

Part II: Using pattern libraries

This part contains information about using the COOL:Plex pattern libraries as you develop applications. It includes a list of pattern libraries, with descriptions of each library, followed by detailed examples of the objects in the FOUNDATION pattern library, and instructions for using wizards, property sheets, and some ActiveX controls in your application.

The major sections are:

What pattern libraries are available?	50
Pattern libraries structure	51
The FOUNDATION pattern library	51
Creating a wizard	74
Creating a property sheet	76
Creating an MDI parent	79
Customizing an ActiveX ToolBar	80
Common pattern library fields	82

What pattern libraries are available?

Pattern libraries are a set of COOL:Plex models that contain patterns. A pattern is a reusable design object that you can use to solve a recognizable business problem in your application. Once you inherit from a pattern, you can customize it to meet your specific needs. Every time you inherit from a COOL:Plex pattern, you create another pattern that you can reuse in future applications.

There are three groups of pattern libraries: Primitives, Technology patterns, and Business fundamentals.

Primitives – basic patterns for creating other patterns

- **OBJECTS** – The OBJECTS pattern library is the most fundamental of the Primitives. It contains basic objects that you can use to construct other patterns.
- **FIELDS** – The FIELDS pattern library contains fields common to all of the other pattern libraries.
- **VALIDATE** – The VALIDATE pattern library contains meta-functions that perform validation of entity and field relations.
- **STORAGE** – The STORAGE pattern library defines design objects for database access.
- **UIBASIC** – The UIBASIC pattern library contains functions with processing for simple panel elements that you can combine to define the user interface of your application.
- **UISTYLE** – The UISTYLE pattern library contains user interface functions and panel elements for listing, adding, updating, and deleting database records.
- **ACTIVE** – The ACTIVE pattern library contains wrapper functions for commonly available ActiveX controls and JavaBeans.

What pattern libraries are available?

- **DATE** – The DATE pattern library contains patterns for date and time calculations, conversions, and validation.

Technology patterns – API interfaces

- **WINAPI** – The WINAPI pattern library contains source code objects, written in C++, that provide functionality for Windows APIs such as messaging and registry support.
- **AS400** – The AS400 pattern library contains a set of objects that are used by AS/400 server functions.
- **ODBC3** – The ODBC3 pattern library provides APIs for those ODBC 3.0 features that are most likely to be used in business applications.
- **JAVAAPI** – The JAVAAPI pattern library contains source code objects that provide calls to Java API functions.

Business fundamentals – basic business objects

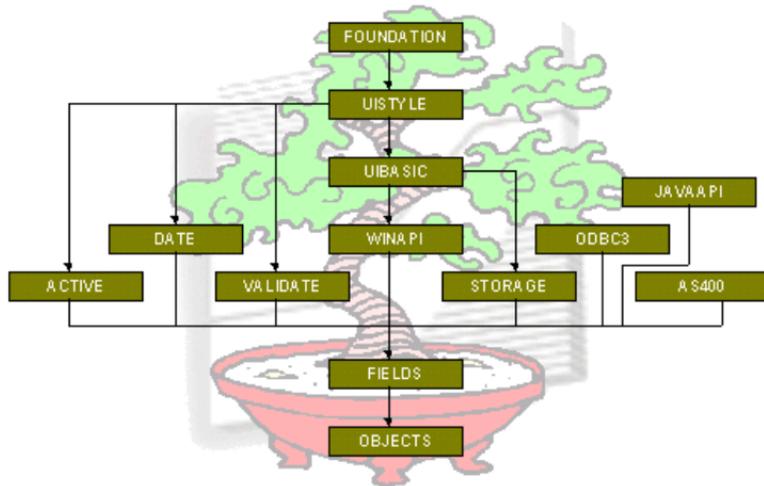
- **FOUNDATION** – The FOUNDATION pattern library contains a group of design objects that provide patterns for basic business objects. Most business applications will be based on patterns from this library.
- **BUSUPPORT** – The BUSUPPORT pattern library contains patterns that work in conjunction with other business support pattern libraries.
- In addition, the business fundamentals layer contains pattern libraries that provide solutions for a number of domains such as Name and Address (**BCONTACT**), Structure (**BSTRUCTURE**), Party-Role-Stage scenarios (**BPARTYROLESTAGE**), and Unit of Measure transformations (**BMEASURE**).

Pattern libraries structure

The COOL:Plex pattern libraries are organized in layers. At the root is the OBJECTS pattern library, which contains basic field, function, and variable definitions. All of the pattern libraries depend on this library.

Other layers contain: common fields, for use by all libraries; database access support for each server variant; technology libraries with platform-specific APIs; wrappers for ActiveX controls; and user-interface elements, in various combinations. Beginning with the FOUNDATION layer, you will find patterns for building business objects, customized to fit a variety of situations.

The pattern libraries are:



The FOUNDATION pattern library

The FOUNDATION pattern library

The FOUNDATION pattern library contains these basic entities:

- *EditDetail* – for basic maintenance of database records
- *EditDialog* – provides separate dialog boxes for adding, changing, and deleting database records
- *ReferredTo* – for entities that are the target of a **refers to** or **owned by** relation
- *Filter* – enables you to define criteria to use for filtering records to read
- *Owned* – the child part of a parent/child relationship
- *OwnedCascade* – adds cascade delete processing to Owned

This library contains support for entities with enumerated keys:

- *Surrogate* – assigns a numeric primary key to an entity
- *SurrogateAlternate* – provides a numeric primary key, but displays database records using an AlternateKey field
- *SurrogateOwned* – assigns a numeric primary key to records of a child entity, such as the detail lines of an order

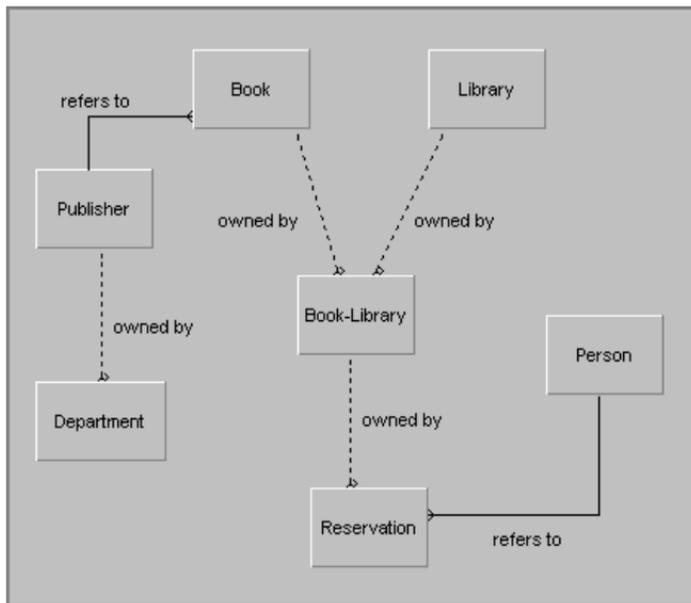
This library also contains patterns to support entities with two owners (**associated with** relationships):

- *AssociationEdit* – a two-parent child relationship that displays second owner records based on a selected first owner record
- *AssociationGrid* – a two-parent child relationship that displays records of the owning entities in parallel grids
- *AssociationDetail* – enables you to add non-key attributes to the association record itself

The Library Books example

The examples in this quick reference use a sample model based on books and libraries. Each section of the model showcases a variety of patterns and how you can use them. You can find this model in the Samples directory of your COOL:Plex installation.

The following entity relationship diagram shows the structure of the Library Books model:



Each entity in the Library Books model inherits from one or more patterns in the FOUNDATION pattern library. In addition, each entity inherits from STORAGE/RelationalTable.

