

AppBuilder  
By Magic Software Enterprises

# Magic Software AppBuilder

Version 3.2

## Developing Applications Guide

Corporate Headquarters:

Magic Software Enterprises  
5 Haplada Street,  
Or Yehuda 60218, Israel  
Tel +972 3 5389213  
Fax +972 3 5389333

© 1992-2013 AppBuilder Solutions

All rights reserved.

Printed in the United States of America.

AppBuilder is a trademark of AppBuilder Solutions. All other product and company names mentioned herein are for identification purposes only and are the property of, and may be trademarks of, their respective owners.

Portions of this product may be covered by U.S. Patent Numbers 5,295,222 and 5,495,610 and various other non-U.S. patents.

The software supplied with this document is the property of AppBuilder Solutions and is furnished under a license agreement. Neither the software nor this document may be copied or transferred by any means, electronic or mechanical, except as provided in the licensing agreement.

AppBuilder Solutions has made every effort to ensure that the information contained in this document is accurate; however, there are no representations or warranties regarding this information, including warranties of merchantability or fitness for a particular purpose. AppBuilder Solutions assumes no responsibility for errors or omissions that may occur in this document. The information in this document is subject to change without prior notice and does not represent a commitment by AppBuilder Solutions or its representatives.

1. Developing Applications Guide	2
1.1 Introduction to Developing Applications	2
1.2 Planning the Application	3
1.2.1 The Development Process	3
1.2.2 Customizing the Environment	6
1.3 Data Modeling	7
1.3.1 Understanding Engineering Diagrams	7
1.3.2 Understanding Entity Relationships	8
1.3.3 AppBuilder Engineering Tools	13
1.4 Diagramming the Application	20
1.4.1 State Transition Diagram	20
1.4.2 Process Dependency Diagram	21
1.4.3 Window Flow Diagram	22
1.5 Designing the Application	37
1.5.1 Rules and Rules Language	39
1.5.2 Designing Rules	40
1.5.3 Writing AppBuilder Rules	47
1.5.4 Other Considerations for Rules	54
1.6 Creating Event-Driven and Converse Applications	55
1.6.1 Using Converse Statement (C Client) Rules	55
1.6.2 Using Event-Driven (Java) Rules	58
1.6.3 Using Subscription (POST Statement)	58
1.6.4 Using the System View	62
1.6.5 Event-Driven and Converse Processing Example	63
1.7 Handling Display Rules	73
1.7.1 Standard Display Rules	73
1.7.2 Display Rules for Thin (HTML) Client	73
1.7.3 Non-Display Rules for Java Client	74
1.7.4 Display Rules with Third-Party Java Bean	74
1.7.5 User Events Handled by Java Bean	75
1.7.6 Rules Controlling Converse Window	76
1.7.7 Converse Events for Java	78
1.7.8 Domain Sets for Window Objects	79
1.8 Native Files Handling	80
1.8.1 Understanding Native Files	81
1.8.2 Understanding File Organization Types	81
1.8.3 Adding and Defining Records from Hierarchy	83
1.8.4 Configuring a Rule to access a Data Source	85
1.8.5 Writing Rules Code to Access Files	85
1.8.6 Package Relationships for Native File Entities and Relationships	86
1.9 Working with User Components	86
1.9.1 Using Subroutine Components	87
1.9.2 Specifying the Component Includes Directory	87
1.9.3 Advantages of Using User Components	88
1.9.4 Using User Components	88
1.9.5 Guidelines for Using User Components	88
1.9.6 Adding a User Component	89
1.9.7 Data Type Comparison	90
1.9.8 Writing a Java User Component	93
1.9.9 Writing a C User Component	94
1.9.10 Calling a C Component from Java	95
1.9.11 Using Sample Component Code	98
1.10 Working with System Components	99
1.11 Creating User Assistance for Applications	102
1.11.1 AppBuilder Internal	102
1.11.2 AppBuilder External	106
1.11.3 Enabling Application Help	109
1.12 Data Type Support	109
1.13 Events Supported in C	113

# Developing Applications Guide

AppBuilder is a repository-based development environment and tool set that let you develop applications while insulating that development from particular languages or environments. AppBuilder began as a tool for developing and updating procedural applications. It can now be used for both procedural and object-oriented applications or for converting procedural to object-oriented applications.

This guide contains the information required to develop your applications using AppBuilder tool set.

The guide includes the following sections and topics:

- [Introduction to Developing Applications](#)
- [Planning the Application](#)
- [Data Modeling](#)
- [Diagramming the Application](#)
- [Designing the Application](#)
- [Creating Event-Driven and Converse Applications](#)
- [Handling Display Rules](#)
- [Native Files Handling](#)
- [Working with User Components](#)
- [Working with System Components](#)
- [Creating User Assistance for Applications](#)
- [Data Type Support](#)
- [Events Supported in C](#)

## Introduction to Developing Applications

### Introduction

AppBuilder is a repository-based development environment and tool set that provides model-driven, platform-independent application development. It uses advanced code generation to allow you to decide your target deployment architecture, independent of your business application logic.

