmartDataObject:

1. Create a SmartWindow.
2. Create a SmartDataObject for the customer’s table that includes the name and creditlimit fields.
3. Create a SmartDataViewer for the SmartDataObject in Step 2, with its fields in it.
4. Drop the SmartDataObject and SmartDataViewer onto the SmartWindow and link them with Data and Update links.
5. Drop a Commit SmartPanel on the SmartWindow and link it to the SmartDataObject with a Commit link.
Developing Your Application’s Business Logic

6. Drop a Save SmartPanel onto the SmartWindow and link it to the SmartDataViewer with a TableIO link.

7. Create in the SmartDataObject an internal procedure called RowObjectValidate that has the following code:

```vbnet
If RowObject.CreditLimit <= 0 Then
    RETURN "Credit limit must be greater than zero".
End Procedure.
```

With this record-level validation procedure in place, an application user who enters zero for the credit limit receives the specified error message.

As another example, the following code, when placed in the RowObjectValidate internal procedure, cancels any data change (add or update) where the value of SalesRep is SLS (this is a new sales rep) and the Balance or CreditLimit fields are out of a specified range:

```vbnet
IF RowObject.SalesRep = "SLS": THEN
    IF RowObject.Balance > 0 OR RowObject.CreditLimit > 5000 THEN
        RETURN "SalesRep Not Authorized For This Change".
    END IF
END IF
```

Transaction-level validation procedures

As updates are made, the ADM creates a version of the RowObject temp-table, called RowObjUpd, that contains only the rows that are being updated (that is, modified, added, or deleted). It passes the RowObjUpd temp-table to the server side of the Commit process, which finds the corresponding database records (using the RowIdent field) and moves the update rows into those records. ADM allows you to implement transaction-level validation at various points in the process.

The SmartDataObject contains four entry points for transaction-level validation procedures: preTransactionValidate, beginTransactionValidate, endTransactionValidate, and postTransactionValidate. These entry points are simply procedure names that execute NO-ERROR. If you want your SmartDataObject to perform custom validation at a particular point in a transaction, you simply write an internal procedure of the corresponding name in the SmartDataObject. These procedures are executed where the database connection is. If the SmartDataObject is divided between client and AppServer, they are executed on the AppServer.

The four entry points operate as follows:

- **preTransactionValidate** — This procedure is executed immediately before a transaction block begins. At this point, all rows in the RowObjUpd temp-table are available, can be read, and, if necessary, can be modified. This is the appropriate place to put checks that would result in canceling a transaction before it begins; for example, checks that verify the validity or consistency of rows being added, updated, or deleted. It also is the place to put code that changes or sets any values in those records that have not been set before; for example, totals.
• **beginTransactionValidate** — This procedure is executed immediately after the beginning of a transaction block. This is the appropriate place for business logic that:

  – Updates related database records that should be updated as part of the same transaction

  – Does not depend on SmartDataObject updates having been made to the database already

• **endTransactionValidate** — This procedure is executed immediately before the end of a transaction block. This is the appropriate place for:

  – Business logic that updates related database records that should be updated as part of the same transaction

  – Performing other checks that are dependent on SmartDataObject updates having been made to the database already

For example, if the logic for an Order SmartDataObject must total all orders to compare them with a credit limit, then it must be able to read not only updated order records but also any records already in the database. This logic is easier to implement if it is executed after the updated rows are written back to the database.

As another example, suppose you add to a SmartDataObject a row for which the following are true:

  – It is a one-to-one join of two database tables.

  – Its key field, which is the join field for the tables, is assigned from a numeric sequence in the database.

In this case, it is impossible to assign the key field value for the second table any sooner than at the end of the transaction. An `endTransactionValidate` procedure can read the primary table record from the database, retrieve the key value, and assign it to the secondary table record.

• **postTransactionValidate** — This procedure is executed immediately after the end of the transaction block. This is the appropriate place to put business logic that performs other checks that should not be part of the basic transaction; for example, sending status messages or other work that should be performed in a separate transaction or in no transaction at all.

Each of these procedures has access to the RowObjUpd table and to any connected database tables. The procedures signal error by `RETURNing` an error message; if any of these procedures does return an error message, processing stops. Thus:

• When `preTransactionValidate` returns an error, it cancels the update before the transaction begins.

• When `beginTransactionValidate` or `endTransactionValidate` returns an error, it undoes the transaction.
Developing Your Application’s Business Logic

- When `postTransactionValidate` returns an error, there is no effect on the transaction: it does not undo the transaction because at that point, the transaction has already been completed.

**Note:** The version of ADM available in Progress V9.0 provided a single entry point called `TransactionValidate` for transaction-level validation. In subsequent releases, the `preTransactionValidate` procedure effectively replaces `TransactionValidate`, but `TransactionValidate` is still supported for backward compatibility. As is the case with `preTransactionValidate`, `TransactionValidate` executes NO-ERROR so it runs only if it was defined for the SmartDataObject.

---

**Database trigger procedures**

Database trigger procedures are not part of the SmartDataObject model, and Progress Software Corporation recommends that you limit them to performing essential data integrity checks. There are two reasons for this recommendation:

- They are relatively more difficult to maintain and examine than code that is part of the business logic of an application.

- The checks are not performed until the actual database update is attempted.

By moving as much as possible of the application’s business logic as far from the database transaction as possible, you ensure that most data errors are reported earlier and more efficiently. If a database trigger fails during a SmartDataObject update, the transaction is aborted and the error message, if any, is reported back to the client object, where either the data must be modified or the update canceled.

The ADM provides a standard error reporting mechanism that database triggers can use. This mechanism, the `addMessage` procedure, allows the SmartDataObject to log one or more error messages and return them in a standard format to a visualization or other client object, which can then present them to the user as appropriate, including repositioning the cursor to the field where the error occurred. The `addMessage` procedure takes as arguments the message text, the field name, if applicable, and the table name, if applicable. If database trigger procedures run `addMessage`, these messages are returned to the client visualization for display, even across an AppServer connection. `MESSAGE` statements are not transmitted across an AppServer connection and are not, therefore, an appropriate way to log error messages in any database that might be used in distributed applications.
**Running SmartDataObjects in a distributed environment**

The SmartDataObject supports distributed computing. You can run a SmartDataObject in a distributed environment using either the traditional client/server model or the AppServer model:

- In the client-server model, the SmartDataObject runs entirely on one machine as a single .r file and is connected to the necessary databases.

- In the AppServer model, the SmartDataObject is split across the network using AppServer technology, as follows:
  - A SmartDataObject proxy, also called a client proxy, resides on the client machine. It interfaces with other client-based procedures and with the complete SmartDataObject on the AppServer machine.
  - A complete SmartDataObject resides with the database on the AppServer machine. It interacts with the database and with the SmartDataObject proxy on the client machine.

The complete SmartDataObject and its proxy interface with each other using AppServer technology.

In this model, it is possible to use a complete SmartDataObject rather than a client proxy on the client machine. This configuration occurs when the partition is configured to run remotely and the client is connected to the database at run time. It happens because the decision on whether to use complete SmartDataObject or a client proxy is based on whether the proper databases are connected, a decision that is independent from the decision on whether to run remotely or locally. Since the partition is an instance property of the SmartDataObject, it is possible to determine which partition is to be used and whether or not it is to be executed locally or remotely only after starting up the SmartDataObject or its proxy. The SmartDataObject runs the same way regardless of whether you run the complete version or its proxy, but the complete version includes code that is not executed by the client in a remote configuration.

Regardless of how you will deploy your SmartDataObject, you need to build it only once and maintain a single set of source code. This is because when you build a SmartDataObject, you use conditional preprocessor statements that identify code sections requiring database access and enable compiling the SmartDataObject into two versions:

- A database-aware version that contains all of the SmartDataObject code. This version can be used either in the traditional client/server model, or in the AppServer model as the AppServer-resident procedure that accesses the required databases.

- A non-database-aware version that is compiled with all database-access sections compiled out of the r-code. This version is suitable only as the client proxy on the client machine in the AppServer model.

**Note:** You need to compile conditionally into database-aware and non-database-aware versions only if your SmartDataObject will run in a distributed environment using the AppServer model. This is because it is the only configuration in which the SmartDataObject is split into database-aware and non-database-aware versions.
The sections that follow describe how to set up the master (.w file) for your SmartDataObject so that it will compile conditionally into database-aware and non-database-aware versions, how the AppBuilder modifies the master, and what happens when the SmartDataObject is initialized in an application.

**Setting up the SmartDataObject master for conditional compilation**

You use the following tools to set up a SmartDataObject master so it will compile into database-aware and a nondatabase-aware versions:

- The `DB-AWARE` procedure setting in the SmartDataObject template. This already is set to YES in the `data.w` template.

- A toggle box labeled `DB-REQUIRED` in the Section Editor for code sections. When you edit an internal procedure or function in the Section Editor, the AppBuilder adds a toggle box labeled `DB-REQUIRED` to the Section Editor for any section that might contain database references. By default, this toggle box is checked (selected). If you know a section has no references to a database, you can uncheck the toggle box.

**AppBuilder modifications to the SmartDataObject master**

When you generate the master (.w file) for the SmartDataObject that has been set up for conditional compilation into database-aware and nondatabase-aware versions, the AppBuilder modifies the master as follows:

1. It checks each code-block section to see whether its `DB-REQUIRED` toggle box is checked. If it is, the AppBuilder brackets the code in that section with the following preprocessor names:

   ```
   {&DB-REQUIRED-START}
   . . .
   {&DB-REQUIRED-END}
   ```

   These preprocessor names resolve to the following code:

   ```
   &IF {&DB-REQUIRED} &THEN
   . . .
   &ENDIF
   ```

   This code excludes the section from the r-code during the compilation of the client proxy.

   If the section’s `DB Required` toggle box is unchecked, the AppBuilder does not bracket its code with the `{&DB-REQUIRED-START}` and `{&DB-REQUIRED-END}` preprocessor names, and the section is included in the r-code during the compilation.
2. If the preprocessor names for a section are present, the AppBuilder generates the following code to define them before it produces any code block that might require them:

/* DB-Required Preprocessor definitions */
&IF DEFINED (DB-REQUIRED) = 0 &THEN
  &GLOBAL-DEFINE DB-REQUIRED TRUE
&ENDIF
&GLOBAL-DEFINE DB-REQUIRED-START &IF {&DB-REQUIRED} &THEN
&GLOBAL-DEFINE DB-REQUIRED-END &ENDIF

If DB-REQUIRED is not defined when the compiler processes this block of code, all code blocks are included in the compiled r-code. However, if DB-REQUIRED is defined as false, code blocks bracketed with {&DB-REQUIRED-START} and {&DB-REQUIRED-END} are not included in the compiled r-code. This is the case when generating a client proxy (a SmartDataObject proxy to be used on a client machine); it means that the proxy is not required to be connected to any database when executed.

You can choose to compile out just a part of an internal procedure or function. To do this, uncheck the DB-REQUIRED toggle box and program the following statements around the section of code to be executed only when a database connection is available:

&IF {&DB-REQUIRED} &THEN
  . . .
&ENDIF

This technique allows you to execute different versions of the same procedure on the client and the AppServer. You might even code the client version to invoke the AppServer version; for an example, see the installed file %DLC%\src\adcm2\cltorsrver.i (Windows) or $DLC/src/adcm2/cltorsrver.i (UNIX).

3. When it generates the .w file for the SmartDataObject, the AppBuilder also produces a client proxy .w file. The proxy has the same name of the form sdoname_cl.w, where sdoname is the name of the SmartDataObject. For example, a SmartDataObject named dcust.w automatically has a proxy named dcust_cl.w. The proxy contains the following code:

/* dcust_cl.w - non-db proxy for dcust.w */
&GLOBAL-DEFINE DB-REQUIRED FALSE
{dcust.w}

This code, when compiled, first sets {&DB-REQUIRED} to FALSE, then creates an r-code proxy named dcust_cl.r that is identical to its sibling SmartDataObject named dcust.r except all sections marked DB-REQUIRED are removed.

4. The AppBuilder turns on the SmartDataObject’s DBAware attribute. It uses this attribute later, when it generates a SmartContainer that contains the SmartDataObject.
Initializing the SmartDataObject

When the AppBuilder generates a SmartContainer, it produces calls to the constructObject method for each contained SmartObject. This method launches the contained SmartObject by running it persistently. If a contained SmartObject is a database-aware SmartDataObject, the AppBuilder appends the DB-AWARE keyword to the SmartDataObject’s name, separated by a CHR(3) character. This serves as a flag to the constructObject method in the containing SmartObject to check the connected databases at run time and determine whether to run the complete SmartDataObject or its proxy.

The constructObject method determines whether to run the complete SmartDataObject or its proxy based on whether all necessary databases are currently connected when the SmartDataObject is about to be executed. If they are, constructObject launches the complete SmartDataObject. If any of the required databases are not connected, constructObject runs the client proxy instead. The proxy then makes the proper AppServer connection to the complete SmartDataObject that is its sibling, based on the partition indicated in its instance attributes setting.
Managing SmartDataObjects in distributed mode

This section describes how to manage SmartDataObjects in a distributed mode, using AppServer.

As described in the “Running SmartDataObjects in a distributed environment” section on page 7–19, you can compile a SmartDataObject into database-aware and non-database-aware versions, with sections that contain database references compiled in or out depending on whether the SmartDataObject version will be run client/server, on a client that talks to an AppServer, or on an AppServer.

A SmartDataObject client proxy contains the code needed to connect to an AppServer Broker, to run the complete SmartDataObject on that AppServer, and to send requests through that connection to retrieve and update database records from the database through the AppServer. To enable a SmartDataObject for this use, follow the steps below, either before saving the master for a new SmartDataObject from the AppBuilder or when editing the master for an existing SmartDataObject. The procedure connects to an AppServer Broker, to run the complete SmartDataObject on that AppServer and send requests through that connection to retrieve and update database records from the database through the AppServer.

To connect to an AppServer Broker:

1. Select the Procedure settings button from the AppBuilder toolbar. The Procedure Settings dialog box appears.
2. Check the AppServer Aware toggle box.
3. In the Partition fill-in field, enter the Application Service name that this SmartDataObject is to be associated with. (The Partition field appears only if you are working with a SmartDataObject master.)

This runs all instances of that SmartDataObject on the AppServer. To change this for a single instance, use the SmartDataObject’s Partition instance property. For example, you might do either of the following:

- Set the procedure properties in the SmartDataObject so all instances except one run on the AppServer. Then set the instance property for that instance to none (that is, it runs locally).
- Leave the procedure property unset in the SmartDataObject so all instances except one run locally, and set the instance property for that instance to run on the AppServer.

If the AppServer is not known earlier or is variable, you can use the Partition instance property to set it when you assemble the application.

SmartDataObject deployment files

The ADM support files are shipped only with the Windows versions of the product. This is because you normally develop SmartObjects only with the AppBuilder, which generates a great deal of code for SmartObjects. Because the include files and super procedures that form the ADM become in effect a part of the final application, it is essential you use the same version of those files to build and run your application on all platforms. As a result, you must port needed super procedures to each deployment platform along with the application code. To test SmartDataObjects on an AppServer on which the Progress software is not installed, you must put these files into an adm2 directory on that machine relative to its PROPATH.
You need the following super procedures to run standard SmartDataObjects on an AppServer, either from a Progress client, from WebSpeed®, or from a non-Progress client such as Java:

- `smart.r`
- `query.r`
- `data.r`

To facilitate testing distributed applications on a single Windows NT workstation (where you can run both a Progress client and a Progress AppServer session), these three super procedures reside in both the `tty` directory and the `gui` directory.

You must move all SmartDataObjects to be run on an AppServer to that machine. Usually you do not need to recompile the super procedures or SmartDataObjects on the target machine, unless it is a platform to which Progress .r code is not portable. However, if it is necessary or desirable to recompile the SmartObject super procedures or the application’s SmartObjects on the target machine, you also must move the following source files, as well as the custom subdirectory and its contents, to that machine, in a `src\adm2` directory relative to the PROPATH:

- `smart.i`
- `query.i`
- `data.i`
- `smrtrpt.prop.i`
- `qryprop.i`
- `dataprop.i`
- `smrtppto.i`
- `qrypto.i`
- `datappto.i`
- `smart.p`
- `query.p`
- `data.p`

When you run a SmartDataObject on an AppServer, the supporting super procedures `smart.r`, `query.r`, and `data.r` are started automatically, just as on the client, and are left running in support of any SmartDataObjects that are run for the duration of that AppServer session. All the super procedures run stateless, without maintaining any data specific to an individual client, thus the same AppServer process can service several clients in succession, even when each client expects to have exclusive use of that AppServer for the duration of its connection. As a result, your application incurs the overhead of starting super procedures once per AppServer session. (You can even run these procedures persistently as part of the session startup.)
Managing SmartDataObjects in distributed mode

**Service choices and SmartDataObject execution**

A SmartDataObject that is part of a client-side application runs on the client if either of the following is true:

- No Application Service is defined for the SmartDataObject.
- The SmartDataObject is AppServer-aware but its Configuration option is set to Local in the PRO*Tools Application Partition Deployment utility.

A SmartDataObject that runs only on the client has its own database connection and accesses the database through that connection.

However, if the SmartDataObject is AppServer-aware and the Application Service on an AppServer host machine is set to an actual service, the client SmartDataObject’s initializeObject procedure establishes a connection to that AppServer, then runs a copy of itself (that is, a file with the same filename) on the AppServer. (If the client is running the client-only SmartDataObject, with the _cl suffix, that suffix is ignored when locating the matching SmartDataObject on the AppServer.) The SmartDataObject on the AppServer opens its database query as part of its own initialization. For more information, see OpenEdge Development: Open Client Introduction and Programming.

When the client-side SmartDataObject runs its openQuery function, the sendRows procedure executes. How this procedure executes depends on whether the application is running in client/server mode or AppServer mode. In client/server mode, when the client uses its own database connection, sendRows retrieves rows from the database and populates the RowObject temp-table, which is then accessed by other objects on the client.