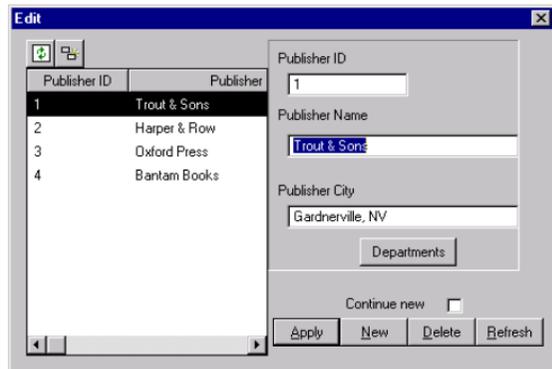
The following table shows the entities and their definitions:

Person	is a FOUNDATION/EditDetail is a FOUNDATION/ReferredTo
Book	is a FOUNDATION/EditDialog is a FOUNDATION/SurrogateAlternate is a FOUNDATION/ReferredTo
Publisher	is a FOUNDATION/EditDetail is a FOUNDATION/ReferredTo
Department	is a FOUNDATION/EditDetail is a FOUNDATION/Owned is a BSUPPORT/Overridden
Library	is a FOUNDATION/EditDetail is a FOUNDATION/Surrogate is a FOUNDATION/Filter is a FOUNDATION/ReferredTo is a BSUPPORT/ListExtension
Book-Library	is a FOUNDATION/AssociationDetail
Reservation	is a FOUNDATION/EditDialog is a FOUNDATION/SurrogateOwned
LibrarySystemSurrogates	is a FOUNDATION/SurrogateSystem

EditDetail – for basic maintenance of database records

OBASE analogue: This entity provides the same basic functionality as Grid Maintained Entity.

EditDetail is an entity that enables end-users to view database records, and to add, change, and delete records. The Edit function scoped to EditDetail lists database records in a grid on the left, and displays the selected row in an editing region on the right:



To add one record, click New. To add several records in succession, select Continue New, and then click New. When you add more than one record at a time, the grid is not refreshed after each addition.

To refresh the grid, click the Refresh button  above the grid.

Implementing EditDetail

To create an EditDetail entity:

1. Add the following inheritance triple to your model:
Publisher **is a** FOUNDATION/EditDetail
2. If your application uses a relational database, add the following inheritance triple:
Publisher **is a** STORAGE/RelationalTable
3. Specify the attributes of your entity, including inheritance triples for each field.
4. In the Object Browser, expand the Edit function you inherited from EditDetail.
5. Edit the large property of the scoped Caption object so that it contains the text you want to display on the title bar of the panel.
6. Open the panel design for the inherited Edit.Panel, and modify the panel so that the fields and controls are positioned where you want them.
You might need to move the grid region so that it does not overlap the Refresh button; or you might want to reposition the Apply, New, Refresh, and Delete buttons.
7. Close the panel design and save your changes.
8. Generate and build your EditDetail entity and all of its scoped objects.
9. To test the new functionality, run the Edit function.

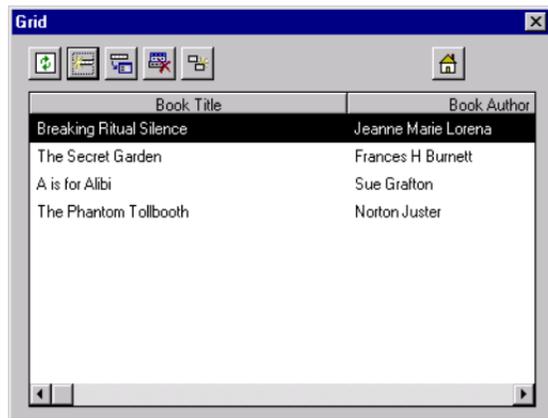
 **EditDialog** – provides separate dialog boxes for adding, changing, and deleting database records

OBASE analogue: This entity provides the same basic functionality as User Maintained Entity.

EditDialog is an entity that scopes a suite of functions that enable end-users to view database records, and to add, change, and delete records in separate dialog boxes.

Consider inheriting from EditDialog when your entity has more than five attributes or if you specifically want individual dialog boxes for adding, changing, and deleting records.

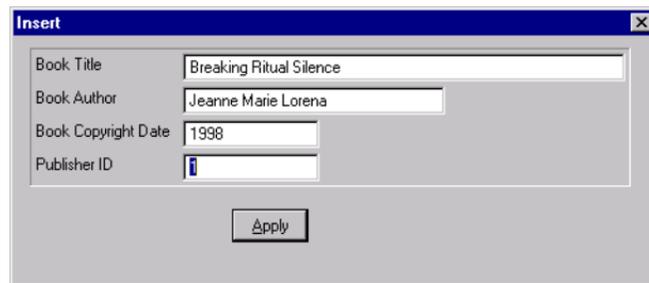
For a list of all records:



Book Title	Book Author
Breaking Ritual Silence	Jeanne Marie Lorena
The Secret Garden	Frances H Burnett
A is for Alibi	Sue Grafton
The Phantom Tollbooth	Norton Juster

The FOUNDATION pattern library

To add a record, click the Add button  on the Grid panel:



Insert

Book Title: Breaking Ritual Silence

Book Author: Jeanne Marie Lorena

Book Copyright Date: 1998

Publisher ID: 1

Apply

To change a record, click the Change button  on the Grid panel:



Update

Book Title: The Phantom Tollbooth

Book Author: Norton Juster

Book Copyright Date: 1961

Publisher ID: 2

Apply Refresh

To delete a record, click the Delete button  on the Grid panel:



Implementing EditDialog

To create an EditDialog entity:

1. Add the following inheritance triple to your model:
Book **is a** FOUNDATION/EditDialog
2. If your application uses a relational database, add the following inheritance triple:
Book **is a** STORAGE/RelationalTable
3. Specify the attributes of your entity, including inheritance triples for each field.
4. In the Object Browser, expand the EditSuite function you inherited from EditDialog.
Notice the four functions in the EditSuite: Delete, Grid, Insert, and Update.
5. In the Object Browser, expand each EditSuite function, and edit the large property of the scoped Caption object so that it contains the text you want to display on the title bar.
6. For each EditSuite function, open the panel design for the scoped Panel, and modify the panel so that the fields and controls are positioned where you want them.
7. Close the panel design and save your changes.
8. Generate and build your EditDialog entity and all of its scoped objects.
9. To test the new functionality, run the Grid function scoped to EditSuite.

ReferredTo – for the target of a *refers to* relation

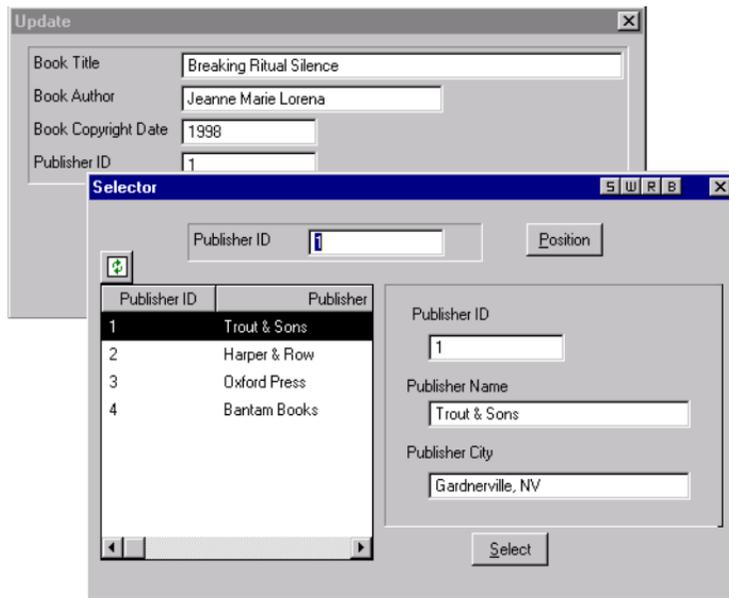
OBASE analogue: This entity provides the same basic functionality as Referenced Entity, but does not return virtual attributes.

ReferredTo is an entity that supports reference checking and prompt processing. If your entity is the target of an ENT **refers to** ENT or ENT **owned by** ENT triple, then you can use ReferredTo to verify the referential integrity of database records.

Inherit from ReferredTo in addition to another pattern such as FOUNDATION/EditDetail or FOUNDATION/EditDialog.

ReferredTo scopes a Selector function that allows end-users to select from a list of existing records to fill in a foreign key field on a panel.

When you double-click a foreign key field, the Selector Panel appears:



Implementing ReferredTo

Inherit from FOUNDATION/ReferredTo, in addition to one of the other patterns in the FOUNDATION pattern library, such as EditDialog or EditDetail.

To define a ReferredTo entity:

1. Assume that your model has the following triples:
 - Book **is a** FOUNDATION/EditDialog
 - Book **is a** STORAGE/RelationalTable
 - Publisher **is a** FOUNDATION/EditDetail
 - Publisher **is a** STORAGE/RelationalTable
 - Book **refers to** Publisher
2. Add the following triple:
 - Publisher **is a** FOUNDATION/ReferredTo
3. Open the panel scoped to the Selector function you inherited from ReferredTo, and make any changes you want to the panel layout.
4. Close the panel design and save your changes.
5. If you want, edit the message object Caption scoped to Selector to change the title bar text for the selector panel.