AppBuilder keeps your application development insulated from technology change; this is one of the core values of AppBuilder. In particular, AppBuilder:

- Provides a robust integrated development environment for planning and modeling applications
- Insulates application development from programming language and technology changes
- Allows you to focus your development effort on business problems rather than technology issues
- Improves productivity and quality with model-based development
- Provides a simpler rules language designed for business analysts
- Extends the lifespan of business assets by allowing the business application to mature and evolve with your business
- Makes reuse possible with less effort because AppBuilder uses repository-managed object-based components
- Facilitates rapid development and quick time to market with an integrated toolset
- Allows flexible development with a choice of deployment to C, C#, Java, HTML/Servlet, EJB, SOAP/Web Services, RMI, OpenCOBOL, and ClassicCOBOL on CICS, IMS and Batch
- Provides options for using a number of web services

The AppBuilder development environment includes:

- Modeling, Drawing and Design Tools
- Entity Relationship Diagram
- State Transition Diagram
- Database Diagram
- Process Dependency Diagram
- Window Flow Diagram
- Matrix Builder
- Window Painter
- Rule Editor
- Set Builder
- Configuration Designer
- Debugging tools (C and JavaRule View and tracing)



For more information on the operations of the Workbench and its capabilities, see the *Development Tools Reference Guide*.

The cornerstone of AppBuilder is its model-based repository technology. The repository stores everything concerning the development of an

application, such as analysis models and diagrams, variable definitions, window designs, application code and deployment options. One of the most important things that a repository stores is the interdependencies between any of these objects. These interdependencies are known as relationships, and they are the glue that binds AppBuilder application together. There are 3 types of repository:

- Personal Repository
- Workgroup Repository
- Mainframe Repository

Each repository stores your development artifacts (entities) as well as the relationships between them. It also provides a structural representation of your application, thereby allowing:

- Impact analysis
- Rebuild analysis
- Scriptable transformations
- Visualization
- Re-use
- Object level security



For more information on repositories and their operations, see the *Repository Administration Guide for Workgroup and Personal Repositories*.

## Using AppBuilder

AppBuilder has proven to be an effective way to develop applications while insulating that development from particular languages or environments. AppBuilder began as a tool for developing and updating procedural applications. It can now be used for both procedural and object-oriented applications or for converting procedural to object-oriented applications.

## Planning the Application

AppBuilder is a robust, integrated toolset for designing, implementing, and maintaining high-volume, multi-platform, distributed applications. With AppBuilder, you can develop full Java, C#, and C clients and servlet-based HTML clients and Enterprise Java Bean (EJB) functionality in a standardized, supported environment. AppBuilder provides the ability to design and develop eBusiness applications independent of the deployment platform or architecture. Applications developed with AppBuilder can be deployed across multiple customer-to-business and business-to-business architectures, as well as traditional client/server environments.

There are many steps to developing applications with AppBuilder, including constructing the logic, designing user interfaces, and adding pre-defined components. This guide describes these development steps for applications that will be deployed in multiple execution platforms. AppBuilder users should have basic application development experience, some familiarity with application development environments, and familiarity with Microsoft Windows? operating systems.

Refer to the following AppBuilder documentation for more information:

- *Getting Started Guide* for creating a basic AppBuilder application
- *Development Tools Reference Guide* for guidance on how to use specific tools in the Construction Workbench
- *Rules Language Reference Guide* for complete information on the AppBuilder procedural Rules Language
- *OO Rules Language Reference Guide* for information about AppBuilder's OO Rules Language
- *Multi-Language User Interface Guide* for designing applications for multiple languages
- *Deploying Applications Guide* for building and deploying the developed application
- *Debugging Applications Guide* for debugging and troubleshooting an application

## The Development Process

The AppBuilder environment enables a high degree of reuse in the data modeling and database design phases of software development. A typical AppBuilder development process consists of the following steps:

1. [Analyze](#) - Model the data and processes of a system.
2. [Design](#) - Design the database and model the flow of the application.
3. [Construct](#) - Develop the application hierarchy, windows and views, and business logic. Use (or re-use) system components and user components.

The AppBuilder development environment includes analysis tools for data and process modeling and database design. It includes a fully-integrated suite of construction tools for writing application code, building the graphical interface, and specifying the application's configuration. It also includes convenient tools for testing and debugging applications built in the workbench.