In AppServer mode, however, the sendRows procedure runs clientSendRows. This in turn runs serverSendRows in the AppServer SmartDataObject, which returns the RowObject temp-table to the client. The client-side SmartDataObject uses the temp-table until it is finished with it. It can then either simply exit or execute the closeQuery function, which disconnects it from the AppServer. The same procedures are used to update one or more rows in the table or to add or delete rows as in client/server mode. When the Commit function is run on the client side, it in turn runs serverCommit in the AppServer SmartDataObject, passing a temp-table containing all modified, deleted, or added rows. The serverCommit procedure writes the records back to the database as described in the SmartDataObject description or, if the update fails, returns any errors to the client.

When a 4GL client running a client-side SmartDataObject connects to the same object on an AppServer, the support code for those procedures contains all of the code needed to get updates back to the server. (The support code is located in data.i, which is compiled into each SmartDataObject, and the super procedure data.p.) You can use the same calls from a non-Progress client (for example, a Java client using a proxy built by ProxyGen). However, in this case, if the client process performs any update operations, it is its responsibility to return to the AppServer SmartDataObject an update temp-table that contains those changes. The update temp-table (called RowObjUpd in the 4GL code) must be of the same form as the RowObject temp-table that is passed to the client with the original rows. See the “SmartDataObject query and update operations” section on page 7–3 for specifications of this temp-table.

Special considerations apply when you access a SmartDataObject from a Java application. For details, see the “Java applications and SmartDataObjects” section on page 7–29.
AppServer session operating modes and SmartDataObjects

In a distributed application, a SmartDataObject inherits its operating mode—state-aware, stateless, or state-reset—from the AppServer it uses to access its database. (This is the default behavior; you need not configure the SmartDataObject specially for this to happen.) These three modes operate as follows:

- **In state-aware mode**, a SmartDataObject connects to an AppServer session and binds it for the entire length of the SmartDataObject’s client connection. At the end of the connection, the context (state) of the AppServer session remains and can be used by the next connection to that session. In this mode, it is unnecessary for a new connection to the AppServer session to re-establish context; however, it might be difficult to design a system in which each user can benefit from the context left by the previous user.

- **In stateless mode**, a SmartDataObject connects to an AppServer session but binds it only briefly to do the following:
  - Re-establish the context of the AppServer session at the end of its last request
  - Make a request
  - Save the context for the next request

  Between requests, nothing is bound. In this mode, each individual AppServer request might take slightly longer, but many clients can share fewer AppServer sessions with minimal performance degradation.

- **In state-reset mode**, the AppServer session’s state is reset to the initial state after each connection. This mode is particularly useful for applications that do not use SmartDataObjects.

In some cases, you might want a SmartDataObject to run in a state-aware mode even though the AppServer is running in stateless mode. For example, a SmartDataObject might have a query that is so complex that the overhead of re-establishing it at every use degrades performance. To request state-aware mode explicitly, you set the **Force to Stateful Operating Mode** instance property in the SmartDataObject’s instance properties dialog box. This property causes the SmartDataObject to run in state-aware mode even though AppServer session is running in stateless mode. See the “SmartDataObject instance properties” section on page 2–8.

**Note:** A SmartDataObject executes exactly the same regardless of its operating mode.
Two parameters to the AppBuilder-supported AppServerConnect function are defined as SmartDataObject properties. The following properties set these parameters:

- **ASUsePrompt** — This determines whether the connect process prompts for username/password. It is false by default.
- **ASInfo** — This is a user-defined character string that can be passed to the CONNECT method as its third parameter, if needed. It is blank by default.

There also are two other AppServer-related SmartDataObject properties:

- **ASHandle** — This holds the procedure handle of the AppServer connection, if there is one. This is set by the Connect process.
- **ASDivision** — This is set automatically when the SmartDataObject determines whether it is running as a divided object on both Client and AppServer. It is set to Client if it is the client-side object, to Server if it is the AppServer-side object, and is left blank otherwise. User code should not set this property but can examine it if needed.
Running SmartDataObjects from a non-Progress client

The SmartDataObject has an extensive API. You can access many of its internal procedures and functions from a non-Progress client application such as one written in Java. You can use the OpenEdge™ ProxyGen tool to generate a proxy that supports this interface. Because the code generated by ProxyGen is based on the INTERNAL-ENTRIES of the Progress procedure, you can control the list of exactly which internal procedures and functions are exposed to ProxyGen by editing the contents of the prototype include files. For more information on prototype include files, see Chapter 6, “Advanced ADM Topics.” Because there is no specific client-side support for SmartDataObjects in Java or Visual Basic, you must write code that uses the SmartDataObject API to control and communicate with the SmartDataObject as required by the application.

The following descriptions provide an overview of which procedures and functions you use to perform various operations. Generally, you can retrieve data from a SmartDataObject and return updates either in batch mode or a row at a time.

Batch-mode operations

In batch mode, you batch data in a Progress temp-table from a SmartDataObject and, when needed, back to the SmartDataObject. When a SmartDataObject is initialized and its database query opened, the object transfers rows from the query into the RowObject temp-table defined in the SmartDataObject. (By default, 200 rows are transferred at a time, but you can reset this with the setRowsToBatch function.) A non-Progress client application should run serverSendRows to receive this table; this function returns the RowObject temp-table, which can be browsed from the client as an SQL dataset. If you make updates, you must construct a RowObjUpd dataset with the RowNum, RowIdent, RowMod, and ChangedFields fields set for each modified row. To return this dataset to the SmartDataObject, run serverCommit, which accepts the temp-table and returns the same table with any additional fields that might have been set by database triggers (key values and so forth), along with OUTPUT parameters that contain any error messages and the RowIdent fields of any records that were not updated because of errors. The calling sequence for the fetchMessages function is described in the format of the error message string.

Single-row operations

If you want to receive and return data a row at a time, use the colValues function to obtain desired column values for the current row in the data set, then use the fetchFirst, fetchNext, fetchPrev, fetchLast, and fetchRowIdent procedures to reposition the SmartDataObject’s query. Additional batches of rows are moved from the database query into the RowObject temp-table query automatically as necessary. To perform updates to individual rows, use the submitRow function to return updated values for a single row to the SmartDataObject. You can use the deleteRow procedure to delete the current row. You also use the addRow and copyRow procedures to create new rows starting with the initial values for the table or with the values from the previous current row. After these initial values have been returned to the caller, use the submitRow function to send the final field values back to the SmartDataObject to be written to the database.

Special considerations apply when you access a SmartDataObject from a Java application. For details, see the “Java applications and SmartDataObjects” section on page 7–29.
Java applications and SmartDataObjects

A Java application running on a client machine can access a remote SmartDataObject on an AppServer. It does this through a Java AppObject, using an interface based on the Java Database Connectivity (JDBC) 2 ResultSet. The supported interface includes many of the standard ResultSet methods and adds a number of extensions.

Figure 7–2 shows how a Java Open Client accesses a Progress SmartDataObject running on an AppServer.

The Java SDOResultSet object can map directly to any specified SmartDataObject instantiated on the connected AppServer.

Figure 7–2: Java open client access to a remote SmartDataObject

To enable a Java client to access a remote Progress SmartDataObject as a JDBC 2 ResultSet, you must:

- Have at least one remote SmartDataObject deployed on the connected AppServer.
- Call the AppObject or SubAppObject class factory method, _createSDOResultSet(), to create an SDOResultSet object that maps to and can access the specified SmartDataObject.

The SDOResultSet object implements the SDOResultSet interface, which is an extended implementation of the JDBC 1 ResultSet interface. Thus, using standard JDBC 2 ResultSet methods on this SDOResultSet object, you can access Progress data provided by the specified SmartDataObject in the same way as a standard JDBC 2 ResultSet. The SDOResultSet interface also provides extensions to JDBC 2 that allow you to access the unique capabilities of SmartDataObjects.
Complete support for access to remote SmartDataObjects includes a small set of classes and interfaces that, together with SDOResultSet, come installed as part of the Open Client Runtime. For reference information on each class and interface, see the Javadoc installed in
\texttt{OpenEdge-Install-Directory/java/doc/*.html}. For information on accessing and using this interface, see \textit{OpenEdge Development: Java Open Clients}.

\textbf{Note:} The SDOResultSet interface allows you to access only the default SmartDataObject API generated by the AppBuilder. If you need to access custom programmer extensions to this API, you must access the SmartDataObject API directly. For more information, see \textit{OpenEdge Development: Java Open Clients}. 
Developing extensions to the ADM means extending the ADM class hierarchy with new or customized classes. This chapter describes how to do this, using either the AppBuilder’s New ADM Class tool or the custom files for an existing class as appropriate.

This chapter contains the following sections:

- Introduction
- Creating a new class at the bottom of the ADM class hierarchy
- Customizing an existing ADM class
- Adding application logic to your new or customized ADM class
  (including descriptions of the custom class files, design rules for modifying class files, and notes on adding properties)
- Examples (of creating a new class and customizing an existing class)
Introduction

In some circumstances when developing Progress 4GL applications that use SmartObjects, it is necessary or useful to create a new ADM class. Suppose, for example, you have several SmartDataBrowser master files that share some set of properties and/or behaviors beyond what is supported in the SmartDataBrowser as shipped, but that have other properties and/or behaviors that are unique to each master. You can create a new SmartDataBrowser template that includes the common functionality as standard behavior, and then build master files from this new template. Your custom SmartDataBrowser constitutes an extension of the ADM—a new ADM class.

Before you begin to develop a new ADM class, be sure you understand the basic ADM architecture as described in Chapter 1, “Overview,” and are familiar with the material in the other chapters of this manual.

Note: The current ADM file structure is different from that in earlier ADM versions, and takes advantage of core features that did not exist in those versions. If you implemented your own customizations to the standard ADM classes, please note the current ADM version does not support backward compatibility. Progress Software Corporation recommends you convert your applications to use the ADM custom files, as described in this chapter.

There are three ways to extend the ADM class hierarchy:

- Creating a new class at the bottom of the ADM hierarchy (also called subclassing) allows you to create new SmartObject templates without affecting existing SmartObjects: it affects only new SmartObjects that reference your new class.

  Figure 8–1 shows the ADM class hierarchy for SmartFrames, to help illustrate bottom-level extensions.

![ADM hierarchical class structure for SmartFrames](image)

The top (root) of the hierarchy is the smart class, which contains the essential behavior shared by all SmartObjects. (Recall the smart class is the root object for all SmartObjects.) As you work down the tree, you encounter successively more specific classes of behavior; in this case, the visual class and then the container class. The standard SmartObject templates are all based on bottom-level (leaf-level) ADM classes.

You use the New ADM Class tool to create the new ADM class files, then modify the standard class files by adding Progress 4GL code as required to provide behavior specific to your class. For details, see the “Creating a new class at the bottom of the ADM class hierarchy” section on page 8–4.
• **Customizing an existing ADM class** allows you to extend or modify its behavior by modifying the custom files that are built into it. You do not create any new class files when you customize an existing class: you simply change the behavior of a class by adding Progress 4GL code to one or more of its custom class files. For details, see the “Customizing an existing ADM class” section on page 8–10.

• **Creating a new class at the middle of the ADM class hierarchy** can be done by inserting a new class between two existing classes. This approach allows you to change or extend the standard behavior of existing SmartObjects; however, you should be able to handle most extension needs by either creating a new class at the bottom of the hierarchy or customizing an existing class.

**Note:** Progress Software Corporation strongly recommends you do not create a new class in the middle of the ADM class hierarchy. It requires extensive knowledge of the ADM and entails significant risk of damaging your Progress installation.
Creating a new class at the bottom of the ADM class hierarchy

You use the AppBuilder’s New ADM Class tool to create a new ADM class. There are three steps in creating a new class:

- Considering and resolving planning issues for your new class
- Using the New ADM Class tool to create the class file
- Adding application logic to the class files as required to provide the custom behavior for your new class

This section describes planning issues and explains how to use the New ADM Class tool to create the class files. For instructions on adding application logic, see the “Adding application logic to your new or customized ADM class” section on page 8–12. For an example of creating a new class, see the “Creating the myviewer class” section on page 8–18.

Planning issues

Before you use the New ADM Class tool, consider some planning issues for questions that will arise during its use. Be sure to resolve these issues before you use this tool to create your new class:

- Decide where to put your new class files at design time. Since an ADM class is a collection of files, most of which are include files, and other files will reference the location of these include files, you might want to have a standard location in which to put all of the new class files. The New ADM Class tool (described later in this chapter) prompts you to enter the name of the directory where you want your new class files to be written. Progress Software Corporation recommends you follow the convention of the standard ADM class files and create an src\adm2 directory for your source files and an adm2 directory for your .r files in your working directory.

- Think about how to name your class files. The New ADM Class tool automatically generates the filenames based on the naming convention used for the standard ADM class files shipped with Progress. Progress Software Corporation recommends you use the same convention with your new class files.

- The New ADM Class tool automatically creates a custom subdirectory under the source directory you specify. This subdirectory will hold all the custom files or hooks that the standard ADM class files reference to allow the end user to customize the class. These files are a necessary part of the class structure, even if you do not plan to use them.

Remember you cannot change the filenames and relative paths of a class once the AppBuilder generates the files. You must be careful to create a development environment that is consistent with the environment and file structure in which you plan to deploy your applications.
Creating a new class at the bottom of the ADM class hierarchy

Creating the new class files

You create the new class files with the New ADM Class tool, located on the AppBuilder menu bar. To start the New ADM Class tool, choose Tools→New ADM Class... from the menu bar, shown in Figure 8–2.

Figure 8–2: New ADM Class menu option

Invoking this command opens the New ADM Class dialog box, shown in Figure 8–3.

Figure 8–3: New ADM Class (initial)

This dialog box contains two tabs. In the Basic tab, you name the class and supply various file and directory names and locations. The Custom Files tab provides information about the new custom files for this class. The following sections describe how to use these tabs.
The Basic tab

The Basic tab initially contains only one fill-in field, Name, in which you enter the name of your new class. After you specify a class name, fill-in fields open for the other labels on this tab. These fields contain automatically generated default names based on the class name you supply. For example, specifying the class name as mybrowser and the class from which to derive as . . . browser.cld produces the result shown in Figure 8–4.

![New ADM Class window](image)

Figure 8–4: Basic tab with fill-in fields

The default names follow the standard Progress naming convention. Progress Software Corporation recommends you use these names but does not require that you do so. Recall that once you save your new class, you cannot change the file and directory names for your new class.

Table 8–1 lists and describes the fill-in fields in this tab.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
<th>Mandatory?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name of the class. This name serves as a base name for the names of all of the files for the class.</td>
<td>Yes</td>
</tr>
<tr>
<td>Class Definition File</td>
<td>A file that references the components of a class. The extension must be .cld.</td>
<td>Yes</td>
</tr>
<tr>
<td>Source Directory</td>
<td>The pathname of the directory where the source files will be generated. You can browse for this directory.</td>
<td>Yes</td>
</tr>
<tr>
<td>Rcode Directory</td>
<td>The pathname of the directory for the r-code for the super procedure. You can browse for this directory.</td>
<td>Yes</td>
</tr>
<tr>
<td>Template Directory</td>
<td>The pathname of the directory for the template file. You can browse for this directory.</td>
<td>Mandatory if there is a value in the Template field</td>
</tr>
</tbody>
</table>
Table 8–1: Basic tab fill-in fields

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
<th>Mandatory?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derive From Class</td>
<td>The class definition file (.cld) to subclass.</td>
<td>No</td>
</tr>
<tr>
<td>Method Library</td>
<td>A file that defines the class name, references a property file, and starts the super procedure. The extension must be .i. This file is also called the primary include file.</td>
<td>Yes</td>
</tr>
<tr>
<td>Property File</td>
<td>A file that defines properties for the class. The extension must be .i.</td>
<td>Yes</td>
</tr>
<tr>
<td>Super Procedure</td>
<td>Defines the get and set functions for readable/writable properties and new behavior for the class. The extension must be .p.</td>
<td>Yes</td>
</tr>
<tr>
<td>Prototype File</td>
<td>A file that references the functions and internal procedures of a super procedure. The extension must be .i.</td>
<td>Yes</td>
</tr>
<tr>
<td>Note: You can create the contents of this file by using the ProtoGen tool from the AppBuilder’s ProTools palette.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Template</td>
<td>The name of the template file. It references the primary include file of the class.</td>
<td>No</td>
</tr>
<tr>
<td>Copy From Template</td>
<td>A template file to copy from. You can select the file on disk by clicking the file button.</td>
<td>No</td>
</tr>
</tbody>
</table>

Check the Replace existing files if exist option at the bottom of the Basic tab if you want to open these files in the AppBuilder once they have been created.
The Custom Files tab

The Custom Files tab displays all the custom files (hooks) that will be generated when the new class is created. Like the file and directory names in the Basic tab’s fill-in fields, these filenames are generated automatically based on the class name you supply, and thus they are not filled until after you specify a class name. For example, specifying the class name mybrowser in the Basic tab produces the result in the Custom Files tab shown in Figure 8–5.

Figure 8–5: Custom Files tab

The custom files allow you to modify and/or extend a class after it is deployed, without altering the basic class itself. They are described in more detail in the “Custom class files” section on page 8–12.

Creating the class files

Once you supply all the information required in the Basic tab, including specifying whether to open the files in the AppBuilder, you are almost ready to create the files.

Before you create them, however, you must decide whether the AppBuilder should replace files that already exist when it creates the class files. To specify this, either check or uncheck the Replace existing files if exist option at the bottom of the New ADM Class dialog box, as appropriate. You can now create the class by clicking OK. (To cancel the operation, click Cancel.)