6. Generate and build the following functions:
 - Publisher.Selector
 - Publisher.Fetch.CheckRow
 - Book.EditSuite.Insert
 - Book.EditSuite.Update
7. Test the Insert and Update functions in Book's EditSuite to see that referential integrity checking and prompt processing are now working.

Filter – enables you to define criteria to use for filtering records to read

OBASE analogue: This entity combines the functionality of Entity With User Filter and Entity With SQL Data.

Filter is an entity that enables you to define filtering criteria to specify which records of a relational database to display on a grid.

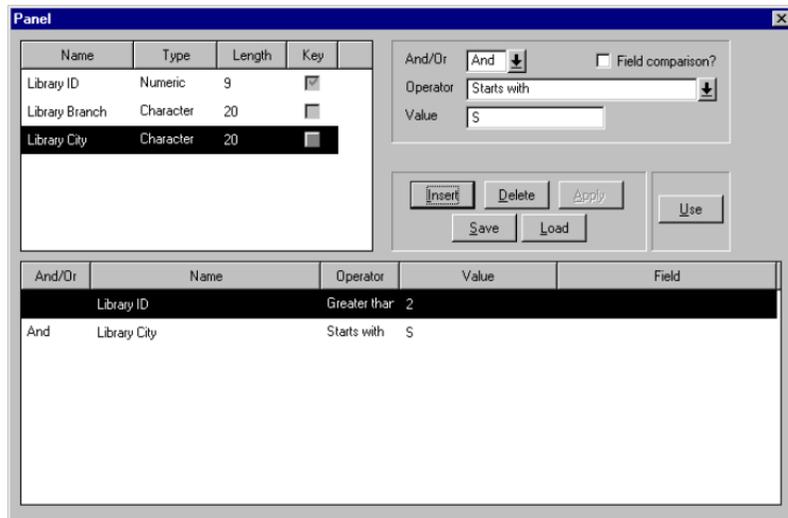
The FilteredGrid function scoped to this entity calls the FilterCriteria function to allow end-users to define filter conditions using a combination of operators and field values. The criteria you define are combined and passed to the server for filtering:

- For SQL databases, you can specify how to combine filtering criteria using the logical operators AND and OR.
- For DDS databases, the filter criteria are ANDed together.
- You can save the filter criteria you define to a file and load them for future use.

Inherit from Filter in addition to another pattern such as FOUNDATION/EditDetail or FOUNDATION/EditDialog.

Note: The FilterCriteria function scoped to this entity makes explicit references to SQL or DDS as a database language. Therefore, you can only inherit from this entity if you have also inherited from STORAGE/RelationalTable.

Using the FilterCriteria panel, you can specify how to filter records:



The screenshot shows a software interface titled "Panel" with a close button (X) in the top right corner. It is divided into two main sections. The top section contains a table with columns "Name", "Type", "Length", and "Key".

Name	Type	Length	Key
Library ID	Numeric	9	<input checked="" type="checkbox"/>
Library Branch	Character	20	<input type="checkbox"/>
Library City	Character	20	<input type="checkbox"/>

To the right of this table is a configuration area with the following elements:

- "And/Or" dropdown menu set to "And".
- "Field comparison?" checkbox, which is unchecked.
- "Operator" dropdown menu set to "Starts with".
- "Value" text input field containing the character "S".
- A group of buttons: "Insert", "Delete", "Apply", "Save", "Load", and "Use".

The bottom section of the panel is a table with columns "And/Or", "Name", "Operator", "Value", and "Field".

And/Or	Name	Operator	Value	Field
	Library ID	Greater than	2	
And	Library City	Starts with	S	

- Click Save to save a set of filter criteria for future use.
- Click Load to use a set of filter criteria that you have previously saved.

Implementing Filter

Remember that you can only use the functionality of this entity if you also inherit from STORAGE/RelationalTable.

To define a Filter entity:

1. Assume that your model has the following triples:

Library **is a** FOUNDATION/EditDetail

Library **is a** STORAGE/RelationalTable

2. Add the following triple:

Library **is a** FOUNDATION/Filter

Inheriting from Filter adds a function called FilteredGrid, and a view called Filter to Library. The FilteredGrid function lets you filter database records for display, using the FilterCriteria function scoped to the Filter view. The Filter view contains the fields you can use to define filter criteria.

3. Add the following triple to your Edit function to enable filter processing:

Library.Edit **replaces** UIBASIC/Grid
... **by** Library.FilteredGrid

Note: If your entity inherits from FOUNDATION/EditDialog instead of FOUNDATION/EditDetail, add this triple to your entity's EditSuite.Grid function instead.

4. Open the panel scoped to your Edit function. Notice that inheriting from FilteredGrid added a Filter button  to the grid region.

5. Select the Filter button and drag it so that it does not cover the Copy button, or any other button in the grid region.
6. Close the panel design and save your changes.
7. Generate and build the following objects:
 - Library.Edit
 - Library.Filter, and its scoped functions
8. To test your changes, run your Edit function and click the Filter button .

Defining filter criteria

Using the FilterCriteria panel, you can define criteria to send to the server to filter database records.

Defining filter criteria involves:

- Specifying filter conditions
 - Fields – by value or field-to-field comparison
 - Operators – numeric vs. character fields
- Combining conditions: SQL vs. DDS
- Loading, saving, or changing filter criteria

Notes:

- For key fields in a DDS database, only the Greater Than Or Equal To operator is available.
- The Starts With and Contains operators only apply to character fields.

Owned – the child part of a parent/child relationship

OBASE analogue: This entity is analogous to the Child entity.

Owned is an abstract entity, used for the child part of a parent/child relationship. An Owned entity has at least one ENT **owned by** ENT triple associated with it. Owned records depend on Owner records; in other words, you cannot work with a record of an owned entity without specifying which owner record it belongs to.

An entity can be both a child and a parent, for grandparent/parent/child relationships, such as Company-Department-WorkGroup. In this case, the keys of Company are automatically used to restrict lists of Department records, and the keys of both Company and Department to restrict records of WorkGroup.

To use an Owned entity, you need to specify the owning entity. When you inherit from Owned, you must replace the abstract entity FOUNDATION/Owner; however, do not inherit from Owner directly. Inheriting from Owner could cause unexpected results in an application.

Note: The Owned entity does not include functionality for cascade deletion (where deleting a parent record also deletes remaining child records for that parent). If you want cascade deletion, inherit from FOUNDATION/OwnedCascade instead. Using OwnedCascade is identical to using Owned in all other respects.

Implementing owner/owned relationships

In this example, a Publisher entity owns an entity called Department. Both Publisher and Department inherit from FOUNDATION/EditDetail for user interface processing.

Because EditDetail does not contain any processing for restricted keys, you need to modify the inherited Edit function to specify a parent record in order to see only corresponding records of the child entity.

Implementing an owner/owned relationship consists of:

- Defining the entities
- Adding restrictor processing to the user interface functions
- Calling the user interface functions with an owner key

To define the entities in the owner/owned relationship:

1. Assume that your model has an entity called Publisher:
Publisher **is a** FOUNDATION/EditDetail
2. Add the following inheritance triples:
Department **is a** FOUNDATION/EditDetail
Department **is a** FOUNDATION/Owned
3. If your application uses a relational database, add the following inheritance triples:
Publisher **is a** STORAGE/RelationalTable
Department **is a** STORAGE/RelationalTable

4. In the Model Editor, select the Department **is a** Owned triple, and click the Editor button .

This opens the Template Editor. You will use this editor to specify which entity in your model actually owns the Department entity.

5. Type `Publisher` in the Replaced By column to identify the owner of Department. Close the Template Editor and save your changes.

This adds the following triple to your model:

```
Department replaces FOUNDATION/Owner
... by Publisher
```

To add restrictor processing to user interface functions:

1. Add the following triples to enable restricted data fetches for EditDetail:

```
Department.Edit input view Department.SuperKeys
... for OBJECTS/Input
```

```
Department.Edit local view Department.SuperKeys
... for STORAGE/Restrict
```

The view `Department.SuperKeys` contains the key fields of the owning entity, in this case `Publisher`. These triples specify that you only want to edit Department records for a specific `Publisher`.