When the AppBuilder creates a new class, it generates these files:

- Class definition file: class-name.cld
- Primary include file: prim-incl-file.i
- Property file: prop-file.i
- Prototype file: proto-file.i
- Super procedure: super-proc.p
Creating a new class at the bottom of the ADM class hierarchy

- **Template:** `template-file.w`
  (if specified in the **Basic** tab of the New ADM Class dialog box)

- **Custom primary include file:** `prim-incl-filecustom.i`

- **Custom property file:** `prop-filecustom.i`

- **Custom prototype file:** `proto-filecustom.i`

- **Custom super procedure:** `super-proccustom.p`

- **Custom exclude definition file:** `class-nameexclcustomm.i`

- **Custom instance definition file:** `class-namedefscustom.i`

- The r-code for the super procedures in the r-code directory.

You already should be familiar with the standard class files. For descriptions of them, see the “Adding application logic to your new or customized ADM class” section on page 8–12.

**Warnings and error messages**

When the AppBuilder tries to create a new class, it uses validation controls to check whether pathnames and filenames are correct and whether files already exist. If it is successful in creating the class, the AppBuilder advises you of this:

![Information dialog]

If the validation process fails, the AppBuilder displays a warning message and does not generate code. The following are warnings and error messages the AppBuilder might display:

- **Saving the Class Without a Class Name** — If you try to save the class without providing a class name, the AppBuilder displays the following error message:

![Information dialog]

- **Generating Your Files Directly in the DLC Directory** — Progress Software Corporation recommends you do not install user-defined code in the DLC directory. If you try to do this, the AppBuilder displays the following warning message:

![Warning dialog]
The warning gives you the option of creating a similar directory structure in your working directory (that is, `\src\adm2`). If you choose to do this, and any of the directories specified do not yet exist, you are prompted to create them:

If you choose not to create the directory, the AppBuilder stops processing and does not generate files. You must provide a valid directory name and then try again.

- **File Generation Error** — If an error occurs while the AppBuilder is generating class files, it displays a message and stops processing. For example:

If the AppBuilder stops processing during file generation, it does not delete files that it has finished generating.

### Modifying the new class files

After creating the class files for your new class, you modify the standard class files as required to provide behavior specific to your new class. You already should be familiar with the standard class files, which are described in detail in Chapter 1, “Overview.”

You modify the standard class files by adding Progress 4GL code that supplies the application logic for the class’ custom behavior. For details, see the “Adding application logic to your new or customized ADM class” section on page 8–12. For an example of creating a new class, see the “Creating the myviewer class” section on page 8–18.

### Customizing an existing ADM class

The custom class files built into a particular ADM class provide a way for you to add modifications and/or extensions without altering its standard class files. Most of the custom class files share essentially the same structure as the corresponding standard class files. The custom files allow you to alter an existing ADM class by:

- Adding new properties
- Adding a new super procedure
- Adding new behaviors
- Replacing or removing existing behaviors
- Adding new prototype definitions
- Adding new instance properties
All standard class files in the %DLC% (Windows) or $DLC$ (UNIX) directory under the adm2 and web2 subdirectories include custom hooks that allow their customization. Similarly, any new class that you create (or that is given to you) has these custom hooks built in.

For descriptions of the custom class files, as well as design rules and special notes on adding properties, see the “Adding application logic to your new or customized ADM class” section on page 8–12. For an example of customizing an existing class, see the “Customizing the visual class” section on page 8–23.
Adding application logic to your new or customized ADM class

Regardless of whether you are creating a new class or customizing an existing class, you must modify class files to provide the application logic for the new or customized class:

- If you are creating a new class, you modify the standard class files.
- If you are customizing an existing class, you modify the custom class files.

Although you are working on different sets of files, the process is similar: you modify the standard class files with the same sorts of changes you apply to the custom class files. This section describes the custom class files, then provides design rules and some special information on adding properties and writing super procedures. You should read the custom class files section even if you are modifying standard files, since you modify them similarly. For examples of both processes, see the “Examples” section on page 8–18.

Custom class files

The custom class files for a given ADM class are built into the class. These files contain a structure for the required application logic, as well as instructions for adding this code. The following list describes the custom class files, including for each file the filename format, the file type, and a discussion of the types of changes you can make with the file:

- Custom primary include file (filename: prim-incl-filecustom.i) — This include file is referenced in the standard primary include file. It contains the code to start the custom super procedure. This code is initially commented out; you must uncomment it manually once you develop your custom super procedure. You also can use this file to initialize properties.

- Custom property file (filename:prop-filecustom.i) — This include file is referenced in the standard property include file. It allows you to extend the behavior of the class by defining new properties.

- Custom super procedure (filename:super-proccustom.p) — This structured procedure file is referenced in the custom primary include file. It allows you to define internal functions and procedures to support new properties in the custom property file (that is, to define new behavior), and generally to extend or override the standard behavior of the class it belongs to:
  - To extend the standard behavior of a class, create a local version of a standard function that contains the statement SUPER( ) or procedure that contains the statement RUN SUPER, then add code lines before and/or after the RUN SUPER statement that provide additional behavior before and/or after the standard behavior.
  - To override a particular behavior, remove the standard behavior by defining an EXCLUDE-{proc-or-func-name} preprocessor value in the custom exclude definitions file for this class. This causes the removal of the corresponding procedure or function from the standard super procedure file. You then define in the custom super procedure a procedure or function of your own that has the same name as the procedure or function that you are overriding. (The custom exclude definitions file is described later in this section.) Be sure to include the RUN SUPER or SUPER( ) statement if the procedure or function that you are placing has it.
• Custom prototype file (filename: *proto-filecustom.i*) — This include file is referenced in the custom property include file. It allows you to define the function and procedure prototypes to match the internal entries of the custom super procedure. These prototypes allow an Open4GL client (for example, a Java client) to identify the entry points of the corresponding SmartObject.

• Custom exclude definition file (filename: *class-nameexclcustom.i*) — This include file is referenced in the standard super procedure file. It allows you to exclude procedures and/or functions by defining an \texttt{EXCLUDE-\{ proc-or-func-name \}} preprocessor value. Each internal entry in a super procedure file is enclosed by special preprocessor code that checks for the existence of a corresponding \texttt{EXCLUDE-\{ proc-or-func-name \}} preprocessor value and removes the procedure or function from the compilation. You typically exclude a procedure or function so that you can rewrite it in the custom super procedure. The \texttt{EXCLUDE} mechanism is available not only for super procedures but also for any standard structured procedure created in the AppBuilder.

The standard class files do not contain a file corresponding to the custom exclude definition file.

• Custom instance definition file (filename: *class-nameinstcustom.i*) — This include file is referenced in the standard property include file. It allows you to modify the list of instance properties for the class, usually by adding properties. You can also use it to change the instance Properties dialog itself, by specifying the filename of a different instance properties dialog box.

The standard class files do not contain a file corresponding to the custom instance definition file.

### Design rules

This section summarizes the design rules for constructing a new SmartObject that inherits from the new class that you created. These rules are fairly simple; follow them to assure the right application structure. The design rules are:

• Each SmartObject type (template) includes a primary include file that, by convention, has the same name as the template but with a .i extension. For example, *viewer.w* includes \%DLC\%\src\adm2\viewer.i (Windows) or \$DLC/src/adm2/viewer.i (UNIX).

• Each primary include file such as *viewer.i* includes its parent’s support include file, plus its own property include file. For example, *viewer.i* includes *datavis.i*, because a SmartDataViewer is a visual object that displays data, and also *viewprop.i*, where the properties specific to SmartDataViewers are defined. It also starts the associated super procedure.

• Each super procedure file includes its own property include file. These nest in such a way as to define all object properties in the proper order.

In addition, the presence of super procedures means it is possible to run many internal procedures and user-defined functions in SmartObjects that are not in those objects, but only in their super procedures.
Each prototype file should include a prototype definition for each internal procedure and function implemented in a SmartObject’s super procedures. The contents of the prototype file are implementation dependent, so initially it is empty. Use the AppBuilder’s Pro*Tool ProtoGen to generate the prototype file based on your super procedure. For more information on the ProtoGen utility, see the online help.

Follow these rules when building a new SmartObject type, to ensure all the relationships are established.

**Adding properties**

When you extend the ADM, whether by creating a new class or by customizing an existing class, you define properties that support the special behavior of your class. If you are creating a new class, you define standard properties in the standard property file. If you are customizing an existing class, you define custom properties in the custom property file.

Within the relevant file, you define properties in the same way. The property definition section of these files appears near the end of the main block section of the file. It has the following format:

```plaintext
&IF "{&ADMSuper}";U = "";U &THEN
  ghADMProps:ADD-NEW-FIELD(prop-specs)
  ghADMProps:ADD-NEW-FIELD(prop-specs)
  ghADMProps:ADD-NEW-FIELD(prop-specs)
  . . .
&ENDIF
```

When adding properties to the property file for a new or customized class, consider defining any properties that will be accessed frequently and for which the best possible performance is important as direct access properties. Properties are stored in the ADMProps temp-table; *direct access properties* are accessed directly from the temp-table rather than using get/set functions. The lack of function-call overhead can significantly improve performance.

You specify that a property is a direct access property by defining a preprocessor constant of the form `xppropname` for it. The preprocessor constant definition section is in the main block of the property file, just before the property definition section. The definitions have this format:

```plaintext
&GLOBAL-DEFINE xppropname
```

Sometimes you might want to read or write a direct access property from outside its own class; however, the ADMProps temp-table can be accessed directly only from objects of its own class and super procedure. To work around this problem, you must define get and/or set functions (as needed) that can be called from outside the class but that internally access the required property directly. This special-purpose get function has the following format:

```plaintext
FUNCTION getpropname RETURNS LOGICAL ( ):
  DEFINE VARIABLE tpropname AS LOGICAL NO-UNDO.
  {get propname tpropname}.
  RETURN tpropname.
END FUNCTION.
```
Its companion set function has the following format:

```plaintext
FUNCTION setproppname RETURNS LOGICAL (pxvariable AS data-type):
  {set proppname pxvariable}.
RETURN TRUE.
END FUNCTION.
```

**Writing super procedures**

Another task you perform when you extend the ADM, whether by creating a new class or by customizing an existing class, is writing super procedures; that is, adding to the super procedure file custom functions and/or internal procedures that support the special behavior of your class. If you are creating a new class, you do this for the standard super procedure file. If you are customizing an existing class, you do it for the custom super procedure file.

Two important points to keep in mind when writing any functions or internal procedures that go into an ADM super procedure are:

- The functions and internal procedures in an ADM super procedure must always refer to the procedure on whose behalf they run.
- An ADM super procedure is designed to be shared among all running instances of the SmartObjects that use that super procedure.

**References in super procedures to other procedures**

When you write the functions or internal procedures that go into an ADM super procedure, keep in mind there is nothing about a super procedure that identifies it as a super procedure. It is simply a Progress procedure file that is run as a persistent procedure and then added as a super procedure to some other running procedure. The source code or compiled r-code for a super procedure contains nothing special that identifies it as a super procedure.

What is special about a super procedure is the style of the code within its functions and internal procedures, which must account for the fact that these entry points are run on behalf of some other procedure. ADM super procedures, therefore, always must refer to the procedure on whose behalf they run. The basic mechanism is the `TARGET-PROCEDURE` built-in function, which evaluates to the procedure handle of the procedure from which the entry point originally was invoked. The `{get}` and `{set}` include files, which provide a pseudo-syntax for getting and setting ADM properties, support this mechanism by always referring back to the `TARGET-PROCEDURE`, so you always should use `{get}` and `{set}` when writing property references in super procedures. In addition, these include files identify whether a given property can be accessed directly out of the ADMProps temp-table record in the `TARGET-PROCEDURE` or instead requires the execution of the actual `getpropname` or `setpropname` function. For detailed information on these property mechanisms, see the “`{get}` and `{set}` pseudo-syntax for object properties” section on page 5–17, and the “Get and set functions for object properties” section on page 5–16.

For example, an internal procedure that operates on the RowObject buffer handle of its TARGET begins by retrieving that property value:

```plaintext
DEFINE VARIABLE hRowObject AS HANDLE NO-UNDO.
...
{get Rowobject hRowObject}.
```
To retrieve the value of a property from an object other than the TARGET-PROCEDURE, you can specify an optional final argument for the \{get\} and \{set\} include files that is the object’s procedure handle. For example, the following code gets the value of the EnabledTables property from the Data-Source of the TARGET-PROCEDURE:

\[
\begin{align*}
\{\text{get DataSource hSource}\}. \\
\{\text{get EnabledTables cTables hSource}\}.
\end{align*}
\]

You also can use the include file \{fn\} as a convenient way to invoke a function in TARGET-PROCEDURE; for example:

\[
\{\text{fn openQuery}\} \quad /* \text{Invoke the openQuery function in my TARGET-PROCEDURE */}
\]

Similarly, the variant \{fnarg\} invokes a function that takes a single argument; for example:

\[
/* \text{Invoke the colValues function in my TARGET-PROCEDURE, and pass } \\
\text{pcViewColList as the one input parameter. */}
\]

\[
\{\text{fnarg colValues pcViewColList}\}.
\]

**Shared super procedures**

ADM super procedures are designed to be shared among all running instances of the SmartObjects that use that procedure.

**Note:** It is possible to create a super procedure intended to serve exactly one running procedure at a time, but this is not the correct design for super procedures that are part of the ADM class hierarchy and that are started in the standard way (that is, by running start-super-proc in the ADM include files).

Because ADM super procedures are shared, you must ensure their code never assumes continuity between calls. If a particular operation involves executing two entry points in a super procedure, the code should not share property values between calls (whether by using local variables defined in the Definitions section or by using another technique); instead, it should retrieve any property values needed at the top of each entry point. Although this might seem slightly inefficient, using \{get\} to retrieve a property value from the property temp-table compiles into a single in-line Progress 4GL ASSIGN statement and is quite fast. Using this technique assures your super procedures can be shared among all running instances of your new class.

Likewise, because super procedures operate on behalf of other procedures, you always must PUBLISH events FROM TARGET-PROCEDURE. If you do not do this, your event normally will not have any effect, because the SmartObjects will not have subscribed to that event in the super procedure itself.
Also, when you invoke one entry point in a super procedure from another, you must preserve the value of TARGET-PROCEDURE in the called entry point. For example, suppose your super procedure contains two procedures, ProcA and ProcB, and you want to run ProcB from ProcA in that super procedure. You have basically two choices, depending on whether ProcB is a PRIVATE internal procedure or function that should never be run from any other procedure, or an entry point you might want to run or to override from its TARGET-PROCEDURE or another procedure:

- If ProcB is a PRIVATE internal procedure or function, one that should never be run from any other procedure, ProcA can run ProcB directly. In this case, if ProcB must refer to the target procedure (to get a property value, for example):
  - TARGET-PROCEDURE must be passed as an input parameter to ProcB.
  - ProcB must not use \{get\} or \{set\} to access the property.

  This is because the value of TARGET-PROCEDURE inside ProcB will not be TARGET-PROCEDURE—it will be the super procedure’s procedure handle, since it was executed directly from the super procedure. In such a case, it is normally better to pass any property values to ProcB from ProcA. The following two examples, which illustrate code in ProcA, show two ways to do this:

  ```plaintext
  RUN procB (INPUT TARGET-PROCEDURE).
  procB:
  DEFINE INPUT PARAMETER hTarget AS HANDLE NO-UNDO.
  DEFINE VARIABLE cDisplayedFields AS CHARACTER NO-UNDO.
  cDisplayedFields = dynamic-function('getDisplayedFields' in hTarget).
  ```

  ```plaintext
  DEFINE VARIABLE cDisplayedFields AS CHARACTER NO-UNDO.
  {get DisplayedFields cDisplayedFields}.
  RUN procB (INPUT cDisplayedFields).
  ```

- If ProcB is an entry point that you might want to run or override from its TARGET-PROCEDURE or another procedure, ProcA should invoke ProcB IN TARGET-PROCEDURE even though they are located in the same super procedure. In this way, references to TARGET-PROCEDURE inside ProcB correctly evaluate to the handle of the SmartObject that caused ProcA to be invoked. (This also makes it possible to override ProcB in a SmartObject, if this is necessary and appropriate.) The following example, which illustrates code in ProcA, shows how to do this:

  ```plaintext
  RUN procB IN TARGET-PROCEDURE.
  procB:
  DEFINE VARIABLE cDisplayedFields AS CHARACTER NO-UNDO.
  /* This works because procB was invoked in TARGET-PROCEDURE. */
  {get DisplayedFields cDisplayedFields}.
  ```
Examples

This section provides two examples. The first illustrates creating a new class with the New ADM Class tool; the second illustrates customizing an existing class.

Creating the myviewer class

The example in this section illustrates creating a new class based on the viewer class. The new class, called myviewer, will contain custom functionality that displays a warning message to an application user who wants to delete a record and allows the user to choose how to proceed. This functionality will be built into all SmartObjects based on this class.

The following sections describe how to create the new myviewer class files, then modify them to create the new functionality.

Note: The example assumes you installed Progress in the D:\PROGRESS directory.

Creating the myviewer class files

To create the myviewer class files:

1. From the AppBuilder main menu, choose Tools→New ADM Class...:

   ![New ADM Class dialog box](image)

   The New ADM Class dialog box opens, with the Basic tab in front.
2. In the Name field, enter myviewer, the new class name:

When you type myviewer, the AppBuilder enters default directories in the directory fill-in fields and names based on myviewer in many of the filename fill-in fields; for example, myviprop.i for the property filename.

3. Select the class from which to derive the myviewer class:

   a. Click on the Class button next to the Derive From Class field. In the Choose Class Definition dialog box that appears, you are placed by default in the %DLC%/src/adm2 (Windows) or $DLC/src/adm2 (UNIX) directory.

   b. Select the viewer.cld class definition file in that directory, then choose Open.

The AppBuilder adds information to the New ADM Class dialog box:
The **Derive From Class** field now contains the `viewer.cld` class definition file. The the **Copy From Template** field automatically is filled with the filename of the viewer template. This happens only because the class definition file contains a definition of this template; if it did not, this field would stay empty.

4. Click the **Custom Files** tab to note the custom class files that will be generated for the `myviewer` class:

![Custom Files Tab Image]

5. Click the **Basic** tab, then check the **Open files in the AppBuilder once generated** check box. This instructs the AppBuilder to open the primary include file, the property file, the super procedure file, and the prototype file after creating the class files:

![Basic Tab Image]
6. Press **OK** to create the files.

7. After the AppBuilder finishes creating the myviewer class files, close the primary include file (**myviewer.i**) and the prototype file (**myviprto.i**). In this example, these files do not need modification.

**Modifying the myviewer standard class files**

You use the following new code to implement the custom myviewer functionality: a `DeleteMessage` property, a `setDeleteMessage` function, and an override procedure for the standard ADM `deleteRecord` method. Adding this functionality requires changes only to the `myviprop.i` property file and the `myviewer.p` super procedure file, both of which should still be open in the AppBuilder.

To add the new functionality in the AppBuilder:

1. Define the new `DeleteMessage` property in the `myviprop.i` property file.

   To do this, go to the file’s main block and add the code shown in **bold typeface** below:

   ```
   /* Include the file that defines prototypes for all of the super procedure's entry points. And skip including the prototypes if we are "any" super procedure. */
   &IF "{&ADMSuper}"="":U &THEN
   {src/adm2/myviprto.i}
   &ENDIF

   /* Put your xp{&Property} preprocessor definition here. Use the following format, e.g. &GLOBAL-DEFINE xpMyProperty. These preprocessors tell at compile time which properties can be retrieved directly from the temp-table */
   &GLOBAL-DEFINE xpDeleteMessage
   {src/adm2/viewprop.i}
   &IF "{&ADMSuper}"="":U &THEN
   /* Put your property field definition here. Use the following syntax, e.g. ghADMProps:ADD-NEW-FIELD('MyProperty':U, 'CHAR':U, ,'x(20)':U, 'Hi':U). */
   ghADMProps:ADD-NEW-FIELD('DeleteMessage':U, 'CHAR':U, 0, ?, 'Are you sure you want to delete this record?':U).
   &ENDIF
   
   {src/adm2/custom/myvipropcustom.i}
   ```

   When you finish making these changes, click on the Save button (the diskette icon) on the AppBuilder main window to save your changes, then close the property file window.
2. Create a new `setDeleteMessage` function in the `myviewer.p` super procedure file. This function allows you to set the new `DeleteMessage` property. To create the new function, add the code shown in **bold typeface**:

```
RETURNS LOGICAL ( INPUT pcDeleteMessage AS CHARACTER ) :
/*-------------------------------------------------------------------
Purpose: 
Notes: 
-------------------------------------------------------------------*/
{set DeleteMessage pcDeleteMessage}.
RETURN TRUE. /* Function return value. */
END FUNCTION.
```

3. Create a new override procedure for the `deleteRecord` method in the `myviewer.p` super procedure file. This procedure displays an alert box when the `deleteRecord` method is invoked; this alert box prompts the user with a message that corresponds to the value of the new `DeleteMessage` property: if the user chooses Yes, the ADM deleteRecord executes, otherwise it returns without executing. To create the new override procedure, add the code shown in **bold typeface**:

```
/{*---------------------------------------------------------------
Purpose: 
Parameters:  <none>
Notes: 
---------------------------------------------------------------*/
DEFINE VARIABLE cDeleteMessage AS CHARACTER   NO-UNDO.
{get DeleteMessage cDeleteMessage}.
MESSAGE cDeleteMessage
   VIEW-AS ALERT-BOX QUESTION BUTTONS YES-NO
   UPDATE lAnswer AS LOGICAL.
IF lAnswer THEN
   RUN SUPER.
END PROCEDURE.
```

When you finish making the changes in **Step 2** and **Step 3**, click on the Save button (the diskette icon) on the AppBuilder main window to save your changes, then close the super procedure file window. The AppBuilder now creates a `.r` version of the super procedure in the `src\adm2` directory under your working directory.

4. Enable the new super procedure file by moving the `.r` file from the `src\adm2` directory under your working directory into the `adm2` directory under your working directory.

**Note:** If you do not move the `.r` file into the `adm2` directory under your working directory, you will not be able to use the new super procedure.
5. Make the template for the myviewer class available on the New dialog box that appears when you choose File→New from the AppBuilder menu. To do this:
   
a. Create a new .cst file and add to it the following code lines:

   ```
   *NEW-SMARTOBJECT SmartDataViewer Demo
   NEW-TEMPLATE src/adm2/template/myviewer.w
   ```

   b. Put the .cst file into the src\adm2\template directory under your working directory.

   c. Load the new .cst file by choosing Menu→Use Custom... from the AppBuilder Palette window and adding the name of the new file to the list.

The new class, with its custom behavior, is now ready for use. Before you work with it, you should restart the AppBuilder to re-establish the super procedure stack.

**Customizing the visual class**

The example in this section illustrates customizing the visual class. The customization defines a special property that you can set to specify the background color for SmartObjects that inherit from this class (that is, all visual SmartObjects), thus standardizing the look of your Progress 4GL applications.

The following sections note which custom class files you need to provide, and describe how to modify them to create the new functionality.

**Copying the visual custom class files**

You customize an existing class by modifying the custom class files built into that class. For this example, you will modify a selected set of the visual custom class files. You work on a copy of the required files in your working directory rather than in their installed location.

To prepare for customization:

Copy the following files from the %DLC%\src\adm2\custom (Windows) or $DLC/src/adm2/custom (UNIX) directory into the src\adm2\custom (Windows) or src/adm2/custom directory under your working directory:

- vispropcustom.i
- visualcustom.p
- visualdefscustom.i

**Modifying the visual custom class files**

You use the following new code to customize the visual class with the new functionality: a BgColor property, setBgColor and getBgColor functions, and an override procedure for the standard ADM initializeObject method. Adding this functionality requires changes to the visualcustom.i primary include file, the vispropcustom.i property file, the visualcustom.p super procedure file, and the visualdefscustom.i custom instance definition file, which you will copy and open as required.
To add the new functionality in the AppBuilder:

1. Define the BgColor property in the vispropcustom.i custom property file in your working directory. To do this, open the file, go to its main block, and add the code shown in bold typeface:

```plaintext
/* Include the file that defines prototypes for all of the super procedure's entry points. And skip including the prototypes if we are *any* super procedure. */
&IF "{&ADMSuper}”:U EQ “”:U &THEN
{src/adm2/custom/visprotocustom.i}
&ENDIF

&GLOBAL-DEFINE xpBgColor
&IF "{&ADMSuper}”:U = "":U &THEN
&ENDIF

/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME

When you finish making these changes, click on the Save button (the diskette icon) on the AppBuilder main window to save your changes, then close the property file window.

2. Create new setBgColor and getBgColor functions in the visualcustom.p super procedure file. These functions allows you to set and get the new Bgcolor property publicly. To create the new functions, open the file, then add the code shown in bold typeface:

```plaintext
FUNCTION getBgColor RETURNS INTEGER
  ( /* parameter-definitions */ ) :
  /*---*/
  DEFINE VARIABLE iBgColor AS INTEGER NO-UNDO.
  {get BgColor iBgColor}.
  RETURN iBgColor. /* Function return value. */
END FUNCTION.

FUNCTION setBgColor RETURNS LOGICAL
  ( INPUT piBgColor AS INTEGER ) :
  /*---*/
  {set BgColor piBgColor}.
  RETURN TRUE. /* Function return value. */
END FUNCTION.
```
3. Create an override procedure for the `initializeObject` method in the `visualcustom.p` custom super procedure file in your working directory. This procedure gets the current value of the new `BgColor` property and sets the background color of container frame accordingly. To create the new override procedure, add the code shown in **bold typeface**:

```plaintext
PROCEDURE initializeObject :
  /*-------------------------------------------------------------------
  Purpose:
  Parameters:  <none>
  Notes:  
  /*-------------------------------------------------------------------*/
  DEFINE VARIABLE hFrame AS HANDLE NO-UNDO.
  DEFINE VARIABLE iBgColor AS INTEGER NO-UNDO.
  {get ContainerHandle hFrame}.
  {get BgColor iBgColor}.
  RUN SUPER.
  /* Don't do anything in design mode */
  IF DYNAMIC-FUNCTION ('getUIBMode':U IN TARGET-PROCEDURE) NE "";U THEN
    RETURN.
  IF hFrame:TYPE = "FRAME";U THEN
    hFrame:BGCOLOR = iBgColor.
  ELSE IF hFrame:TYPE = "WINDOW";U THEN
    ASSIGN
      hFrame = hFrame:FIRST-CHILD
      hFrame:BGCOLOR = iBgColor.
  END PROCEDURE.
```

When you finish making the changes in **Step 2** and **Step 3**, click on the Save button (the diskette icon) on the AppBuilder main window to save your changes, then close the super procedure file window. The AppBuilder now creates a .r version of the super procedure in the `src\adm2` directory under your working directory.

4. Edit the `visualcustom.i` primary include file, and uncomment the line that starts the custom super procedure. When you finish making this change, click on the Save button (the diskette icon) on the AppBuilder main window to save your change, then close the primary include file window.
5. Add the BgColor property to the list of instance properties, and specify a filename for the instance properties dialog in the visualdefscustom.i custom instance definition file in your working directory. To add the property and filename, open the file, then add the code shown in **bold typeface**:

```plaintext
File : visualdefscustom.i
Purpose : Entry point to define:
    New definitions
    {&xcInstanceProperties}
    {&ADM-PROPERTY-DLG}
Syntax : {src/adm2/custom/visualdefscustom.i}
Modified : 06/03/1999
Notes : Referenced in {src/adm2/visprop.i}
*---------------------------------------------------------------------*/
&IF "{&xcInstanceProperties}";U NE "";U &THEN
    &GLOBAL-DEFINE xcInstanceProperties {&xcInstanceProperties},
&ENDIF
&GLOBAL-DEFINE xcInstanceProperties {&xcInstanceProperties} BgColor
&GLOBAL-DEFINE ADM-PROPERTY-DLG adm2/support/myvisuald.w
```

You have specified adm2/support/visuald.w—that is, the file visuald.w in the adm2\support directory under your working directory—as the instance properties dialog box file. (Progress code is not sensitive to whether you use forward or reverse slashes in pathnames.) You will add support for this dialog box in the next step.

When you finish making these changes, click on the Save button (the diskette icon) on the AppBuilder main window to save your changes, then close the property file window.

6. Provide support for your new property in the instance properties dialog box. In this example, the simplest approach is to copy the instance property dialog box for the visual class, visuald.w, from the %DLC%\src\adm2\support directory to src\adm2\support (Windows) or from the $DLC/src/adm2/support directory to src/adm2/support (UNIX) under your working directory, then modify the copy as needed.

Alternatively, you can either modify the original visuald.w instance properties dialog box in the %DLC%\src\adm2\support (Windows) or $DLC/src/adm2/support (UNIX) directory or create a completely new instance properties dialog box that includes support for the new property (as well as all other properties required by the class).

Regardless of which technique you use, you must be sure that the instance property dialog box filename matches the name that you specified in the custom instance definition file in your working directory. (See Step 5.)

7. Enable the new super procedure file by moving the .r file from the src\adm2 directory under your working directory into the adm2 directory under your working directory.

**Note:** If you do not move the .r file into the adm2 directory under your working directory, you will not be able to use the new super procedure.

The customized class, with its new property, is now ready for use. Before you work with it, you should restart the AppBuilder to re-establish the super procedure stack.
ADM Standard and Custom Class Files

All class files of a particular type have the same basic structure, regardless of which class they support. For example, the standard property file for the viewer class has the same basic structure as the browser property file, though the specifics—for example, which properties are defined—differ.

This appendix contains the following section:

- Container class files

It shows the contents of the container class files, as a way of illustrating the basic structures of all class files. For specifics on other classes, see their class files in the following directories:

- Standard class files: %DLC%\src\adm2 (Windows) or $DLC/src/adm2 (UNIX)
- Template files: %DLC%\src\adm2\template (Windows) or $DLC/src/adm2/template (UNIX)
- Custom class files: %DLC%\src\adm2\custom (Windows) or $DLC/src/adm2/custom (UNIX)
Container class files

The container class consists of a set of standard class files, three template files, and a set of custom class files. All ADM classes contain a parallel set of standard and custom class files; however, not all classes contain template files.

The following sections show the contents of the container class files, as well as one of its templates.

Standard class files

The container class has the following standard class files, located in the %DLC%/src/adm2 (Windows) or $DLC/src/adm2 (UNIX) directory:

- `containr.i`: primary include file
- `cntnprop.i`: property file
- `containr.p`: super procedure file
- `cntnprto.i`: prototype file

The `containr.i` primary include file follows:

```plaintext
&ANALYZE-SUSPEND _VERSION-NUMBER AB_v9r12
&ANALYZE-RESUME
&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _DEFINITIONS Method-Library
/*-------------------------------------------------------------------------
Library     : containr.i
Purpose     : Default V9 Main Block code and Method Procedures
              for ADM Container procedures.
Syntax      : {src/adm2/containr.i}
Modified     : May 19, 1999 Version 9.1A
-------------------------------------------------------------------------*/
```
**Container class files**

```plaintext
/***********************  DEFINITIONS  ***********************************/
&IF "{&ADMClass}":U = "":U &THEN
 &GLOB ADMClass containr
 &ENDIF
&IF "{&ADMClass}":U = "containr":U &THEN
 {src/adm2/cntnprop.i}
 &ENDIF
/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME

&ANALYZE-SUSPEND _UIB-PREPROCESSOR-BLOCK
/* ********************  Preprocessor Definitions  *******************/
/* _UIB-PREPROCESSOR-BLOCK-END */
&ANALYZE-RESUME
/* ******************** Procedure Settings  *************************/
&ANALYZE-SUSPEND _PROCEDURE-SETTINGS
/* Settings for THIS-PROCEDURE
 Type: Method-Library
 Allow:
 Frames: 0
 Add Fields to: Neither
 Other Settings: INCLUDE-ONLY
*/
&ANALYZE-RESUME _END-PROCEDURE-SETTINGS
/* ******************** Create Window  *******************************/
&ANALYZE-SUSPEND _CREATE-WINDOW
/* DESIGN window definition (used by the UIB)
 CREATE WINDOW Method-Library ASSIGN
  HEIGHT = 8
  WIDTH  = 60.
/* END WINDOW DEFINITION */
*/
```
The cntnprop.i property file follows:

```plaintext
/* ************************* Included-Libraries *************************** */
{src/adm2/visual.i}
/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME

/* ***************************  Main Block *************************** */
/* Best default for GUI applications - this will apply to the whole session: */
PAUSE 0 BEFORE-HIDE.

/* _ADM-CODE-BLOCK-START _CUSTOM _INCLUDED-LIB-CUSTOM CUSTOM */
{src/adm2/custom/containrcustom.i}
/* _ADM-CODE-BLOCK-END */
/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
```

The cntnprop.i property file follows:

```plaintext`
&ANALYZE-RESUME
&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _INCLUDED-LIB Method-Library
/* ********** Included-Libraries ********** */
{src/adm2/visual.i}
/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME

&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _MAIN-BLOCK Method-Library
/* ********** Main Block ********** */
RUN start-super-proc("adm2/contain.r.p":U).
/* Best default for GUI applications - this will apply to the whole session: */
PAUSE 0 BEFORE-HIDE.

/* _ADM-CODE-BLOCK-START _CUSTOM _INCLUDED-LIB-CUSTOM CUSTOM */
{src/adm2/custom/containrcustom.i}
/* _ADM-CODE-BLOCK-END */
/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
```
/*          This .W file was created with the Progress UIB.             */
/*---------------------------------------------------------------------*/
/* ********************** Definitions ******************************* */
{src/adm2/custom/containdefscustom.i}
/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
&ANALYZE-SUSPEND _UIB-PREPROCESSOR-BLOCK
/* ******************* Preprocessor Definitions ********************* */
/* _UIB-PREPROCESSOR-BLOCK-END */
&ANALYZE-RESUME
/* ****************** Procedure Settings ************************** */
&ANALYZE-SUSPEND _PROCEDURE-SETTINGS
/* Settings for THIS-PROCEDURE */
/* Type: Include */
/* Allow: */
/* Frames: 0 */
/* Add Fields to: Neither */
/* Other Settings: INCLUDE-ONLY */
/* */
&ANALYZE-RESUME _END-PROCEDURE-SETTINGS
/* **************************************** Create Window ****************** */
&ANALYZE-SUSPEND _CREATE-WINDOW
/* DESIGN Window definition (used by the UIB) */
/* CREATE WINDOW Include ASSIGN */
/* HEIGHT = 8 */
/* WIDTH = 60. */
/* END WINDOW DEFINITION */
/* */
&ANALYZE-RESUME
&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _MAIN-BLOCK Include
/* ***********************************  Main Block    *********************************** */

/* Include the file which defines prototypes for all of the super procedure's entry points. Also, start or attach to the super procedure. Skip start-super-proc if we *are* the super procedure. And skip including the prototypes if we are *any* super procedure. */
&IF 

&ENDIF

/* Preprocs to identify to compiler which properties are in the temp-table.*/
&GLOB xpCurrentPage
&GLOB xpContainerTarget
&GLOB xpContainerTargetEvents
&GLOB xpPageNTarget
&GLOB xpPageSource
&GLOB xpFilterSource
&GLOB xpDataTarget
&GLOB xpUpdateSource
&GLOB xpUpdateTarget
&GLOB xpStartPage
&GLOB xpRunMultiple

/* Now include the next-level-up property include file. This builds up the property temp-table definition, which we will then add our field definitions to. */
{src/adm2/visprop.i}

&IF 

&ENDIF

{src/adm2/custom/cntnpropcustom.i}

/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
An abridged version of the containr.p super procedure file follows; the complete file is quite long:

```
containr.p

&ANALYZE-SUSPEND _VERSION-NUMBER AB_v9r12
&ANALYZE-RESUME
&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _DEFINITIONS Procedure
/*-------------------------------------------------------------------------
File : containr.p
Purpose : Code common to all containers, including WIndows and Frames
Syntax : adm2/containr.p
Modified : July 22, 1999 Version 9.1A
-------------------------------------------------------------------------*/
/*          This .W file was created with the Progress UIB.             */
/*---------------------------------------------------------------------*/
/* ********************************** Definitions ********************************** */
/* Tell cntnattr.i that this is the Super Procedure */
&SCOP ADMSuper containr.p
  {src/adm2/custom/containrexclcustom.i}
DEFINE VARIABLE giPrevPage AS INTEGER NO-UNDO.
/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
&ANALYZE-SUSPEND _UIB-PREPROCESSOR-BLOCK
/* ********************************** Preprocessor Definitions ********************************** */
&Scoped-define PROCEDURE-TYPE Procedure
&Scoped-define DB-AWARE no
/* _UIB-PREPROCESSOR-BLOCK-END */
&ANALYZE-RESUME
```
Function Prototypes

IFDEF DEFINED(EXCLUDE-getContainerTarget) = 0 &THEN
&ANALYZE-SUSPEND _UIB-CODE-BLOCK _FUNCTION-FORWARD getContainerTarget
Procedure
FUNCTION getContainerTarget RETURNS CHARACTER
   ( ) FORWARD.
/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
&ENDIF.

[Similar prototypes for the following functions:
getContainerTargetEvents    pageNTargets
getCurrentPage              setContainerTarget
getDataTarget               setDataTarget
getFilterSource             setFilterSource
getPageNTarget              setPageNTarget
getPageSource               setPageSource
getRunMultiple              setRunMultiple
getUpdateSource             setUpdateSource
getUpdateTarget             setUpdateTarget]

Procedure Settings

&ANALYZE-SUSPEND _PROCEDURE-SETTINGS
/* Settings for THIS-PROCEDURE
Type: Procedure
Allow:
Frames: 0
Add Fields to: Neither
Other Settings: CODE-ONLY COMPILE
*/
&ANALYZE-RESUME _END-PROCEDURE-SETTINGS
/* ****************************  Create Window  ************************** */

&ANALYZE-SUSPEND _CREATE-WINDOW
/* DESIGN Window definition (used by the UIB)
 CREATE WINDOW Procedure ASSIGN
  HEIGHT             = 15
  WIDTH              = 60.
/* END WINDOW DEFINITION */

&ANALYZE-RESUME

&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _INCLUDED-LIB Procedure
/* **************************** Included-Libraries **************************** */

{src/adm2/cntnprop.i}

&ANALYZE-RESUME

&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _MAIN-BLOCK Procedure
/* **************************** Main Block  ******************************* */

&ANALYZE-RESUME

/* **************************** Internal Procedures **************************** */

&IF DEFINED(EXCLUDE-assignPageProperty) = 0 &THEN

&ANALYZE-SUSPEND _UIB-CODE-BLOCK _PROCEDURE assignPageProperty Procedure
PROCEDURE assignPageProperty :
/*-------------------------------------------------------------------------
Purpose:  Sets the specified property in all objects on the CurrentPage of
calling SmartContainer.

Params:  INPUT pcProp AS CHARACTER -- property to set.
         INPUT pcValue AS CHARACTER -- value to assign to that property.
         This is specified in CHARACTER form but can be used to
         assign values to non-character properties.

Notes:  This variation on assignLinkProperty is necessary because
the notion of paging does not fit well with PUBLISH/SUBSCRIBE.
All objects in a Container will subscribe to initializeObject, etc.,
but the paging performs the operation on subsets of those objects
*/
at a time. That is, the container will not publish 'initializeObject' to objects on a page other than zero until that page is first viewed. So properties such as HideOnInit which are set as part of initialization must be set page-by-page.

```
DEFINE INPUT PARAMETER pcProp AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER pcValue AS CHARACTER NO-UNDO.
DEFINE VARIABLE iVar AS INTEGER NO-UNDO.
DEFINE VARIABLE cObjects AS CHARACTER NO-UNDO.

{get CurrentPage iVar}.
cObjects = pageNTargets(TARGET-PROCEDURE, iVar).

DO iVar = 1 TO NUM-ENTRIES(cObjects):
  DYNAMIC-FUNCTION("set":U + pcProp IN WIDGET-HANDLE(ENTRY(iVar, cObjects)),
                   pcValue) NO-ERROR.
END.
RETURN.
END PROCEDURE.
/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
&ENDIF

[Similar code for the following procedures:
  changePage       initPages
  confirmExit      notifyPage
  constructObject  passThrough
  createObjects    removePageNTarget
  deletePage       selectPage
  initializeObject viewObject
  viewPage]
/* ********************  Function Implementations  ****************** */

&IF DEFINED(EXCLUDE-getContainerTarget) = 0 &THEN
&ANALYZE-SUSPEND _UIB-CODE-BLOCK _FUNCTION getContainerTarget Procedure
FUNCTION getContainerTarget RETURNS CHARACTER (  ) :
/*----------------------------------------------------------
Purpose: Returns a list of the handles of the object's contained objects.
Params: <none>
----------------------------------------------------------*/

DEFINE VARIABLE cTarget AS CHARACTER NO-UNDO.
{get ContainerTarget cTarget}.
RETURN cTarget.

END FUNCTION.

/*@ _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME

&ENDIF

[Similar code for the following functions:
  getContainerTargetEvents  pageNTargets
  getCurrentPage    setContainerTarget
  getDataTarget    setDataTarget
  getFilterSource    setFilterSource
  getPageNTarget    setPageNTarget
  getPageSource    setPageSource
  getRunMultiple    setRunMultiple
  getUpdateSource    setUpdateSource
  getUpdateTarget    setUpdateTarget]
The `cntnprto.i` prototype file follows:

```plaintext
/*
 * Prototype include file: C:\adm90\src\adm2\cntnprto.i
 * Created from procedure: C:\adm90\src\adm2\containr.p at 09:47 on 10/27/98
 * by the PROGRESS PRO*Tools Prototype Include File Generator
 */

/***** Start excluded prototypes

PROCEDURE viewPage IN SUPER:
    DEFINE INPUT PARAMETER piPageNum AS INTEGER.
END PROCEDURE.

PROCEDURE selectPage IN SUPER:
    DEFINE INPUT PARAMETER piPageNum AS INTEGER.
END PROCEDURE.

PROCEDURE removePageNTarget IN SUPER:
    DEFINE INPUT PARAMETER phTarget AS HANDLE.
    DEFINE INPUT PARAMETER piPage AS INTEGER.
END PROCEDURE.

PROCEDURE notifyPage IN SUPER:
    DEFINE INPUT PARAMETER pcProc AS CHARACTER.
END PROCEDURE.

PROCEDURE initPages IN SUPER:
    DEFINE INPUT PARAMETER pcPageList AS CHARACTER.
END PROCEDURE.

PROCEDURE initializeObject IN SUPER:
END PROCEDURE.

PROCEDURE deletePage IN SUPER:
    DEFINE INPUT PARAMETER piPageNum AS INTEGER.
END PROCEDURE.
```
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createObjects</td>
<td>PROCEDURE createObjects IN SUPER:</td>
</tr>
<tr>
<td>constructObject</td>
<td>PROCEDURE constructObject IN SUPER:</td>
</tr>
<tr>
<td>confirmExit</td>
<td>PROCEDURE confirmExit IN SUPER:</td>
</tr>
<tr>
<td>changePage</td>
<td>PROCEDURE changePage IN SUPER:</td>
</tr>
<tr>
<td>assignPageProperty</td>
<td>PROCEDURE assignPageProperty IN SUPER:</td>
</tr>
<tr>
<td>start-super-proc</td>
<td>PROCEDURE start-super-proc IN SUPER:</td>
</tr>
<tr>
<td>getContainerTarget</td>
<td>FUNCTION getContainerTarget RETURNS CHARACTER IN SUPER.</td>
</tr>
<tr>
<td>getContainerTargetEvents</td>
<td>FUNCTION getContainerTargetEvents RETURNS CHARACTER IN SUPER.</td>
</tr>
<tr>
<td>getDataTarget</td>
<td>FUNCTION getDataTarget RETURNS CHARACTER IN SUPER.</td>
</tr>
<tr>
<td>getPageNTarget</td>
<td>FUNCTION getPageNTarget RETURNS CHARACTER IN SUPER.</td>
</tr>
<tr>
<td>getPageSource</td>
<td>FUNCTION getPageSource RETURNS HANDLE IN SUPER.</td>
</tr>
<tr>
<td>pageNTargets</td>
<td>FUNCTION pageNTargets RETURNS CHARACTER</td>
</tr>
<tr>
<td></td>
<td>(INPUT phTarget AS HANDLE,</td>
</tr>
<tr>
<td></td>
<td>INPUT piPageNum AS INTEGER) IN SUPER.</td>
</tr>
<tr>
<td>setContainerTarget</td>
<td>FUNCTION setContainerTarget RETURNS LOGICAL</td>
</tr>
<tr>
<td></td>
<td>(INPUT pcObject AS CHARACTER) IN SUPER.</td>
</tr>
<tr>
<td>setDataTarget</td>
<td>FUNCTION setDataTarget RETURNS LOGICAL</td>
</tr>
<tr>
<td></td>
<td>(INPUT pcTarget AS CHARACTER) IN SUPER.</td>
</tr>
<tr>
<td>setPageNTarget</td>
<td>FUNCTION setPageNTarget RETURNS LOGICAL</td>
</tr>
<tr>
<td></td>
<td>(INPUT pcObject AS CHARACTER) IN SUPER.</td>
</tr>
<tr>
<td>setPageSource</td>
<td>FUNCTION setPageSource RETURNS LOGICAL</td>
</tr>
<tr>
<td></td>
<td>(INPUT phObject AS HANDLE) IN SUPER.</td>
</tr>
<tr>
<td>setCurrentPage</td>
<td>FUNCTION setCurrentPage RETURNS INTEGER</td>
</tr>
</tbody>
</table>
Templates

The container class has the following templates, located in the %DLC%\src\adm2\template (Windows) or $DLC/src/adm2/template (UNIX) directory:

- `cntnrdlg.w`: dialog box template.
- `cntnrfrm.w`: frame template.
- `cntnrwin.w`: window template.

The `cntnrfrm.w` frame template follows (the other templates are similar to this):

```plaintext
&ANALYZE-SUSPEND _VERSION-NUMBER AB_v9r12 GUI ADM2
/* Procedure Description
"ADM2 SmartFrame Object Template

Use this template to create a new frame which supports SmartObjects. Draw your SmartObjects on this container and establish the appropriate SmartLinks to connect them."
*/
&ANALYZE-RESUME
&Scoped-define WINDOW-NAME CURRENT-WINDOW
&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _DEFINITIONS fFrameWin
/*------------------------------------------------------------------------
File:
Description: from cntnrfrm.w - ADM2 SmartFrame Template
Input Parameters:
 <none>
Output Parameters:
 <none>
------------------------------------------------------------------------*/
/* This .W file was created with the Progress AppBuilder. */
```
/* Create an unnamed pool to store all the widgets created
   by this procedure. This is a good default which assures
   that this procedure's triggers and internal procedures
   will execute in this procedure's storage, and that proper
   cleanup will occur on deletion of the procedure. */
CREATE WIDGET-POOL.

/* ***************************  Definitions  ************************** */
/* Parameters Definitions --- */
/* Local Variable Definitions --- */
/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME

&ANALYZE-SUSPEND _UIB-PREPROCESSOR-BLOCK

/* ***********************  Control Definitions  ********************** */
/* Custom List Definitions */
/* _UIB-PREPROCESSOR-BLOCK-END */
&ANALYZE-RESUME

/* ***********************  Control Definitions  ********************** */
/* ************************  Frame Definitions  *********************** */
DEFINE FRAME fMain
  WITH 1 DOWN NO-BOX KEEP-TAB-ORDER OVERLAY
  SIDE-LABELS NO-UNDERLINE THREE-D
  AT COL 1 ROW 1
  SIZE 79.86 BY 11.92.

/* *********************** Procedure Settings ************************ */
&ANALYZE-SUSPEND _PROCEDURE-SETTINGS
/* Settings for THIS-PROCEDURE
Type: SmartFrame Template
Allow: Basic,Browse,DB-Fields,Query,Smart
Container Links:
Data-Target,Data-Source,Page-Target,Update-Source,Update-Target
Other Settings: PERSISTENT-ONLY
*/

/* This procedure should always be RUN PERSISTENT. Report the error, */
/* then cleanup and return. */
IF NOT THIS-PROCEDURE:PERSISTENT THEN DO:
  MESSAGE "{&FILE-NAME} should only be RUN PERSISTENT."
  VIEW-AS ALERT-BOX ERROR BUTTONS OK.
  RETURN.
END.

&ANALYZE-RESUME _END-PROCEDURE-SETTINGS

/* *********************** Create Window ******************************* */
&ANALYZE-SUSPEND _CREATE-WINDOW
/* DESIGN Window definition (used by the UIB)
CREATE WINDOW fFrameWin ASSIGN
  HEIGHT   = 11.91
  WIDTH    = 79.8.
/* END WINDOW DEFINITION */
*/
A SmartFrame instantiates and initializes the SmartObject instances it contains. At initialization, the SmartFrame sets the position of the instances and links them together.

USING A SMARTFRAME

Step 1
Build your desired SmartObject masters, save, and close them.

Step 2
Draw instances of the SmartObject masters into the SmartFrame.

Step 3
Add all necessary SmartLinks between SmartObject instances.

Note: During assembly, the PROGRESS Advisor suggests links and creates them for you. However, you can also add and remove SmartLinks with the SmartLinks dialog box. To access this dialog, choose the Procedure button from the UIB main window. Then choose the SmartLinks button from the Procedure Settings dialog box.

/* Actions: adecomm/_so-cue.w ? adecomm/_so-cued.p ? adecomm/_so-cuew.p */
/* ***********  Runtime Attributes and AppBuilder Settings *********** */

&ANALYZE-SUSPEND _RUN-TIME-ATTRIBUTES
/* SETTINGS FOR WINDOW fFrameWin
   VISIBLE,,RUN-PERSISTENT */
/* SETTINGS FOR FRAME fMain
   NOT-VISIBLE */
ASSIGN
   FRAME fMain:HIDDEN = TRUE.
/* _RUN-TIME-ATTRIBUTES-END */
&ANALYZE-RESUME

/* Setting information for Queries and Browse Widgets fields */
&ANALYZE-SUSPEND _QUERY-BLOCK FRAME fMain
/* Query rebuild information for FRAME fMain
   _Options = ""
   _Query is NOT OPENED */ /* FRAME fMain */
&ANALYZE-RESUME
&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _MAIN-BLOCK fFrameWin
/* ***************************  Main Block  *************************** */
&IF DEFINED(UIB_IS_RUNNING) <> 0 &THEN
   /* Now enable the interface if in test mode - otherwise this happens when
    * the object is explicitly initialized from its container. */
   RUN initializeObject.
&ENDIF
/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
/* **********************  Internal Procedures  *********************** */

&ANALYZE-SUSPEND _UIB-CODE-BLOCK _PROCEDURE adm-create-objects fFrameWin
_ANM-CREATE-OBJECTS
PROCEDURE adm-create-objects :
/*---------------------------------------------------------------*/
  Purpose: Create handles for all SmartObjects used in this procedure.
           After SmartObjects are initialized, then SmartLinks are added.
  Parameters: <none>
---------------------------------------------------------------*/
END PROCEDURE.

/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME

&ANALYZE-SUSPEND _UIB-CODE-BLOCK _PROCEDURE disable_UI fFrameWin
_DEFAULT-DISABLE
PROCEDURE disable_UI :
/*---------------------------------------------------------------*/
  Purpose: DISABLE the User Interface
  Parameters: <none>
  Notes: Here we clean-up the user-interface by deleting
dynamic widgets we have created and/or hide
frames. This procedure is usually called when
we are ready to "clean-up" after running.
---------------------------------------------------------------*/
/* Hide all frames. */
HIDE FRAME fMain.
IF THIS-PROCEDURE:PERSISTENT THEN DELETE PROCEDURE THIS-PROCEDURE.
END PROCEDURE.

/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME

&ANALYZE-SUSPEND _UIB-CODE-BLOCK _PROCEDURE enable_UI fFrameWin
_DEFAULT-ENABLE
PROCEDURE enable_UI :
/*---------------------------------------------------------------*/
  Purpose: ENABLE the User Interface
  Parameters: <none>
  Notes: Here we display/view/enable the widgets in the
user-interface. In addition, OPEN all queries
associated with each FRAME and BROWSE.
These statements here are based on the "Other
Settings" section of the widget Property Sheets.
---------------------------------------------------------------*/
{&OPEN-BROWSERS-IN-QUERY-fMain}
END PROCEDURE.