2. Add the following triple to replace `BlockFetch` with the specialized function `BlockFetchSet` that you inherited from `Owned`:

```
Department.Edit replaces Department.Fetch.BlockFetch
... by Department.Fetch.BlockFetchSet
```

This function uses the key values that you specified for the local variable `Restrict` in the preceding step.

To specify a parent key:

1. To edit Departments associated with a specific `Publisher`, open the panel scoped to `Publisher.Edit`, and add a push button labeled `Departments`.
2. Add an event, `Show Departments`, and map it to the `Pressed` physical event on your `Departments` button. Close the panel design and save your changes.
3. Open the action diagram for `Publisher.Edit` and go to the Events collection point.
4. Add an event construct for your `Show Departments` event, with a call to `Department.Edit`:



5. Map the Department ID from the `DetailP` variable. Close the action diagram and save your changes.

Putting it all together:

1. Generate and build:
 - the Department entity and its scoped objects
 - `Publisher.Edit`
2. To test the owner/owned relationship, run `Publisher.Edit`, and click the `Departments` button you created.

Surrogate – assigns a numeric primary key to an entity

OBASE analogue: This entity is analogous to Enumerated Entity.

Surrogate is an entity that assigns numeric primary keys to database records. Keys are assigned sequentially as you add each new record.

Inherit from this entity to supplement the functionality of an entity such as FOUNDATION/EditDetail. You do not need to specify a **known by** relation for your entity; inheriting from Surrogate provides a key field, FIELDS/Surrogate.

Note: Whenever you inherit from FOUNDATION/Surrogate or FOUNDATION/SurrogateAlternate, your model must have an entity that inherits from FOUNDATION/SurrogateSystem. SurrogateSystem maintains the surrogate values assigned to all of the entities in your application.

Implementing Surrogate

Implementing a Surrogate entity consists of:

- Defining the inheritance and replacement triples
- Modifying panel designs
- Specifying a SurrogateSystem entity

To define the inheritance and replacement triples:

1. Assume that your model includes the following triples:
Library **is a** FOUNDATION/EditDetail
Library **is a** STORAGE/RelationalTable
Library ID **is a** FIELDS/Surrogate
2. Add the following inheritance triple to your model:
Library **is a** FOUNDATION/Surrogate
3. Select the inheritance triple you just added and click the Editor button .
This opens the Template Editor. You will use this editor to specify which field in your model is actually the primary key of Library.
4. Type `Library ID` in the Replaced By column. Close the Template Editor and save your changes.
This adds the following triple to your model:
Library **replaces** FIELDS/Surrogate
... **by** Library ID

To modify the panel designs:

1. For any panels in your application where end-users can add or change Surrogate records (in this example, Library.Edit.Panel), open the panel design and either set the mode of your Surrogate field to Read Only, or set the Visible property of the field to No.
2. Close the Panel Designer and save your changes.

To specify the SurrogateSystem entity:

1. If you have not already defined a FOUNDATION/SurrogateSystem entity for your model, define one now.

This entity maintains information about numeric surrogate values assigned to all of the Surrogate and SurrogateAlternate entities in your application.

2. Add the following replacement triple to use your SurrogateSystem entity:

```
Library replaces FOUNDATION/SurrogateSystem  
    ...by LibrarySystemSurrogates
```

Putting it all together:

1. Generate and build your Surrogate entity and all of its scoped objects. If you have not already generated and built your application's SurrogateSystem entity, generate and build it as well.
2. Add some records to your Surrogate entity to test the automatic key assignments.

Implementing SurrogateSystem

To create a SurrogateSystem entity to maintain surrogate key values assigned to entities in your application:

1. Add the following inheritance triple to your model:
MySurrogateSystem **is a** FOUNDATION/SurrogateSystem
2. If your application uses a relational database, add the following inheritance triple:

```
MySurrogateSystem is a STORAGE/RelationalTable
```

3. For each entity in your model that inherits from either FOUNDATION/Surrogate or FOUNDATION/SurrogateAlternate, add the following replacement triple:

```
MySurrogateEntity replaces FOUNDATION/SurrogateSystem  
    ...by MySurrogateSystem
```

Note: You must add this triple in the Model Editor; you cannot use the Template Editor to add it.

4. Generate and build MySurrogateSystem and all of its scoped objects.

SurrogateAlternate – provides a numeric primary key, but displays records using a specified AlternateKey field

OBASE analogue: This entity provides an extended version of the functionality of Enumerated Entity.

SurrogateAlternate is an entity that assigns numeric primary keys to database records. Keys are assigned sequentially, as you add each new record, and stored in a special field, FOUNDATION/HiddenSurrogate. This field scopes another field, AlternateKey, whose values will be displayed instead of the primary key on panels in your application.

When you use SurrogateAlternate, records are listed in order by the AlternateKey field. The functions scoped to this entity prevent you from entering duplicate AlternateKey values.

As an example, a Book entity that inherits from SurrogateAlternate will have a HiddenSurrogate as its primary key, and the scoped AlternateKey field associated with each record could display the Book Title rather than the numeric key assigned to it.

Inherit from this entity to supplement the functionality of an entity such as FOUNDATION/EditDetail or FOUNDATION/EditDialog.

Note: Whenever you inherit from FOUNDATION/SurrogateAlternate or FOUNDATION/Surrogate, your model must have a FOUNDATION/SurrogateSystem entity. SurrogateSystem maintains the surrogate values assigned to all of the entities in your application.

Implementing SurrogateAlternate

Implementing a SurrogateAlternate entity is similar to implementing a Surrogate entity, with one significant addition. For a SurrogateAlternate entity, you define an AlternateKey field, which will be displayed in place of the numeric key assigned to database records.

Implementing a SurrogateAlternate entity consists of:

- Defining the inheritance and replacement triples
- Defining the AlternateKey field
- Modifying panel designs
- Specifying a SurrogateSystem entity

To define the inheritance and replacement triples:

1. Assume that your model includes the following triples:
Book **is a** FOUNDATION/EditDialog
Book **is a** STORAGE/RelationalTable
Book ID **is a** FOUNDATION/HiddenSurrogate
2. Add the following inheritance triple to your model:
Book **is a** FOUNDATION/SurrogateAlternate
3. Select the inheritance triple you just added and click the Editor button .
This opens the Template Editor. You will use this editor to specify which field in your model is actually the primary key of Book.
4. Type `Book ID` in the Replaced By column for HiddenSurrogate. Close the Template Editor and save your changes.

This adds the following triple to your model:

Book **replaces** FOUNDATION/HiddenSurrogate
... **by** Book ID

5. Add the following triple to use the specialized BlockFetch function that returns records in AlternateKey order:

Book.EditSuite.Grid **replaces** Book.Fetch.BlockFetch
...**by** Book.AlternateKey.BlockFetch

To define the AlternateKey field:

1. Add an inheritance triple to the AlternateKey field scoped by your HiddenSurrogate field to define properties for this field; for example:

Book ID.AlternateKey **is a** FIELDS/ShortDescription

2. Add label, left label, and top label triples to your AlternateKey field to specify how you want this field identified on panels; for example you could use a label of Book Title:

Book ID.AlternateKey **label** BookTitle

BookID.AlternateKey **left label** BookTitle

Book ID.AlternateKey **top label** BookTitle

3. Add the literal value `Book Title` to the large property of the BookTitle label you just created.
4. Add the following triples to replace the views used by the IntToExt and ExtToInt functions scoped to your HiddenSurrogate field:

Book ID.IntToExt **replaces** SurrogateAlternate.Fetch
... **by** Book.Fetch

Book ID.ExtToInt **replaces** SurrogateAlternate.AlternateKey
... **by** Book.AlternateKey

The FOUNDATION pattern library

To modify the panel designs:

1. For any panels in your application where end-users can add or change records of your SurrogateAlternate entity (in this example, the functions scoped to Book.EditSuite), open the panel design and either set the mode of your HiddenSurrogate field to Read Only, or set the Visible property of the field to No.
2. Close the Panel Designer and save your changes.

To specify the SurrogateSystem entity:

1. If you have not already defined a FOUNDATION/SurrogateSystem entity for your model, define one now (for more information, see page 63).

This entity maintains information about numeric surrogate values assigned to all of the FOUNDATION/Surrogate and FOUNDATION/SurrogateAlternate entities in your application.

2. Add the following replacement triple:

Book **replaces** FOUNDATION/SurrogateSystem
...**by** LibrarySystemSurrogates

Putting it all together:

1. Generate and build your SurrogateAlternate entity, your HiddenSurrogate field, and the objects scoped to each. If you have not already generated and built your application's SurrogateSystem entity, generate and build it as well.
2. Add some records to your SurrogateAlternate entity to test the automatic key assignments and the display of the alternate field you specified.