/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
Custom class files

The container class has the following custom class files, located in the %DLC%\src\adm2\custom (Windows) or $DLC/src/adm2/custom (UNIX) directory:

- containrcustom.i: custom primary include file
- cntnpropcustom.i: custom property file
- containrcustom.p: custom super procedure file
- cntnprtocustom.i: custom prototype file
- containrexclcustom.i: custom exclude definition file
- containrexclcustom.i: custom instance definition file

The containrcustom.i custom primary include file follows:

```
&ANALYZE-SUSPEND _VERSION-NUMBER UIB_v9r12
&ANALYZE-RESUME
&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _DEFINITIONS Include
/*------------------------------------------------------------------------
File        : containrcustom.i
Purpose     : References the start of the custom super procedure.
              Allows properties initialization.
Syntax      : {src/adm2/custom/containrcustom.i}
Description :
Created     : 06/03/1999
Notes       : Referenced in {src/adm2/containr.i}
----------------------------------------------------------------------*/
/*          This .W file was created with the Progress AppBuilder.      */
/*----------------------------------------------------------------------*/
/* ***************************  Definitions  ************************** */
/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
&ANALYZE-SUSPEND _UIB-PREPROCESSOR-BLOCK
```
/* ********************  Preprocessor Definitions  ******************** */

/* _UIB-PREPROCESSOR-BLOCK-END */
&ANALYZE-RESUME

/* *********************** Procedure Settings ************************ */

&ANALYZE-SUSPEND _PROCEDURE-SETTINGS
/* Settings for THIS-PROCEDURE */
Type: Include
Allow:
Frames: 0
Add Fields to: Neither
Other Settings: INCLUDE-ONLY
*/&ANALYZE-RESUME _END-PROCEDURE-SETTINGS

/* *************************  Create Window  ************************** */

&ANALYZE-SUSPEND _CREATE-WINDOW
/* DESIGN Window definition (used by the UIB) */
CREATE WINDOW Include ASSIGN
    HEIGHT             = 15
    WIDTH              = 60.
/* END WINDOW DEFINITION */

/* ***************************  Main Block  *************************** */
/* Starts here the custom super procedure */
Uncomment to run it */

/*RUN start-super-proc ("adm2/custom/containrcustom.p":U).*/

/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
The `cntnpropcustom.i` custom property file follows:

```
&ANALYZE-SUSPEND _VERSION-NUMBER UIB_v8r12
&ANALYZE-RESUME
&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _DEFINITIONS Include
/****************************************************************************
File    : cntnpropcustom.i
Purpose : Defines basic properties.
Syntax  : {src/adm2/custom/cntnpropcustom.i}
>Description : Modified : 06/03/1999
Notes   : Referenced in {src/adm2/cntnprop.i}
*/
/*          This .W file was created with the Progress UIB.             */
/*--------------------------------------------------------------------*/
/*_UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
&ANALYZE-SUSPEND _UIB-PREPROCESSOR-BLOCK
/****************************************************************************
Preprocessor Definitions ***************************************************/
/*_UIB-PREPROCESSOR-BLOCK-END */
&ANALYZE-RESUME
/****************************************************************************
Procedure Settings *******************************************************/
&ANALYZE-SUSPEND _PROCEDURE-SETTINGS
/****************************************************************************
Settings for THIS-PROCEDURE
Type: Include
  Allow:
  Frames: 0
  Add Fields to: Neither
  Other Settings: INCLUDE-ONLY
*/
&ANALYZE-RESUME _END-PROCEDURE-SETTINGS
```
/* ******************** Preprocessor Definitions ********************** */

/* _UIB-PREPROCESSOR-BLOCK-END */
&ANALYZE-RESUME

/* *********************** Procedure Settings ************************ */
&ANALYZE-SUSPEND _PROCEDURE-SETTINGS
/* Settings for THIS-PROCEDURE */
/* Type: Include */
/* Allow: */
/* Frames: 0 */
/* Other Settings: INCLUDE-ONLY */
&ANALYZE-RESUME _END-PROCEDURE-SETTINGS

/* ************************* Create Window ************************** */
&ANALYZE-SUSPEND _CREATE-WINDOW
/* DESIGN Window definition (used by the UIB) */
CREATE WINDOW Include ASSIGN
    HEIGHT = 15
    WIDTH = 60.
/* END WINDOW DEFINITION */
&ANALYZE-RESUME

&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _MAIN-BLOCK Include

/* *************************** Main Block *************************** */
/* Starts here the custom super procedure */
/* Uncomment to run it */
/*RUN start-super-proc ("adm2/custom/containrcustom.p":U).*/

/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
The containrcustom.p custom super procedure file follows:

```
containrcustom.p

&ANALYZE-SUSPEND _VERSION-NUMBER AB_v9r12
&ANALYZE-RESUME
&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _DEFINITIONS Procedure
/*-------------------------------------*/
  File        : containrcustom.p
  Purpose     : Super procedure to extend containr class.
  Syntax      : containrcustom.p
  Modified    : 06/03/1999
/*-------------------------------------*/
/* This .W file was created with the Progress UIB. */
/*-------------------------------------*/
/* ******************************************** Definitions ******************************************** */
&SCOPED-DEFINE ADMSuper containrcustom.p

/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
&ANALYZE-SUSPEND _UIB-PREPROCESSOR-BLOCK
/* ******************************************** Preprocessor Definitions ******************************************** */
&Scoped-define PROCEDURE-TYPE Procedure
&Scoped-define DB-AWARE no
/* _UIB-PREPROCESSOR-BLOCK-END */
&ANALYZE-RESUME
/* ******************************************** Procedure Settings ******************************************** */
&ANALYZE-SUSPEND _PROCEDURE-SETTINGS
/* Settings for THIS-PROCEDURE*/
  Type: Procedure
  Allow:
  Frames: 0
  Add Fields to: Neither
```
Other Settings: CODE-ONLY COMPIL

&ANALYZE-RESUME _END-PROCEDURE-SETTINGS

/ * *********************** Create Window *********************** */

&ANALYZE-SUSPEND _CREATE-WINDOW
/* DESIGN Window definition (used by the UIB)
   CREATE WINDOW Procedure ASSIGN
   HEIGHT = 15
   WIDTH = 60.
/* END WINDOW DEFINITION */

* /
&ANALYZE-RESUME

&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _INCLUDED-LIB Procedure
/* ************************ Including Libraries ************************ */

{src/adm2/cntnprop.i}

/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME

&ANALYZE-SUSPEND _UIB-CODE-BLOCK _CUSTOM _MAIN-BLOCK Procedure
/* ************************* Main Block ************************** */

/* _UIB-CODE-BLOCK-END */
&ANALYZE-RESUME
The `cntnprtocustom.i` custom prototype file follows:

```plaintext
/*
 * Prototype include file: src/adm2/custom/cntnprtocustom.i
 * Created from procedure: src/adm2/custom/containrcustom.p at 20:11:24 on 06/03/1999
 * by the PROGRESS PRO*Tools Prototype Include File Generator
 */
```

The `containrexclcustom.i` custom exclude definition file follows:

```plaintext
/*-----------------------------------------------------------------------
File        : containrexclcustom.i
Purpose     : Defines EXCLUDE-xxx preprocessor variables to exclude functions or internal procedures in the super procedure where it is referenced.
Syntax      : {src/adm2/custom/containrexclcustom.i}
Modified    : 06/03/1999
Notes       : Referenced in src/adm2/containr.p
-----------------------------------------------------------------------*/
```

The `containrdefscustom.i` custom instance definition file follows:

```plaintext
/*-----------------------------------------------------------------------
File        : containrdefscustom.i
Purpose     : Entry point to define:
    New definitions
    {&xcInstanceProperties}
    {&ADM-PROPERTY-DLG}
Syntax      : {src/adm2/custom/containrdefscustom.i}
Modified    : 06/03/1999
Notes       : Referenced in {src/adm2/cntnprop.i}
-----------------------------------------------------------------------*/
```
ADM1 to ADM2 Conversion Utility

There is a utility to convert SmartObjects from ADM1 (Versions 8.1 and 8.2) to ADM2 (Version 9 and later). This appendix describes how to use this utility. It contains the following sections:

- Conversion support for ADM SmartObjects in later versions
- Using the ADM1 to ADM2 conversion utility

Note: You cannot convert Version 8.0 SmartObjects directly to ADM2 SmartObjects. Instead, you must first convert them to Version 8.1/8.2 SmartObjects using the Version 8.0 to Version 8.1/8.2 SmartObject conversion utility. (You can elect to add the icon for this utility to the PRO*Tools palette.) Then you can convert them to ADM2 SmartObjects as described in this appendix.

Throughout this appendix, “ADM1” refers to Versions 8.1 and 8.2, and “ADM2” (or “ADM”) refers to Version 9 and OpenEdge Release 10.
Conversion support for ADM SmartObjects in later versions

OpenEdge provides a SmartObject conversion utility that aids in converting ADM1 objects to ADM2 objects. Because conversion is optional (recompiled Version 8 SmartObjects will work in later versions), carefully consider whether you have specific development or application needs that require you to convert objects. If you need to convert, make sure to devise a conversion strategy to accomplish your tasks, before beginning the conversion. At a minimum, your conversion strategy should address how you will use the conversion utility and how you will test your final conversion results. Both aspects are critical to the conversion process.

**Caution:** If you customized any part of the ADM1 method libraries, added local procedures to SmartObjects, added code to a SmartObject instance, or made other customizations, Progress Software Corporation does not guarantee that converted code will work as it did in Version 8. In fact, the conversion utility cannot completely convert most SmartObjects due to the lack of necessary information: a fully functional Version 8 SmartViewer does not contain the information necessary to build a fully functional OpenEdge SmartDataViewer, and the same is true for other SmartObjects.

Progress Software Corporation recommends you use the conversion utility only as an aid, with the understanding you will need to manually adjust your customized SmartObject applications well beyond what the conversion utility provides. Progress Software Technical Support provides only general, high-level assistance for the conversion of SmartObject applications, not code fixes. For detailed assistance with converting SmartObjects, contact Progress Software Consulting Services.

You can employ many different strategies to attempt your conversion activities; however, Progress Software Corporation recommends that as a first step, you consider converting one window and all its components. Controlling the conversion process in this way helps to identify any difficulties you might encounter with your conversion attempts on a limited scale. This approach also helps to assess your approach to any subsequent conversions.

Testing also is a critical component of the conversion strategy. Progress Software Corporation recommends you define a full complement of test routines to ensure your conversion results meet your expectations.

Keep in mind the conversion utility always requires supplemental human intervention to complete the conversion tasks. The utility does not support a one-for-one conversion of files, nor can it be completely effective in converting files; consequently, you must examine the results of your conversion attempts closely and, as described later in this section, take appropriate steps to facilitate the completion of the conversion process manually.

When you create an application, be aware that ADM1 and ADM2 objects cannot coexist in the same design window; at the window level, all objects must be of the same type. Therefore, if you previously created ADM1 objects and now want to use them in OpenEdge with ADM2 objects, consider attempting to convert your ADM1 objects and perform the necessary manual conversion-related tasks.
Code changes to SmartObject container type objects

All SmartObjects have include files that are built into their templates; for example {src/adm2/container.i}. In OpenEdge, the ADM has similar include files, but they have different content and different names.

Additional miscellaneous points

The following points refer to SmartObject code in Smart Container files:

- ADM1 Smart Containers have procedures named row-available and send-records. These procedures do not appear in an ADM2 SmartObject, because ADM2 SmartObjects pass values instead of records.

- The ADM1 procedure state-changed is not in the ADM2 SmartObjects, because of the new PUBLISH/SUBSCRIBE event model.

- Local-exit is replaced by exitObject.
Using the ADM1 to ADM2 conversion utility

Keep the following points in mind as you review this section:

- How you employ the conversion utility depends on your conversion strategy; for example, whether you attempt to convert all files at a single time, or choose to attempt only one directory and its related subdirectories at a time.

- Converting a file means the file is processed by the conversion utility and an attempt is made to convert the file to the ADM2 format; however, the utility cannot convert all files successfully. You will need to examine your conversion attempts and results, and manually perform additional changes to complete the conversion process. How little or how much manual clean-up work is required for a given conversion attempt varies; to determine this adequately, you must examine the results and perform test routines. For information about manual conversions, see the “Errors and the conversion utility log file” section on page B–10 and the “Customizing the conversion utility” section on page B–12.

This section discusses the following topics:

- Accessing the conversion utility
- Defining conversion options
- Building the list of files to process
- Performing the conversion (including the conversion process and errors and the conversion utility log file)
- Reviewing conversion results
- Customizing the conversion utility

**Note:** Progress Software Corporation strongly advises you back up all files before performing any of the procedures described in this section.

Accessing the conversion utility

You access the conversion utility from the PRO*Tools Palette, by clicking on the SmartObject conversion utility icon, as shown in Figure B–1.
This opens the **ADM1 to ADM2 SmartObject Conversion Utility** window, as shown in Figure B–2.

![ADM1 to ADM2 SmartObject Conversion Utility window](image)

**Figure B–2:** ADM1 to ADM2 SmartObject Conversion Utility window

This window has two main sections: an upper portion that defines directory and conversion options and a lower portion that displays specific file information relative to the conversion process.

**Defining conversion options**

In the upper portion of the **Conversion Utility** window, you specify file directory information, file filter specifications, and conversion options:

![Conversion Utility window](image)

Table B–1 lists and describes the objects you use to set conversion options.

<table>
<thead>
<tr>
<th>Table B–1: Setting up file conversion options</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
</tr>
<tr>
<td><strong>Directory</strong> fill-in field</td>
</tr>
<tr>
<td><strong>Filter</strong> combo box</td>
</tr>
</tbody>
</table>
To set up a conversion option:

1. In the Directory fill-in field, specify the directory of the application to be converted.

   The directory you choose to specify depends on your conversion strategy. For example, you might decide to convert an entire application, and therefore specify the root directory of the application. Alternatively, your strategy might be to specify only one directory at a time to convert.

   **Note:** Any directory you intend to convert should be either part of the PROPATH or a subdirectory of a PROPATH directory.

2. Using the Include Subdirectories check box, indicate whether to convert files in the subdirectories (or any subdirectories at any lower levels) of the directory identified in Step 1.

3. Using the Compile after Conversion check box, indicate whether to compile each file after the conversion attempt.

4. In the Filter combo box, specify the types of files to process.

   Typically, you convert only .w files. This is because you are converting Version 8 SmartObjects, which normally have the .w extension.
Building the list of files to process

In addition to the objects noted in the previous section, the upper portion of the ADM1 to ADM2 SmartObject Conversion Utility window contains the **Press the Build File List** button. This button populates the browser located in the lower portion of the window according to the directory, filter, and conversion specifications you set, as described in the previous section. For example:

After pressing this button, be sure to review the files that display before you begin the actual conversion. The **Files to Convert** browser contains two display-only fields:

- **Status** — Identifies the current status of a file. Status labels are described later in this section. The **Not Conv** status means the file was not processed since it was entered into the list.

- **Name** — Identifies the name of the file to be converted.

The amount of time required for the build process varies according to the number of files being prepared. If the build process takes several minutes (a likely situation when loading thousands of files), the browse query is opened to display the first screen full of files, and an overlay frame appears to identify the current directory that is being loaded and the current number of files being loaded.

**Table B–2** summarizes the various activities that you can perform based on the file information in the **Files to Convert** browser.

**Table B–2: Files in the browser**

<table>
<thead>
<tr>
<th>Status</th>
<th>Course of action</th>
</tr>
</thead>
<tbody>
<tr>
<td>No further file changes are required.</td>
<td>Continue with the information in this section and bypass the information in the “Refining the file selections” section on page B–8.</td>
</tr>
<tr>
<td>You need to modify (add or delete) files in the browser.</td>
<td>Go to the “Refining the file selections” section on page B–8 and return to this section when the changes are complete.</td>
</tr>
</tbody>
</table>
Refining the file selections

Before converting, you can refine your file selection using any of the three additional options that are available by using the three buttons to the right of the browser. Table B–3 identifies these options and the functionality they support.

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort...</td>
<td>Changes the displayed sort order of the files in the browser from the Sort Options dialog box. The sort options are Status, Name, and Extension. The sort order also determines the order in which the files are processed.</td>
</tr>
<tr>
<td>Add a File...</td>
<td>Adds individual files, one at a time, to the browser from the System File dialog box. (This is the only way to add files once the list of files to process has been built.)</td>
</tr>
<tr>
<td>Remove</td>
<td>Removes one or more files from the browser. You select the files and then press the Remove button.</td>
</tr>
</tbody>
</table>

Performing the conversion

Progress Software Corporation strongly recommends that before you attempt a conversion as described in this section, you make a backup copy of all source files. This backup copy should be separate from and in addition to the original source files that are saved into the V8-ADM directory structure as described Step 1 below.

Press the Start Conversions button to initiate file conversion according to the sort sequence that you have established. You can convert only files whose status is shown as Not Conv.

Note: The conversion utility does not require the AppBuilder to be running because the utility converts by parsing the code; the files are not loaded into the AppBuilder. However, the application databases should be connected.
The Conversion Process

The conversion utility performs the following steps:

1. The utility moves the files from their current directory structure to a similar structure with a root name V8-ADM, V8-ADM is a subdirectory of the root directory originally defined in the Directory fill-in field in the upper portion of the utility window. This structure always is created; if it already exists, it is overwritten.

   **Note:** As previously noted, it is essential that you make a backup copy of all source files before attempting a conversion.

2. As each file is converted, the utility changes its status from Not Conv to ..., and concludes with ###-changes.

3. If an error occurs during the conversion process, the utility displays an error alert box that identifies the problem and prompts you to either stop the process (that is, do not attempt to perform any more file conversions) or continue with more files. Regardless of your choice, the utility attempts to convert the file that caused the error and marks its code with an &MESSAGE statement. The statement indicates that a problem occurred during the conversion of the file and notes that the file must be adjusted manually. (Most files require some manual intervention after conversion.) This message displays each time you compile the file, flagging it as untrustworthy until you correct the problem manually.

4. Once the conversion is complete, the utility replaces each original file that was copied to the V8-ADM directory structure with the corresponding converted file.

   **Note:** To abort conversion processing, press the Abort Conversions button. The conversion utility finishes processing only the current file, stopping before the next scheduled file.

Figure B–3 shows how the Conversion Utility window might look once the conversion process is complete.

*Figure B–3: Conversion Utility window after a conversion*
Table B–4 identifies and describes possible status labels in this window.

Table B–4: Conversion status labels displayed in the browser

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No-change</td>
<td>The file was converted, but no changes were made.</td>
</tr>
<tr>
<td>...</td>
<td>The conversion process is taking place.</td>
</tr>
<tr>
<td>###-changes</td>
<td>The file was converted and some number of changes were made. (### represents a numeric value that identifies how many conversion-related changes were made to a given file.)</td>
</tr>
<tr>
<td>Comp.Err.</td>
<td>After the conversion attempt, the compiler found some errors.</td>
</tr>
<tr>
<td>Version 9</td>
<td>The conversion utility determined that the file was either previously converted or was built as a Version 9 file, and therefore no conversion attempt was made.</td>
</tr>
</tbody>
</table>

Errors and the conversion utility log file

The conversion utility cannot handle all coding styles and possible code constructs. These situations require manual intervention. To facilitate this, the utility generates a log file named V89conv.log in your working directory. This log identifies what procedures were converted, approximately how many changes were made to the file, whether the file was compiled, and if the compiler encountered errors. Also, if the conversion utility determines it had difficulty converting a procedure, this log tries to capture that information and report where the problem occurred. For information on modifying the conversion utility, see the “Customizing the conversion utility” section on page B–12.