 **SurrogateOwned** – assigns a numeric primary key to records of a child entity, such as the detail lines of an order

OBASE analogue: This entity provides a modified version of the functionality of Enumerated Entity.

SurrogateOwned is an entity that assigns a numeric primary key value to each record of an entity that is owned by another entity.

Because SurrogateOwned contains no user interface elements, you must either inherit from FOUNDATION/EditDetail or FOUNDATION/EditDialog, or define your own user interface processing using patterns from the UIBASIC or UISTYLE pattern libraries.

The BySurrogate view scoped to this entity determines the next available surrogate value to assign to records associated with a specific record of the owning entity, FOUNDATION/Owner.

Implementing SurrogateOwned

In this example, the Book-Library entity owns an entity called Reservation. Reservation is a SurrogateOwned entity and inherits from FOUNDATION/EditDialog for user interface processing.

Because EditDialog does not provide any processing for restricted keys, you need to modify the inherited Insert and Grid functions to specify a parent record to see only corresponding records of the child entity.

Implementing a SurrogateOwned entity consists of:

- Defining the owner and owned entities
- Adding restrictor processing to the user interface functions
- Calling the user interface functions with an owner key

To define the entities in the owner/owned relationship:

1. Assume that your model has an association entity called Book-Library:
Book-Library **is a** FOUNDATION/AssociationDetail
2. Add the following inheritance triples:
Reservation **is a** FOUNDATION/EditDialog
Reservation **is a** FOUNDATION/SurrogateOwned
3. If your application uses a relational database, add the following inheritance triples:
Book-Library **is a** STORAGE/RelationalTable
Reservation **is a** STORAGE/RelationalTable
4. In the Model Editor, select the Reservation **is a** SurrogateOwned triple, and click the Editor button .
This opens the Template Editor. You will use this editor to specify which entity actually owns the Reservation entity, and which field is actually the primary key of Reservation (in addition to the keys of the owning entity).
5. Type `Book-Library` in the Replaced By column to identify the owner of Reservation, and type `Reservation Number` in the Replaced By column to identify the enumerated key field.
6. Close the Template Editor and save your changes.

This adds the following triples to your model:

Reservation **replaces** FOUNDATION/Owner
...by Book-Library

Reservation **replaces** FIELDS/Surrogate
...by Reservation Number

To add restrictor processing to user interface functions:

1. Add the following triples to enable restricted data fetches for the Insert and Grid functions of EditDialog's EditSuite:

Reservation.EditSuite.Insert **input view** Reservation.SuperKeys
...for OBJECTS/Input

Reservation.EditSuite.Insert **local view** Reservation.SuperKeys
...for STORAGE/Restrict

Reservation.EditSuite.Grid **input view** Reservation.SuperKeys
...for OBJECTS/Input

Reservation.EditSuite.Grid **local view** Reservation.SuperKeys
...for STORAGE/Restrict

The view Reservation.SuperKeys contains the key fields of the owning entity, in this case Book-Library. These triples specify that you only want to edit Reservation records for a specific Book-Library record.

2. Add the following triple to replace BlockFetch with the specialized function BlockFetchSet that you inherited from SurrogateOwned:

Reservation.EditSuite.Grid **replaces** Reservation.Fetch.BlockFetch
...by Reservation.Fetch.BlockFetchSet

This function uses the key values that you specified for the local variable Restrict in the preceding step.

To specify an owner key:

1. To edit Reservations for a specific Book and Library combination, open the panel scoped to Book-Library.Edit, and add a push button labeled Reservations in the detail region at the bottom of the panel.
2. Add an event, Reservations, and map it to the Pressed physical event on your Reservations button. Close the panel design and save your changes.
3. Open the action diagram for Book-Library.Edit and go to the Events collection point.
4. Add an event construct for your Reservations event, with a call to Reservation.EditSuite.Grid:



5. Map the Book ID and the Library ID owner key fields from the DetailP variable. Close the action diagram and save your changes.

Putting it all together:

1. Generate and build:
 - the Reservation entity and all of its scoped objects
 - Book-Library.Edit
2. To test the new functionality, run Book-Library.Edit and click the Reservations button you created.

AssociationEdit – a two-parent child relationship that displays second owner records based on a selected record of the first owner

OBASE analogue: This entity is analogous to Two Parent Child, with the **edit dialog** function option set to No.

AssociationEdit is an entity that contains functionality to work with an entity that is **owned by** two parent entities. You can use this entity to model an **associated with** relationship. Because this “many-to-many” relationship cannot be resolved directly in a relational database, you can use a new entity to store the cross references.

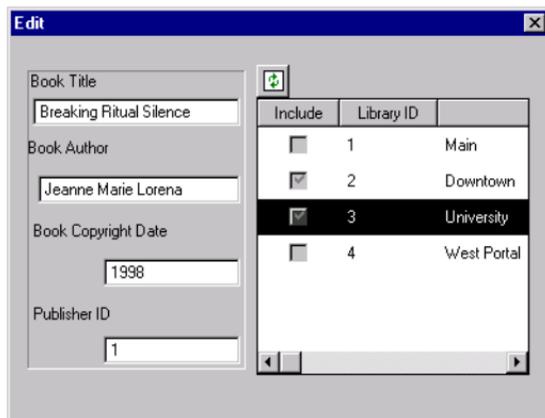
AssociationEdit is **owned by** two abstract entities, FOUNDATION/Owner and FOUNDATION/Owner2. When you inherit from AssociationEdit, you replace these abstract entities with the actual parent entities in your model.

This pattern displays a first owner record in a detail region, and enables end-users to associate it with second owner records displayed in a grid. To use this functionality, call AssociationEdit.Edit with a first owner record selected.

Note: If you use this pattern, do not define any attributes on the entity that inherits from it. If you want to specify non-key attributes for an entity of this sort, inherit from FOUNDATION/AssociationDetail instead.

End-users specify associations by selecting a first owner record, and clicking the Include column of the grid twice to select or deselect second owner records.

- When an end-user selects Include, this function calls Update.InsertRow to add an AssociationEdit record.
- When an end-user deselects Include, this function calls Update.DeleteRow to remove the AssociationEdit record.



Implementing AssociationEdit

To create an AssociationEdit entity; for example, one that associates books with libraries:

1. Assume that your model has the following triples:
Book **is a** FOUNDATION/EditDialog
Book **is a** STORAGE/RelationalTable
Library **is a** FOUNDATION/EditDetail
Library **is a** STORAGE/RelationalTable
2. Add the following inheritance triple to your model:
Book-Library **is a** FOUNDATION/AssociationEdit
3. If your application uses a relational database, add the following inheritance triple:
Book-Library **is a** STORAGE/RelationalTable
4. In the Model Editor, select the triple where you inherit from AssociationEdit and click the Editor button .
This opens the Template Editor. You will use this editor to specify which entities in your model actually own the Book-Library entity.
5. In the Replaced By column for Owner, type `Book`, and in the Replaced By column for Owner2, type `Library`. Close the Template Editor and save your changes.
This adds the following triples to your model:
Book-Library **replaces** FOUNDATION/Owner
...by Book
Book-Library **replaces** FOUNDATION/Owner2
...by Library

The FOUNDATION pattern library

6. To use the functionality in AssociationEdit, you need to call AssociationEdit.Edit with a first owner record; in this case, a Book record. Open the Panel scoped to Book.EditSuite.Grid and add an event called AssignBookToLibrary.
7. Add a push button to the panel and map the Clicked physical event to the event you just created. Close the panel design and save your changes.
8. Open the action diagram for Book.EditSuite.Grid and go to the Events collection point.
9. Add the following event construct:



10. Map the input parameter (in this case, Book ID) to the corresponding field in the GridP region. Close the action diagram and save your changes.
11. Generate and build your AssociationEdit entity and all of its scoped objects, along with the function that calls AssociationEdit.Edit.
12. Ensure that you have defined records for both of the owning entities in your model (in this case, Book and Library).
13. To test the new functionality, run the function that calls your Edit function (in this example, Book.EditSuite.Grid).

AssociationGrid – a two-parent child relationship that displays records of the owning entities in parallel grids

OBASE analogue: This entity is analogous to Two Parent Child, with the **edit dialog** function option set to No.