Reviewing conversion results

This section describes the specific changes that occur when converting ADM1 SmartObjects to ADM2 SmartObjects. The changes it describes include ADM1-to-ADM2 SmartObject conversions, specific ADM file-conversion changes, and additional conversion changes.

Table B–5 lists the ADM1-to-ADM2 SmartObject file conversions the conversion utility tries to perform.

Table B–5: ADM1 to ADM2 SmartObject conversions

<table>
<thead>
<tr>
<th>ADM1 SmartObject</th>
<th>Corresponding ADM2 SmartObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADM1 SmartWindow</td>
<td>ADM2 SmartWindow</td>
</tr>
<tr>
<td>ADM1 SmartDialog</td>
<td>ADM2 SmartDialog</td>
</tr>
<tr>
<td>ADM1 SmartFrame</td>
<td>ADM2 SmartFrame</td>
</tr>
<tr>
<td>ADM1 SmartBrowser</td>
<td>ADM2 SmartDataBrowser</td>
</tr>
<tr>
<td>ADM1 SmartQuery</td>
<td>ADM2 SmartDataObject</td>
</tr>
<tr>
<td>ADM1 SmartViewer</td>
<td>ADM2 SmartDataViewer</td>
</tr>
</tbody>
</table>
Table B–6 identifies specific ADM1-to-ADM2 file conversion changes.

**Table B–6: Specific file conversion changes**

<table>
<thead>
<tr>
<th>For this element . . .</th>
<th>All . . .</th>
<th>Are . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUN DISPATCH</td>
<td>RUN dispatch [ IN hd1 ] ( [ INPUT ] 'proc').</td>
<td>Converted to: RUN proc [ IN hd1 ] Or, if it is a local copy of itself: RUN SUPER</td>
</tr>
<tr>
<td></td>
<td>RUN notify [ IN hd1 ] ( [ INPUT ] 'proc').</td>
<td>Converted to: PUBLISH 'proc' [ FROM hd1 ]</td>
</tr>
<tr>
<td>GET-ATTRIBUTE</td>
<td>RUN get-attribute [ IN hd1 ] ( [ INPUT ] 'attr'). var=RETURN-VALUE.</td>
<td>Converted to: var=[hd1:]getpropname ( )</td>
</tr>
<tr>
<td>SET-ATTRIBUTE-LIST</td>
<td>RUN set-attribute-list ( [ INPUT ] 'attr1=val1, attr2=val2, ... ').</td>
<td>Replaced by: {setpropname1 val1}, {setpropname2 val2}</td>
</tr>
</tbody>
</table>

A variable that contains a list is not converted. You must write your own get and set property functions for any customized attributes.

Some common ADM1 attribute names automatically are converted to the corresponding ADM2 property names (for example, Current-Page to Current Page), based on entries in the file protools/v89names.dat. You can extend this list as described in the “Customizing the conversion utility” section on page B–12.

This list identifies operations that the conversion utility performs for all converted files:

- Removes the following prefixes from procedures and procedure dispatches:
  - adm-.
  - local-.
  - broker-.
  - Any custom prefix defined in protools/convt89.p variable cCustomPrefix.
- Replaces all strings found in the ++ Old Name column of protools/v89names.dat file with the corresponding string in the ++ New Name column.
- Removes all procedures and attribute references found in the protools/v89names.dat file ++ Remove List.
- Replaces local-exit procedures with exitObject procedures.
- Edits the ADM-SUPPORTED-LINKS preprocessor to reflect the Version 9 link types.
• When TABLES-IN-QUERY... and ENABLED-TABLE-IN-QUERY... have more than one table, generates SECOND-TABLE-IN-QUERY..., SECOND-ENABLED-TABLE-IN-QUERY..., preprocessors up to the TENTH-TABLE....

• When converting a SmartViewer or SmartBrowser, generates an include file with a reasonably appropriate set of field definitions for the ADM2 SmartDataObject RowObject temp-table and flagged with an &MESSAGE statement to indicate that it should be replaced with an include file from an appropriate SmartDataObject.

• Removes external table definitions.

• If a procedure is dispatched inside of an instance of itself (for example, dispatching abc inside local-abc), converts this to RUN SUPER syntax.

• Converts Init-Object procedures to constructObject procedures. This includes the conversions of variable attributes to their ADM2 names. For example, Layout becomes ObjectLayout, Edge-Pixels becomes EdgePixels, SmartPanelType becomes PanelType, Right-to-Left becomes RightToLeft, and so on.

• Converts the procedure settings to the ADM2 format. These new formats are required by the AppBuilder to re-establish the correct state when reading the .w file.

• Creates appropriate field definitions in the Runtime Attribute Settings when converting a Version 8 SmartQuery to an ADM2 SmartDataObject.

• Converts states based on the values of the cStateVals and cStateProps variables in the _convt89.p procedure.

• Removes the entire Method Library section and replaces it with the appropriate Version 9 include files based on the type of object to which it is being converted. The utility determines this by reading the ADM1 template type found in the Definitions section, as shown below:

| Description: from VIEWER.W - Template for SmartViewer Objects |

If the utility does not find this line, the SmartObject cannot be properly converted. Be sure to restore the line in the Definitions section before attempting the conversion.

If a SmartObject has ADM1 custom method libraries you need to retain, you need to re-insert them after the conversion. Generally, you have to convert these custom method libraries, as they might not work with ADM2.

**Customizing the conversion utility**

This section identifies the conversion utility source code and describes why you might want to access it to make modifications.
The source code for the conversion utility is shipped with the tool. This code was written in the AppBuilder and should be maintained in the AppBuilder. The procedure `protools/convt89.p` performs the majority of the conversion tasks; it has no user interface. This procedure is called by the `protools/v89conv.w` dialog, which contains most of the conversion utility user interface. Much of the conversion process is table driven from the `protools/v89names.dat` file.

**Caution:** Carefully review the contents of the `protools/v89names.dat` file to ensure that you know how the conversion utility will change your code. You can edit this file to ensure your customized code is not removed. For example, if you customized the state-changed internal procedure and do not want this procedure to be removed during the conversion, make the appropriate edit in the `protools/v89names.dat` file.

Generally, the conversion utility can fulfill most of the user’s conversion needs. If you need to perform a few one-for-one substitutions, however, you can do this by creating more entries in the `++ Old Name  ++New Name` section of the `protools/v89names.dat` file.

**Note:** Use caution when performing these one-for-one substitutions. The conversion utility checks each line of code in each of the files to be converted for each of these entries. Consequently, if you convert 1000 files and each file contains an average of 600 lines of code, each entry in this section causes 600,000 lookups.

The `++ Remove List` section of the `protools/v89names.dat` file lists several attributes and procedures that are used in Version 8 but are obsolete in later versions. You can add to this list to remove occurrences of attributes and procedures that must be removed from the source files.

If you want to add or modify features in the conversion utility, you should study the code in `protools/convt89.p` and perform the changes. An easy change to make is to use custom prefixes (that is, prefixes other than `adm-` and `local-`) for your ADM1.1 methods. You should edit `protools/convt89.p` and initialize the variable `cCustomPrefix` with your own prefix. For example, if your prefix is `ajax-`, then you must give `cCustomPrefix` the initial value of `ajax-` (including the hyphen).

The `++ Method List` section contains four columns:

1. The first column contains the ADM1 method name.
2. The second column contains the name of the ADM2 equivalent.
3. The third column starts with `P` if the method remains a procedure or `FL` if it is converted to a function that returns a logical data type.
4. The fourth column contains parameter information for the method. It is a quoted string containing comma-delimited triplets, each of which includes the following components:
   a. The first component is the parameter name.
   b. The second component is the data type.
   c. The third component, which identifies the parameter type, is either `I` for `INPUT`, `O` for `OUTPUT`, or `I-O` for `INPUT-OUTPUT`.

The triplets themselves are delimited by the vertical bar (|) character.
Index

A

Accessing properties 5–20
SmartDataObjects from Java AppObjects 7–29
SmartObjects 2–4

Add Multiple Records instance property 2–28

Add One Record instance property 2–28

Adding properties to ADM extensions 8–14
SmartDataFields to SmartDataViewers 2–22, 2–25
SmartObjects to SmartDataViewers 2–22

addMessage procedure 5–7

addQueryWhere( ) function 7–8

ADM 1–2
advanced topics 6–1
architecture 1–2
class files 1–4
classes 1–4, 8–2
custom class files 1–6
data management 4–1
developing extensions 8–1
directory and file structures 5–2
properties 5–11
standard class files 1–5
templates 1–4, 1–6

ADM1 B–1

ADM2 B–1

adm2 directory 5–2

ADMProps temp-table 5–11

Advanced ADM topics 6–1

AppBuilder interactions with 6–10
modifying SmartDataObject masters 7–20

Application Development Model. See ADM

Application logic
adding to new or customized ADM class 8–12

Applications debugging 6–7
monitoring 6–6
SmartObject benefits 1–3

Architecture, ADM 1–2

ASDivision property 7–27

ASHandle property 7–27

ASInfo property 7–27

assignQuerySelection( ) function 7–7

ASUsePrompt property 7–27

B

Basic tab 8–6
fill-in fields 8–6

Batch-mode operations 7–28

BEGIN instance property 2–37
Browse instance property 2–34
Browse Keys instance property 2–47
Browse Label instance property 2–46
Browsing
data 4–2, 4–3
records 2–14
Buttons
on SmartPanels 2–26

C

Calculate Width instance property 2–16
Cancel on Exit of Browse instance property 2–45
Check Current Changed instance property 2–9
Checking compatibility 4–4

Class files 1–4
class files A–1
creating 8–8
custom 1–6, 8–12, A–20
relationships among 5–5
standard 1–5
templates 1–6, A–14
Classes 1–2, 1–4, 8–2
class files 1–4
custom class files 1–6
existing, customizing 8–3, 8–10
hierarchical structure 1–4
new, creating 8–3, 8–4
standard class files 1–5
templates 1–4, 1–6
Column instance property 2–37, 2–47
Column-level validation. See Field-level validation
columnQuerySelection( ) function 7–8
Columns
going and setting properties 7–11
Combo–box instance property 2–37
Commit instance property 2–41
Commit instance property 2–41
Commit SmartLinks 3–3, 3–10
events 3–10
Commit SmartPanels 2–26, 4–11
Committing multiple record updates 2–26
Compatibility checking 4–4
signatures and field lists 4–4
SmartDataBrowsers 4–5
SmartDataViewers 4–5
Compiling SmartDataObjects conditionally 7–20
Connecting to SmartFilters in other Windows 4–9
Container class
class files A–1
custom class files A–20
standard class files A–2
templates A–14
Container SmartLinks 3–2, 3–6
events 3–6
Container SmartObjects 2–5
CONTAINS instance property 2–37
Conversion Utility B–1
Creating
class files 8–8
new ADM classes 8–3, 8–4
adding application logic 8–12
example 8–18
modifying class files 8–10
New ADM Class tool 8–5
planning issues 8–4
SmartDataFields 2–24
CTRL–HOME trigger 2–18
Custom class files 1–6, 8–12, A–20
Custom exclude definition files 1–7, A–26
Custom Files tab 8–8
Custom instance definition files 1–7, A–26
Custom primary include files 1–6, A–20
Custom property files 1–6, A–22
Custom prototype files 1–7, A–26
Custom super procedures 1–7, A–24
Customizing
application logic for ADM extensions 8–12
existing ADM classes 8–3, 8–10
SmartFilters 4–8
SmartPanels 4–13
Index

D

Data
- browsing 4–2, 4–3
- committing multiple record updates 2–26
- displaying 4–2, 4–3
- filtering 2–32
- managing 4–1
- modifying records 2–26
- navigating records 2–26
- retrieving records 2–8
- signaling availability in queries 7–11
- transferring in queries 7–9
- updating records 2–26
- validation 7–13

Data Dictionary validation 2–22, 7–13

Data SmartLinks 3–3, 3–9, 7–12
- events 3–9
- retrieving records from multiple data sources 3–15
- using 3–17, 4–7

Data Source instance property 2–46, 2–47

Data sources
- retrieving records 2–8
- triggers 7–18

Data Type instance property 2–46

Database records. See Records

Debugger
- Progress 6–7

Default instance property 2–34

Define Any Key Trigger on Field instance property 2–47

Defining SmartLinks 3–18
- dynamic 3–19
- static 3–18

Destroy on each stateless Web Request instance property 2–9

destroyObject procedure 5–7

Dialog boxes 2–7
- instance property
  - dynamic SmartDataBrowser 2–14
  - Navigation SmartPanel 2–27
  - Simple SmartObject 2–51
  - SmartDataField 2–24
  - SmartDataObject 2–8
  - SmartDataViewer 2–21
  - SmartFilter 2–32
  - SmartFolder 2–30
  - SmartSelect 2–43

SmartToolBar 2–40
- Update SmartPanel 2–28
- New ADM Class 8–5

Direct access properties 8–14

Directories
- ADM 5–2
- adm2 5–2
- support 5–2
- template 5–2

Disconnect AppServer on each Web Request instance property 2–9

Display Field instance property 2–44

Display instance property 2–45

Displaying
- data 4–2, 4–3
- fields 2–23, 2–32
- records 2–14, 2–21

Distributed environments
- managing SmartDataObjects 7–23
- running SmartDataObjects 7–19

Down instance property 2–16

Dynamic properties 5–21

Dynamic SmartDataBrowsers 2–17
- compared to static SmartDataBrowsers 2–17
- instance properties 2–15
- instance property dialog box 2–14
- setting field properties 2–20

E

Edge Pixels instance property 2–27, 2–28

Edit Browse Columns instance property 2–45

Edit Displayed field list instance property 2–45

Edit Enabled field list instance property 2–15

Enable instance property 2–15, 2–17, 2–22, 2–45

Encapsulation 1–8

END trigger 2–18

Error handling 7–10
Error messages
   New ADM Class tool 8–9

Event on change instance property 2–45, 2–48

Events
   Commit SmartLink 3–10
   Container SmartLink 3–6
   Data SmartLink 3–9
   GroupAssign SmartLink 3–9
   Navigation SmartLink 3–7
   Page SmartLink 3–7
   procedures and functions 7–7
   TableIO SmartLink 3–8
   writing local functions 6–5
   writing local procedures 6–4

Exclude definition files
   custom 1–7, 8–13

Exit Browser on Default Action instance property 2–45

Explicit instance property 2–35, 2–36

Extensions, ADM
   adding properties 8–14
   design rules 8–13
   developing 8–1
   writing super procedures 8–15

External Field instance property 2–44

External procedures 1–14

F

Fetch Data to Fill Browse on Reposition to End of Batch instance property 2–16, 2–17

Field lists 4–4

Field-level validation 7–13

Fields
   displaying 2–23, 2–32

Fields instance property 2–34

Files
   class 1–4
   container class A–1
   custom class 1–6, 8–12
   get include 5–17
   primary include 5–2
   property 5–4
   prototype 5–4
   set include 5–17
   SmartDataObject deployment 7–23
   standard class 1–5
   super procedure 5–3
   templates 1–6