AssociationGrid is an entity that contains functionality to work with an entity that is **owned by** two parent entities. You can use this entity to model an **associated with** relationship. Because this “many-to-many” relationship cannot be resolved directly in a relational database, you can use a new entity to store the cross references.

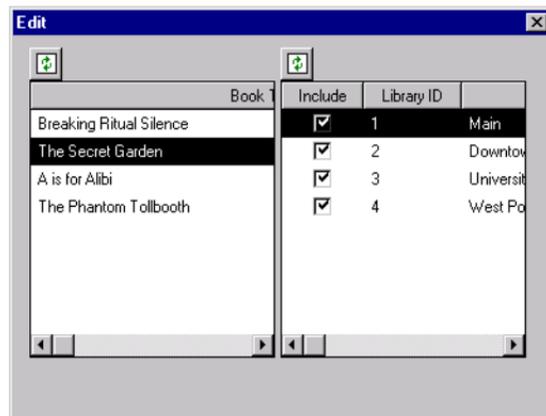
AssociationGrid is **owned by** two abstract entities, FOUNDATION/Owner and FOUNDATION/Owner2. When you inherit from AssociationGrid, you replace these abstract entities with the actual parent entities in your model.

This pattern lets you select a first owner record from a grid, and associate it with second owner records in an adjacent grid.

Note: If you use this pattern, do not define any attributes on the entity that inherits from it. If you want to specify non-key attributes for an entity of this sort, inherit from FOUNDATION/AssociationDetail instead.

End-users specify associations by selecting a first owner record in the grid on the left, and clicking the Include column of the grid on the right twice to select or deselect records of the second owning entity.

- When an end-user selects Include, this function calls Update.InsertRow to add an AssociationGrid record.
- When an end-user deselects Include, this function calls Update.DeleteRow to remove the AssociationGrid record.



Implementing AssociationGrid

To create an AssociationGrid entity; for example, one that associates books with libraries:

1. Assume that your model has the following triples:
Book **is a** FOUNDATION/EditDialog
Book **is a** STORAGE/RelationalTable
Library **is a** FOUNDATION/EditDetail
Library **is a** STORAGE/RelationalTable
2. Add the following inheritance triple to your model:
Book-Library **is a** FOUNDATION/AssociationGrid
3. If your application uses a relational database, add the following inheritance triple:
Book-Library **is a** STORAGE/RelationalTable
4. In the Model Editor, select the triple Book-Library **is a** AssociationGrid, and click the Editor button .
This opens the Template Editor. You will use this editor to specify which entities in your model actually own the Book-Library entity.
5. In the Replaced By column for Owner, type `Book`, and in the Replaced By column for Owner2, type `Library`. Close the Template Editor and save your changes.
This adds the following triples to your model:
Book-Library **replaces** FOUNDATION/Owner
 ...by Book
Book-Library **replaces** FOUNDATION/Owner2
 ...by Library

The FOUNDATION pattern library

6. Generate and build your AssociationGrid entity and all of its scoped objects.
7. Ensure that you have defined records for both of the owning entities in your model (in this case, Book and Library).
8. To test the new functionality, run the Edit function.

AssociationDetail – a two-parent child relationship that enables you to add non-key attributes to the association record itself

OBASE analogue: This entity is analogous to Two Parent Child, with the **edit dialog** function option set to Yes.

AssociationDetail is an entity that contains functionality to work with an entity that is **owned by** two parent entities. You can use this entity to model an **associated with** relationship. Because this “many-to-many” relationship cannot be resolved directly in a relational database, you can use a new entity to store the cross references.

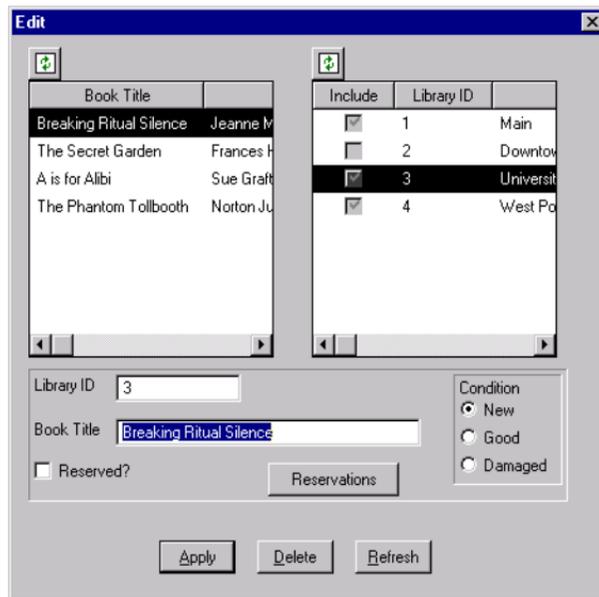
AssociationDetail is **owned by** two abstract entities, FOUNDATION/Owner and FOUNDATION/Owner2. When you inherit from AssociationDetail, you replace these abstract entities with the actual parent entities in your model.

This pattern lets you select a first owner record from a grid, and associate it with second owner records in an adjacent grid. Using this pattern, you can also specify attributes for your AssociationDetail entity, in addition to the two parent keys.

Note: This is the only one of the Association patterns where you can define non-key attributes.

End-users specify associations by selecting a first owner record, selecting a second owner record, and then modifying the values in the detail region:

- When an end-user clicks Apply, this function calls Update.InsertRow to add an AssociationDetail record.
- When an end-user clicks Delete, this function calls Update.DeleteRow to remove the AssociationDetail record.



Implementing AssociationDetail

To create an AssociationDetail entity; for example, one that associates books with libraries:

1. Assume that your model has the following triples:
Book **is a** FOUNDATION/EditDialog
Book **is a** STORAGE/RelationalTable
Library **is a** FOUNDATION/EditDetail
Library **is a** STORAGE/RelationalTable
2. Add the following inheritance triple to your model:
Book-Library **is a** FOUNDATION/AssociationDetail
3. If your application uses a relational database, add the following inheritance triple:
Book-Library **is a** STORAGE/RelationalTable
4. In the Model Editor, select the triple where you inherit from AssociationDetail and click the Editor button .
This opens the Template Editor. You will use this editor to specify which entities in your model actually own the Book-Library entity.
5. In the Replaced By column for Owner, type `Book`, and in the Replaced By column for Owner2, type `Library`. Close the Template Editor and save your changes.
This adds the following triples to your model:
Book-Library **replaces** FOUNDATION/Owner
...by Book
Book-Library **replaces** FOUNDATION/Owner2
...by Library

The FOUNDATION pattern library

6. Add any non-key attributes you want for your AssociationDetail entity, including inheritance triples.
7. Generate and build your AssociationDetail entity and all of its scoped objects.
8. Ensure that you have defined records for both of the owning entities in your model (in this case, Book and Library).
9. To test the new functionality, run the Edit function.

Creating a wizard

Wizards use two patterns from the UISTYLE pattern library that take advantage of COOL:Plex's panel frames feature: UISTYLE/FrameWizard and UISTYLE/FrameChild. The wizard parent, FrameWizard, is the function that starts the wizard sequence. The child functions form the individual parts of the sequence.

In addition to FrameWizard and FrameChild, you can inherit from functions in the UIBASIC pattern library to customize the child functions in your wizard sequence.

Defining a wizard includes the following stages:

- Design the wizard sequence and define the functions involved
- Identify fields that will be used as global data among the functions in the wizard sequence
- Add custom code to enable the wizard functionality
- Modify the panels and captions of the child functions
- Call the wizard parent function to start the wizard

To define the wizard sequence:

1. Specify a function to be the starting point for the wizard, and inherit from UISTYLE/FrameWizard. For example, the Library Books model contains a function called ReserveABook:
ReserveABook **is a** UISTYLE/FrameWizard
2. For each function in the wizard sequence, inherit from UISTYLE/FrameChild, in addition to any other functions you need to inherit from to define your function, such as UIBASIC/FullGrid or UIBASIC/Input. For example:

Person.IdentifyPerson **is a** UISTYLE/FrameChild
Book.SelectBook **is a** UISTYLE/FrameChild
Book-Library.SelectLibrary **is a** UISTYLE/FrameChild
Reservation.SetReservationDate **is a** UISTYLE/FrameChild

3. To specify the functions in the wizard sequence, add a FNC **comprises** FNC triple for each child function in the order they'll occur in the wizard sequence. For example:

ReserveABook **comprises** Person.IdentifyPerson
ReserveABook **comprises** Book.SelectBook
ReserveABook **comprises** Book-Library.SelectLibrary
ReserveABook **comprises** Reservation.SetReservationDate

To specify fields common to all of the wizard sequence:

Rather than using input and output parameters, functions in a wizard use fields in the Local variable UISTYLE/SharedData to communicate information with other functions in the sequence. This variable appears in all functions that inherit from either FrameWizard or FrameChild. Because all child functions are running at once, they can use the global properties array that FrameWizard creates using the fields in SharedData.