Filter instance property 2–41, 2–44

Filter SmartLinks 2–38, 3–3, 3–11

Filter Target instance property 2–34

Filtering
   data 2–32
   records 4–8

'First' on Left instance property 2–28

'First' on Right instance property 2–28

Font instance property 2–9

Force to Stateful Operating Mode instance property 2–9

Foreign key fields
   initializing 4–3

Format instance property 2–47

Frame SmartObjects 2–6

Functions 1–14
   event 6–5, 7–7
   running 5–7

G

Get functions 5–16

Get include files
   logic 5–17
   using in Progress code 5–18

{get} pseudo-syntax 5–17

Getting column properties 7–11

GroupAssign SmartLinks 3–3, 3–9
   events 3–9
   using 6–11

H

Height instance property 2–47

Help ID instance property 2–34, 2–47

HOME trigger 2–18

I

Implicit instance property 2–35

Include files 1–6
   See also Files

initializeObject procedure 5–7
**Index**

<table>
<thead>
<tr>
<th>Page</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4–3</td>
<td>Initializing foreign-key fields</td>
</tr>
<tr>
<td>7–22</td>
<td>SmartDataObjects</td>
</tr>
<tr>
<td>7–10</td>
<td>Initiating update operations</td>
</tr>
<tr>
<td>2–36</td>
<td>Inline instance property</td>
</tr>
<tr>
<td>2–46</td>
<td>Inner Lines instance property</td>
</tr>
<tr>
<td>2–31</td>
<td>Insert instance property</td>
</tr>
<tr>
<td>1–7, 8–13</td>
<td>Instance definition files</td>
</tr>
<tr>
<td>2–4, 5–19</td>
<td>Instance properties</td>
</tr>
<tr>
<td>2–28</td>
<td>Add Multiple Records</td>
</tr>
<tr>
<td>2–28</td>
<td>Add One Record</td>
</tr>
<tr>
<td>2–37</td>
<td>BEGINS</td>
</tr>
<tr>
<td>2–34</td>
<td>Browse</td>
</tr>
<tr>
<td>2–47</td>
<td>Browse Keys</td>
</tr>
<tr>
<td>2–46</td>
<td>Browse Label</td>
</tr>
<tr>
<td>2–16</td>
<td>Calculate Width</td>
</tr>
<tr>
<td>2–45</td>
<td>Cancel on Exit of Browse</td>
</tr>
<tr>
<td>2–9</td>
<td>Check Current Changed</td>
</tr>
<tr>
<td>2–37, 2–47</td>
<td>Column</td>
</tr>
<tr>
<td>2–37</td>
<td>Combo-box</td>
</tr>
<tr>
<td>2–41</td>
<td>Commit</td>
</tr>
<tr>
<td>2–37</td>
<td>CONTAINS</td>
</tr>
<tr>
<td>2–46, 2–47</td>
<td>Data Source</td>
</tr>
<tr>
<td>2–46</td>
<td>Data Type</td>
</tr>
<tr>
<td>2–34</td>
<td>Default</td>
</tr>
<tr>
<td>2–47</td>
<td>Define Any Key Trigger on Field</td>
</tr>
<tr>
<td>2–9</td>
<td>Destroy on each stateless Web Request</td>
</tr>
<tr>
<td>2–9</td>
<td>Disconnect AppServer on each Web Request</td>
</tr>
<tr>
<td>2–45</td>
<td>Display</td>
</tr>
<tr>
<td>2–44</td>
<td>Display Field</td>
</tr>
<tr>
<td>2–16</td>
<td>Down</td>
</tr>
<tr>
<td>2–27, 2–28</td>
<td>Edge Pixels</td>
</tr>
<tr>
<td>2–45</td>
<td>Edit Browse Columns</td>
</tr>
<tr>
<td>2–15</td>
<td>Edit displayed field list</td>
</tr>
<tr>
<td>2–15</td>
<td>Edit enabled field list</td>
</tr>
<tr>
<td>2–34</td>
<td>Edit Field List</td>
</tr>
<tr>
<td>2–45, 2–48</td>
<td>Enable</td>
</tr>
<tr>
<td>2–45</td>
<td>Event on change</td>
</tr>
<tr>
<td>2–45</td>
<td>Exit Browser on Default Action</td>
</tr>
<tr>
<td>2–35, 2–36</td>
<td>Explicit</td>
</tr>
<tr>
<td>2–44</td>
<td>External Field</td>
</tr>
<tr>
<td>2–34</td>
<td>Fields</td>
</tr>
<tr>
<td>2–44</td>
<td>Filter</td>
</tr>
<tr>
<td>2–34</td>
<td>Filter Target</td>
</tr>
<tr>
<td>2–28</td>
<td>'First' on Left</td>
</tr>
<tr>
<td>2–28</td>
<td>'First' on Right</td>
</tr>
<tr>
<td>2–31</td>
<td>Font</td>
</tr>
<tr>
<td>2–9</td>
<td>Force to Stateful Operating Mode</td>
</tr>
<tr>
<td>2–47</td>
<td>Format</td>
</tr>
<tr>
<td>2–47</td>
<td>Height</td>
</tr>
<tr>
<td>2–47</td>
<td>Help ID</td>
</tr>
<tr>
<td>2–35</td>
<td>Implicit</td>
</tr>
<tr>
<td>2–36</td>
<td>Inline</td>
</tr>
<tr>
<td>2–46</td>
<td>Inner Lines</td>
</tr>
<tr>
<td>2–31</td>
<td>Instance Properties</td>
</tr>
<tr>
<td>2–48</td>
<td>Key</td>
</tr>
<tr>
<td>2–44</td>
<td>Key Field</td>
</tr>
<tr>
<td>2–46</td>
<td>Label</td>
</tr>
<tr>
<td>2–15, 2–17, 2–22, 2–22, 2–22</td>
<td>Layout</td>
</tr>
<tr>
<td>2–16</td>
<td>Max Width</td>
</tr>
<tr>
<td>2–40</td>
<td>Menu</td>
</tr>
<tr>
<td>2–41</td>
<td>Navigation</td>
</tr>
<tr>
<td>2–46</td>
<td>No Value</td>
</tr>
<tr>
<td>2–38</td>
<td>Number of Lines in Editors</td>
</tr>
<tr>
<td>2–46</td>
<td>Optional</td>
</tr>
<tr>
<td>2–9</td>
<td>Partition</td>
</tr>
<tr>
<td>2–37</td>
<td>Radio-set</td>
</tr>
<tr>
<td>2–36</td>
<td>Range</td>
</tr>
<tr>
<td>2–9</td>
<td>Rebuild Dataset on Reposition</td>
</tr>
<tr>
<td>2–31</td>
<td>Remove</td>
</tr>
<tr>
<td>2–45</td>
<td>Reposition Data Source</td>
</tr>
<tr>
<td>2–47</td>
<td>Row</td>
</tr>
<tr>
<td>2–9</td>
<td>Rows To Batch</td>
</tr>
<tr>
<td>2–28</td>
<td>Save</td>
</tr>
<tr>
<td>2–15</td>
<td>Search Field</td>
</tr>
<tr>
<td>2–44</td>
<td>Show Border</td>
</tr>
<tr>
<td>2–21</td>
<td>SmartDataObject</td>
</tr>
<tr>
<td>2–45</td>
<td>Sort</td>
</tr>
<tr>
<td>2–31</td>
<td>Tab Labels</td>
</tr>
<tr>
<td>2–41</td>
<td>Tableio</td>
</tr>
<tr>
<td>2–34</td>
<td>Target</td>
</tr>
<tr>
<td>2–41</td>
<td>Toolbar</td>
</tr>
<tr>
<td>2–47</td>
<td>Tooltip</td>
</tr>
<tr>
<td>2–28</td>
<td>Update</td>
</tr>
<tr>
<td>2–31</td>
<td>Use Fixed Tabs</td>
</tr>
<tr>
<td>2–15, 2–17, 2–22, 2–38</td>
<td>View</td>
</tr>
<tr>
<td>2–35</td>
<td>View as range fields</td>
</tr>
<tr>
<td>2–46</td>
<td>View-as</td>
</tr>
<tr>
<td>2–47</td>
<td>Width</td>
</tr>
<tr>
<td>2–37</td>
<td>Width of Character Fields</td>
</tr>
<tr>
<td>2–37</td>
<td>Width of Other Fields</td>
</tr>
</tbody>
</table>

**Instance Properties instance property** 2–44

**Instance property dialog boxes**
- dynamic SmartDataBrowser 2–14
- Navigation SmartPanel 2–27
- simple SmartObject 2–51
- SmartDataField 2–24
- SmartDataObject 2–8
- SmartDataViewer 2–21
- SmartFilter 2–32
- SmartFolder 2–30
- SmartSelect 2–43
- SmartToolbar 2–40
- Update SmartPanel 2–28

**Instantiating SmartObjects** 1–11

**Interactions among SmartObjects** 5–1

**Internal procedures** 1–14
- running 5–7

**Introduction** 1–2
Index

J
Java AppObjects 7–29

K
Key Field instance property 2–44
Key instance property 2–48
Keys
  foreign 4–3

L
Label instance property 2–34, 2–46
Layout instance property 2–15, 2–17, 2–22
Links. See Pass-through links, SmartLinks
Localizing SmartObject behavior 5–8
Locking records 4–15

M
Managing data 4–1
Manipulating database queries dynamically 7–7
Masters 1–8, 1–10
  interactions with super procedures 5–3
SmartDataObject
    AppBuilder modifications 7–20
    compiling conditionally 7–20
Max Width instance property 2–16
Menu instance property 2–40
Methods 1–14
  See also Functions, Internal procedures
Modifying
class files 8–10
  records 2–26
Modifying SmartToolbars 2–42
Monitoring SmartObject applications 6–6

N
Navigating database records 2–26
Navigation instance property 2–41

Navigation SmartLinks 3–2, 3–6
  events 3–7
Navigation SmartPanels 2–26, 4–10
  instance property dialog box 2–27
New ADM Class tool 8–5
  creating class files 8–8
    dialog box
      Basic tab 8–6
      Custom Files tab 8–8
    warnings and error messages 8–9
No Value instance property 2–46
NO–ASSIGN property 2–18
Number of Lines in Editors instance property 2–38

O
Object classes
  See Classes
OFF–END
  event, scrolling a query’s result set 2–19
  trigger 2–18
OFF–HOME
  event, scrolling a query’s result set 2–19
  trigger 2–18
Opening database queries 7–7
Operating modes for SmartDataObjects 7–26
Optional instance property 2–46

P
Page 0 in SmartFolders 2–31
Page SmartLinks 3–2, 3–7
  events 3–7
PageN SmartLinks 3–2, 3–8
Pages
  selecting in a SmartContainer 2–30
Paging 2–6
Parameterizing SmartObject instances 6–2
Partition instance properties 2–9
Pass-through links 2–6, 2–38, 3–13, 3–15
  diagram 3–13
  inner and outer connections 3–15
  setting up 3–14
Primary include files 5–2
  container class A–2
  custom 1–6, 8–12
  standard 1–5

Pro*Spy Plus 6–9

PRO*Tools
  Pro*Spy Plus 6–9
  Procedure Object Viewer 6–6

Procedure Object Viewer 6–6
  starting 6–6
  viewing objects 6–7

Procedures
  See also Super procedures
  event 7–7
  event, writing local 6–4
  external 1–14
  internal 1–14
  maintained by AppBuilder 2–4
  monitoring 6–6
  top-level 1–11
  tracking 6–6
  viewing procedure objects 6–7

Properties 1–2, 1–13, 5–11
  See Instance properties
  accessing 5–20
  adding to ADM extensions 8–14
  ADMProps temp-table 5–11
  ASDivision 7–27
  ASHandle 7–27
  ASInfo 7–27
  ASUsePrompt 7–27
  columns, getting and setting 7–11
  direct access 8–14
  dynamic 5–21
  field, setting for dynamic
    SmartDataBrowsers 2–20
  get and set functions 5–16
  {get} and {set} pseudo-syntax 5–17
  instance 5–19
  NO—ASSIGN 2–18
  SearchField 2–20
  special 5–19
  translatable 5–19
  USE-DICT-EXPS 2–22

Property files 5–4
  container class A–4
  custom 1–6, 8–12
  standard 1–5

Prototype files 5–4
  container class A–12
  custom 1–7, 8–13
  standard 1–5

PUBLISH statement 3–5, 3–12

Q

Queries 7–3
  manipulating dynamically 7–7
  opening 7–7
  signaling data availability 7–11
  transferring data 7–9

R

Radio–set instance property 2–37

Range instance property control 2–36

Rebuild Dataset on Reposition instance property 2–9

Record-level validation 7–15

Records
  browsing 2–14
  committing multiple updates 2–26
  displaying 2–14, 2–21
  filtering 4–8
  locking 4–15
  modifying 2–26
  navigating 2–26
  retrieving 2–8, 3–15
  updating 2–14, 2–21, 2–26, 7–3

Remove instance property 2–31

removeQuerySelection() function 7–8

Reposition Data Source instance property 2–45

Resizing SmartPanels 2–29

Retrieving records
  from databases 2–8
  from multiple data sources 3–15

Row instance property 2–47

ROW–ENTRY trigger 2–18

ROW–ENTRY trigger 2–18

RowObject temp-table 4–2

Rows To Batch instance properties 2–9

RUN statement 5–7

Running
  internal procedures and functions 5–7
  SmartDataObjects 7–28
    batch-mode 7–28
    single-row 7–28
Save instance property 2–28

Scroll Remote Results List instance property 2–16

Scrolling a query’s result set 2–19

SCROLL–NOTIFY trigger 2–18

Search Field instance property 2–15

SearchField property 2–20

Selecting pages in a SmartContainer 2–30

Set functions 5–16

Set include files
  logic 5–17
  using in Progress code 5–18

{set} pseudo-syntax 5–17

setQuerySort( ) function 7–8

setQueryWhere( ) function 7–8

Setting properties
  column 7–11
  field, dynamic SmartDataBrowsers 2–20

Setting up pass-through links 3–13

Show Border instance property 2–27, 2–28, 2–41

Signaling data availability in queries 7–11

Signatures 4–4

Simple SmartObjects 1–9, 2–51
  instance property dialog box 2–51

Single-row operations 7–28

SmartContainers 1–8, 2–5
  See also SmartDataViewers, SmartDialogs, SmartFrames, SmartWindows
  contained objects 2–5
  paging 2–6
  pass-through links 2–6
  selecting pages 2–30

SmartDataBrowsers 1–8, 2–14
  browsing data 4–3
  compatibility checking 4–5
  dynamic 2–17
  dynamic and static compared 2–17
  files 2–14
  instance property dialog box 2–14
  multiple selection 2–19
  static 2–17

triggers 2–18
  usage notes 2–18

SmartDataFields 1–8, 2–23
  adding to SmartDataViewers 2–22, 2–25
  creating 2–24
  files 2–23
  instance property dialog box 2–24
  usage notes 2–24

SmartDataObject instance property 2–15, 2–44

SmartDataObjects 1–9, 2–8, 2–22
  accessing data 4–2
  accessing from Java AppObjects 7–29
  AppBuilder modifications to masters 7–20
  AppServer-related properties 7–27
  conditional compilation 7–20
  deployment files 7–23
  features 7–2
  files 2–8
  in distributed environments 7–19
  initializing 7–22
  instance properties 2–9
  instance property dialog box 2–8
  managing in distributed mode 7–23
  operating modes 7–26
  running from a non-Progress client 7–28
  service choices 7–25

SmartDataViewers 1–9, 2–21
  adding SmartDataFields 2–22, 2–25
  adding SmartObjects 2–22
  compatibility checking 4–5
  displaying data 4–3
  files 2–21
  instance properties 2–22
  instance property dialog box 2–21

SmartDialogs 1–8, 2–7
  files 2–7

SmartFilters 1–9, 2–32, 4–8
  connecting to in other SmartWindows 4–9
  customizing 4–8
  files 2–32
  instance properties 2–33
  instance property dialog box 2–32
  usage notes 2–38

SmartFolders 1–9, 2–30
  files 2–30
  instance properties 2–31
  instance property dialog box 2–30
  Page 0 2–31
  usage notes 2–31

SmartFrames 1–8, 2–6
  files 2–7
SmartLinks 1–2, 1–12, 3–1
See also Pass-through links
Commit 3–10
Container 3–6
Data 3–9
defining 3–18
dynamic, defining 3–19
Filter 2–38, 3–11
GroupAssign 3–9, 6–11
invoking behavior in linked 3–12
Navigation 3–6
overview 3–2
Page 3–7
PageN 3–8
source and target objects 3–4, 3–5
static, defining 3–18
TableIO 3–8
types 3–2
Update 3–10

SmartObjects
accessing 2–4
adding to SmartDataViewers 2–22
AppBuilder requirements 6–10
application benefits 1–2
class file relationships 5–5
contained 2–5
Data SmartLinks 7–12
design rules 8–13
include files 1–8
instance properties 2–4
instantiating 1–11
interactions 5–1
life cycle 1–10
localizing standard behavior 5–8
masters 1–8, 1–10
monitoring applications 6–6
parameterizing instances 6–2
properties 1–13
simple 2–51
structure 5–2
summary 2–3
super procedures 1–8
templates 1–8, 1–10
top-level procedure 1–11
types 1–8, 2–3
Update SmartLinks 7–12

SmartPanels 1–9, 2–26
as toolbars 2–29, 2–42
Commit 2–26
customizing 4–13
files 2–26
instance properties 2–27, 2–28
instance property dialog box 2–27, 2–28
Navigation 2–26
resizing 2–29
Update 2–26
usage notes 2–29
using 4–10

SmartSelects 1–9, 2–43
files 2–43
instance properties 2–44
instance property dialog box 2–43
usage notes 2–48

SmartToolbars 1–9, 2–39
customizing 4–14
files 2–39
instance property controls 2–40
instance property dialog box 2–40
modifying 2–42
usage notes 2–42
using 4–13

SmartWindows 1–8, 2–6
files 2–6
Sort instance property 2–45
Source objects 3–4, 3–5
SOURCE–PROCEDURE function 1–16
Standard class files 1–5
container class A–2
Starting super procedures 5–6
Starting the Procedure Object Viewer 6–6
State-aware operating mode 7–26
Stateless operating mode 7–26
State-reset operating mode 7–26
Static SmartDataBrowsers 2–17
compared to dynamic
SmartDataBrowsers 2–17
instance properties 2–17
SUBSCRIBE statement 3–5, 3–12
Super procedures 1–2, 1–16, 5–3
container class A–7
custom 1–7, 8–12
interactions with masters 5–3
localizing standard behavior 5–8
standard 1–5
starting 5–6
writing for ADM extensions 8–15
support directory 5–2

Support include files
See Primary include files

T

Tab Labels instance property 2–31
Tableio instance property 2–41
Index

TableIO SmartLinks 3–2, 3–8
   events 3–8
Target instance property 2–34
Target objects 3–4, 3–5
TARGET–PROCEDURE function 1–16
template directory 5–2
Templates 1–4, 1–6, 1–8, 1–10
   container class A–14
   frame A–14
Temp-tables
   ADMProps 5–11
   RowObject 4–2
Toolbar instance property 2–41
Toolbars 2–39
   SmartPanels 2–29, 2–42
Tooltip instance property 2–34, 2–47
Top-level procedures 1–11
Transaction-level validation 7–16
Transactions 4–15
Transferring data in database queries 7–9
Translatable properties 5–19
Triggers
   database 7–18
   SmartDataBrowser 2–18

U
Update instance property 2–28
Update SmartLinks 3–3, 3–10, 7–12
Update SmartPanels 2–26, 4–10
   instance property dialog box 2–28
Updating records 2–14, 2–21, 2–26, 7–3
   initiating 7–10
Use Fixed Tabs instance property 2–31
USE-DICT-EXPS property 2–22

Using
   Data SmartLinks 3–17, 4–7
   GroupAssign SmartLinks 6–11
   SmartPanels 4–10
   SmartToolbars 4–13

V
Validating data 7–13
   Data Dictionary 2–22, 7–13
   fields 7–13
   records 7–15
   transactions 7–16
VALUE–CHANGED trigger 2–18
View as range fields instance property 2–35
View instance property 2–15, 2–17, 2–22, 2–38
View–as instance property 2–46
Viewing procedure objects 6–7

W
Warnings
   New ADM Class tool 8–9
Width instance property 2–31, 2–34, 2–47
Width of Character Fields instance property 2–37
Width of Other Fields instance property 2–37
Window SmartObjects 2–6
Writing
   local event functions 6–5
   local event procedures 6–4
   super procedures for ADM extensions 8–15