1. Review your wizard design and identify all of the fields that will be needed by more than one function in your wizard sequence.

For example, child functions in the Reserve Library Book wizard share the fields Person ID, Person Name, Book ID, Library ID, Library Branch, Reserved Date, and TimeReserved.

The first function in the sequence, IdentifyPerson, captures a person's ID and retrieves the person's name, which is displayed in the second function, SelectBook. Similarly, the second function

saves a selected book, which the third function, `SelectLibrary`, uses to display a list of associated libraries.

2. Open the action diagram for each function in your wizard sequence, including the wizard parent, and drag the global fields you identified from the Object Browser to the Local variable `SharedData`.

`COOL:Plex` adds a triple to your model for each field you add to `Data`; for example:

```
Person.IdentifyPerson local Person ID
...for UISTYLE/SharedData
```

3. Close each action diagram and save your changes.

Note: You can add the `FNC local FLD` triples to each function directly, rather than dragging them into the variable in the action diagram, if you prefer.

To add custom code to the wizard part functions:

Because the functions in a wizard sequence communicate with each other by triggering events, processing does not always begin with the Initialization section of a function. Instead, the `FrameChild` pattern provides collection points where you can add processing at the beginning of a wizard function or at the end, when an end-user clicks `Next` or `Back` to move to another function.

- When you need to initialize values in a child function, open that function's action diagram and use the `End Show Panel` collection point.
- To add processing at the end of a child function, before moving to another function, add the code in the `Set Release Control Flag` collection point.

Creating a wizard

Note: If you do not want to exit the child function, set the field `FIELDS/ReleaseControl`, in the local variable `FrameChildL` to `No`.

To customize the wizard panels:

The size of the child site region on the parent panel determines the display size of child panels in your wizard sequence.

1. Open the panel scoped to each function in your wizard sequence in the Panel Designer.
 - In each child function, set the panel size to the size of the largest child panel in your wizard sequence.
 - In the parent function, set the size of the child site to the size of your child panels.
2. Close the Panel Designer and save your changes.

Tip: You can define a customized child function (for example, `FrameWizardPart`), that has panel regions for a bitmap image and instruction text. Then, each child function in your wizard can inherit from your new function instead of inheriting directly from `UISTYLE/FrameChild`. The Library Books model contains an example of such a function, called `FrameWizardPart`.

Putting it all together:

1. Choose a function in your application that will start the wizard, and add a call in that function to your wizard parent function.
2. Generate and build the functions in your wizard sequence and the function that calls the wizard parent function.
3. Run your application and test your wizard sequence.

Creating a property sheet

Property sheets use two patterns from the UISTYLE pattern library that use COOL:Plex's panel frames feature: UISTYLE/FrameProperty and UISTYLE/FrameChild. FrameProperty uses the ActiveX pattern ACTIVE/TabStrip to display the tabs on the property sheet. The parent function, based on FrameProperty, is the function that controls the property sheet. The child functions, based on FrameChild, form the individual parts.

In addition to FrameProperty and FrameChild, you can inherit from functions from the UIBASIC pattern library to customize the child functions in your property sheet.

Defining a property sheet includes the following stages:

- Design the property sheet structure and define the functions involved
- Specify images to display on the tabs (optional)
- Modify the panels of the child functions
- Identify fields that will be used as global data among the parts
- Add custom code to enable the property sheet parts
- Call the parent function to start the property sheet

To define the property sheet structure:

1. Specify a function to be the starting point, and inherit from UISTYLE/FrameProperty. For example, the Library Books model contains a function called LibrarySystemProperties:
LibrarySystemProperties **is a** UISTYLE/FrameProperty

2. For each child function, inherit from UISTYLE/FrameChild, in addition to any other functions you need to inherit from to define your function, such as UIBASIC/FullGrid or UIBASIC/Input. For example:

Person.PersonProperties **is a** UISTYLE/FrameChild
Book.BookProperties **is a** UISTYLE/FrameChild
Library.LibraryProperties **is a** UISTYLE/FrameChild
Publisher.PubProperties **is a** UISTYLE/FrameChild

3. To specify the functions for the property sheet tabs, add a FNC **comprises** FNC triple for each function in the order they'll occur on the tabs. For example:

LibrarySystemProperties **comprises** Person.PersonProperties
LibrarySystemProperties **comprises** Book.BookProperties
LibrarySystemProperties **comprises** Library.LibraryProperties
LibrarySystemProperties **comprises** Publisher.PubProperties

To specify images to display on the tabs (optional):

1. Identify the images you want to display on each of the tabs.
2. For each image in your list, add a FNC **option** NME triple to the scoped Scripts function you inherited from the image list control, in the order you want to place the images:

LibrarySystemProperties.Scripts **option** People
LibrarySystemProperties.Scripts **option** Books
LibrarySystemProperties.Scripts **option** Libraries
LibrarySystemProperties.Scripts **option** Publishers

Note: The Scripts function contains no code. Therefore, placing the **option** triples on this function cannot interfere with any other function options that could have been defined.

- For each NME object, click the Name button.
 - Under Narrative, type the name of a bitmap to display; for example, `./people.bmp`.
- Note:** The images must be in bitmap (.bmp) format and must reside in the directory where you will run your application. The dot before the name indicates the current directory. Notice the forward slash (/).
- Open the action diagram for your parent function, and go to the Set Tab Properties On Initialize collection point.
 - Add the following code to specify where to find the images to display on each tab:

```

Post Point Set tab properties on initialize
  TabStripL<Tab> contains the current index for the tab being initialized.
  Setting the TabStripL<Image> value to this number puts the corresponding image
  from the image list onto the tab.
Set TabStripL<Image> = TabStripL<Tab>
  
```

- Close the action diagram and save your changes.

To customize the child panels:

The unscoped name of each child function determines the text on the tab label for that function. To override this, you can add FNC **name** NME triples to specify the text to display on that function's tab in the property sheet. For example:

Person.PersonProperties **name** People

Book.BookProperties **name** Books

Library.LibraryProperties **name** Libraries

Publisher.PubProperties **name** Publishers

Creating a property sheet

The size of the child site region on the parent panel determines the display size of child panels in your property sheet.

- Open the panel scoped to each function in your property sheet in the Panel Designer.
 - In each child function, set the panel size to the size of the largest child panel.
 - In the parent function, set the size of the child site to the size of your child panels.
- Close the Panel Designer and save your changes.

To specify fields common to all members of the property sheet:

Rather than using input and output parameters, functions in a property sheet use the Local variable UISTYLE/SharedData to communicate information with each other. Because all child functions are running at once, they can use the global properties array that FrameParent creates using the fields in SharedData.

All functions that inherit from either FrameProperty or FrameChild have SharedData as a local variable. In each function, you only need to add the fields that are actually needed by that function to SharedData.

- Review your property sheet design and identify the fields that will be needed by functions on more than one tab.
- Open the action diagram for each property sheet function, including the parent function, and drag the global fields you identified from the Object Browser to the Local variable SharedData.

COOL:Plex adds a triple to your model for each field you add to SharedData; for example:

```
Person.PersonProperties local Person ID
... for UISTYLE/SharedData
```

3. Close each action diagram and save your changes.

Note: You can add the FNC **local** FLD triples to each function directly, rather than dragging them into the variable in the action diagram, if you prefer.

To add custom code to the property sheet functions:

Because the parent function activates child functions on a property sheet by triggering events, processing does not always begin with the Initialization section of a function. Instead, the FrameChild pattern provides collection points where you can add processing at the beginning of a child function or at the end, when an end-user clicks a tab to move to another function.

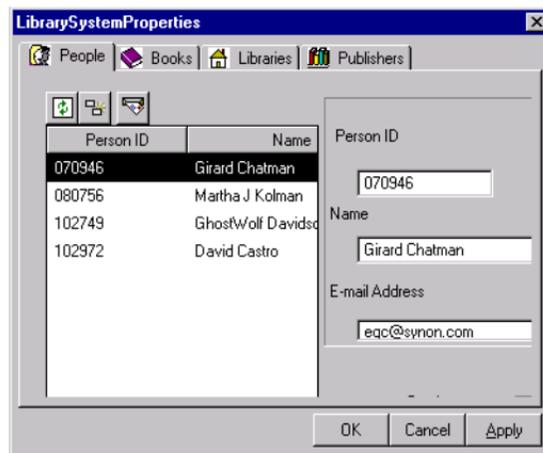
- When you need to initialize values in a child function, open that function's action diagram, and use the End Show Panel collection point.
- To add processing at the end of a child function, before moving to the next function in the sequence, add the code in the Set Release Control Flag collection point.

Note: If you do not want to exit the child function, set the field FIELDS/ReleaseControl, in the local variable FrameChildL to No.

Putting it all together:

1. Choose a function in your application that will start the property sheet, and add a call in that function to the parent function.
2. Generate and build the functions in your property sheet, and the function that calls the parent function.
3. Run your application and test your property sheet.

Note: To test your property sheet independently of the function that calls it, choose Create Exe from the Build menu, and run the executable program from Windows Explorer.



Creating an MDI parent

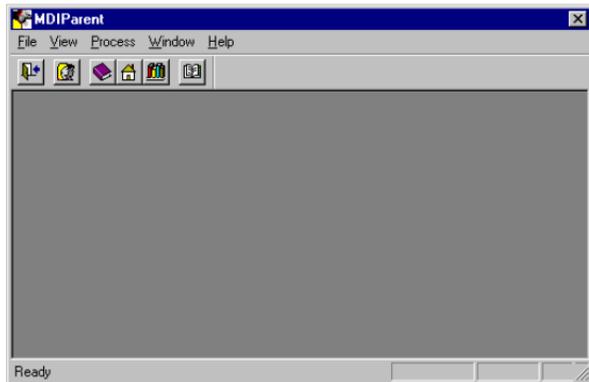
OBASE analogue: This function is analogous to Default Objects.MDI Template.

To create an MDI application:

1. Add an inheritance triple to an unscoped function in your model that will become the entry point for your application. For example:
BookMenuMDI **is a** UIBASIC/MDI
2. Add the following triples to create a recognizable name for your application so that you can run it as a separate executable program:
BookMenuMDI.MDIParent **impl name** BOOKS
BookMenuMDI.MDIParent **file name** BOOKS
3. Open the panel design for BookMenuMDI.MenuShell.
4. Add menu items and toolbar buttons for the common elements of your application. Link the toolbar buttons to the corresponding menu items. Remember to specify a menu ID for each new menu item.
5. Close MenuShell and save your changes.
6. Open the panel design for the panel scoped to BookMenu.MDIParent. Add events for the menu items you defined on your MenuShell. Close the panel design and save your changes.
7. In the action diagram for BookMenu.MDIParent, go to the Events collection point and add processing for each event you defined.

Creating an MDI parent

8. For each of the child windows in your application, add an inheritance triple for your MenuShell. For example, if you have a FOUNDATION/EditDialog entity called Book and a FOUNDATION/EditDetail entity called Library, add the triples:
Book.EditSuite.Grid.Panel **is a** BookMenuMDI.MenuShell
Library.Edit.Panel **is a** BookMenuMDI.MenuShell
9. Generate and build.
10. Test your application by running your MDIParent function.



11. To turn your application into a separate executable program, select your MDIParent function, and choose Create Exe from the Build menu.
You can then run your program by double-clicking the file BOOKS.exe.

Customizing an ActiveX ToolBar

The pattern ACTIVE/ToolBar displays a customizable toolbar with images, text, or both. You can inherit from this function to create a top-level interface for your application as an alternative to using MDI parent and child windows.

Defining a toolbar consists of:

- Specifying the toolbar buttons
- Defining events for the buttons
- Specifying the functionality associated with each button

To specify toolbar buttons:

1. Define a function that inherits from ToolBar; for example:
BookMenu **is a** ACTIVE/ToolBar
2. Add a FNC **option** NME triple for each button on the toolbar. The source of the triple is the Scripts function you inherited from the image list control, which is scoped to your menu function:
BookMenu.Scripts **option** People
BookMenu.Scripts **option** Books
BookMenu.Scripts **option** Publishers

The names you enter will become the labels on the toolbar buttons, in the order you entered them.

Note: The Scripts function contains no code. Therefore, placing the option triples on this function cannot interfere with any other function options that could have been defined.

3. For each of these triples, select the target object and click the Name button.
4. Under Narrative, type the name of a bitmap to display; for example, `./people.bmp`.

Note: The images must be in bitmap (.bmp) format and must reside in the directory where you will run your application. The dot before the name indicates the current directory. Notice the forward slash (/).

5. Open the Panel scoped to your toolbar function and edit the panel title. Change the size of the ToolBar control to match the approximate sizes of the bitmaps you specified.
6. Close the panel design and save your changes.

To define events for the toolbar buttons:

1. Define a field that inherits from FIELDS/Button, and add values corresponding to each button on your toolbar. For example:
BookMenuEvent **is a** FIELDS/Button
BookMenuEvent **value** People
BookMenuEvent **value** Books
BookMenuEvent **value** Publishers

2. Assign numeric values for the large property of each value, beginning with 1.

You can ignore any warnings that your field has duplicate literals.

To define processing for each toolbar button:

1. Open the action diagram for your toolbar function, and go to the Start Toolbar Button Clicked collection point.
2. Create a Local variable and drag your event field (in this case, BookMenuEvent) to it.
3. Add the following action diagram code to specify the function to call for each toolbar button:



4. Close the action diagram and save your changes.

Putting it all together:

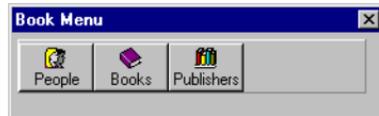
1. Add implementation name and file name triples for your toolbar function:

BookMenu **impl name** BookMenu

BookMenu **file name** BookMenu

2. Generate and build your toolbar function.

3. In the Generate and Build window, right-click on your function and choose Create Exe from the pop-up menu.
4. In Windows Explorer, locate the executable program you just created (in this case, BookMenu.exe) and double-click it to test the new functionality.



Common pattern library fields

Date and time fields

- *DATE/CheckedDateISO* – A date field in ISO date format. The actual format depends on your system’s configuration. This is the date format you will probably use most often.
- *DATE/CheckedDate7* – A 7-digit numeric field that represents dates in the form CCYYMMDD. For example, August 7, 1998 is 0980807 in this format. Inherit from this field if your database uses dates in this format.
- *DATE/CheckedDate8* – An 8-digit numeric field that represents dates in the form CCYYMMDD. For example, August 7, 1998 is 19980807 in this format. Inherit from this field if your database uses dates in this format.
- *DATE/CheckedTimeISO* – A time in ISO time format. The actual format depends on your system’s configuration.
- *DATE/CheckedTime6* – A 6-digit numeric field that represents a time in the format HHMMSS. Inherit from this field if your database uses times in this format.

Text fields

- *FIELDS/Identifier* – A 10-character field that you can use for the unique identifier of an object.
- *FIELDS/ShortDescription* – A 20-character field that you can use for the description of an object.
- *FIELDS/LongDescription* – A character field with varying length that you can use for an extended description. When you inherit from this field, add a **length** triple.

Numeric fields

- *FIELDS/Number* – A numeric field with a length of 9 and C format Integer. Inherit from this field for quantities up to 9999.
- *FIELDS/Long* – A numeric field with a length of 9 and C format Long. Inherit from this field for quantities greater than 10,000.
- *FIELDS/Price* – A numeric field with a length of 9 and 2 decimal places. Inherit from this field to represent the price of an item.
- *FIELDS/Surrogate* – A system-assigned numeric key for an entity. When you inherit from FOUNDATION/Surrogate, you replace this field with the key field for your entity.

Other useful fields

- *FIELDS/Status* – A field with a finite set of values from which you want end-users to select. By default, Status fields appear as a Combo box on a panel. If you want validation, inherit from FIELDS/CheckedStatus instead.
- *FIELDS/CheckedStatus* – A Status field with built-in validation. When you inherit from this field, you must generate and build the scoped Check function.
- *FIELDS/YesNo* – A Status field with the values Yes and No.
- *FIELDS/LanguageSupport* – Facilitates translation for other national languages. Inheriting from this field scopes a label object to the field, along with left and top labels, which take their values from the literal value of the label object.