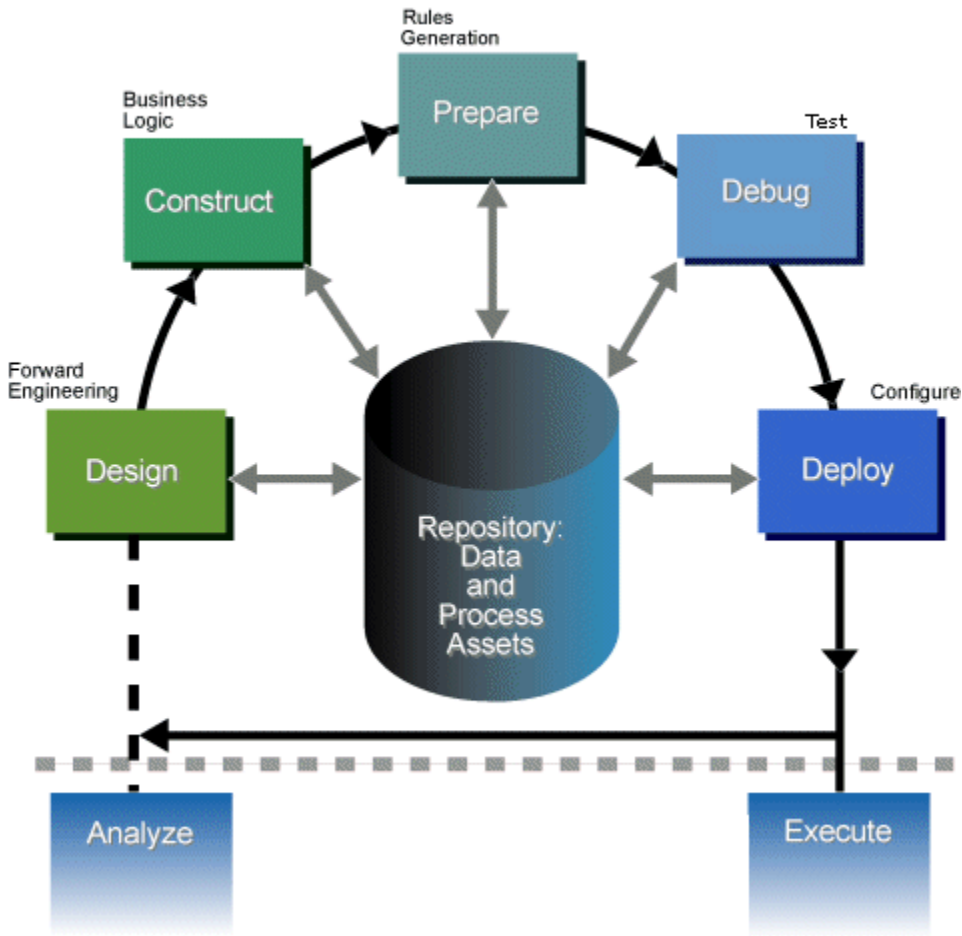
The deployment process (covered in more detail in the *Deploying Applications Guide* and *Debugging Applications Guide*) consists of the following steps:

1. [Prepare](#) - Generate the rules and windows.
2. [Debug](#) - Test and debug the rules and application.
3. [Deploy](#) - Configure the application for the target environment.

4. [Execute](#) - Run the application on the selected environment.

**Development and deployment process**

**AppBuilder Development Model Processes**



**Analyze**

Before you develop a new application, you should thoroughly analyze the current business environment. This includes examining all aspects of the current business process and defining a target. From this analysis, you can develop several deliverables, including work flow models, organization structure, and migration plans.

**Design**

After determining your needs, you can use several AppBuilder design tools to move from the data you have collected to generate a data model. The design tools are described in the following table:

**AppBuilder design tools**

Tool	Usage
Entity Relationship Diagram	An Entity Relationship Diagram (ERD) is a representation of the entities and relationships within a system. The Entity Relationship Diagram tool can be used to produce a logical data model. Typically, the diagram contains labeled boxes representing entities with lines between the boxes representing relationships.
Database Diagram	AppBuilder provides a Database Diagram tool for modeling a database by modifying tables and their primary, index, and foreign keys.

Process Dependency Diagram	A diagram that illustrates the dependency of business activities (processes) on each other and the events that trigger their initiation. The PDD shows a sequenced set of actions that occur in response to a given event stimulus.
State Transition Diagram	A State Transition Diagram is a model that identifies the stages (or states) a data object goes through in its lifecycle, and the external and internal business events that trigger a set of processes and cause the data object to move from one state to another. A State Transition Diagram constitutes a data object's control view, because it captures business rules that translate into system controls.

With AppBuilder, you can build an ERD that contains boxes representing the entities and lines representing the relationships between them, much like a flow chart. This is illustrated in the next chapter on [Data Modeling](#). You can then forward engineer the ERD to translate the ERD logical entities (entities, relationships, and identifiers) to a relational data model consisting of tables, columns, and keys in a Database Diagram (DBD). This can then be transformed into rule and view objects that can be shown in the Hierarchy Window. More details about using the engineering and design tools to diagram an application are provided in [Diagramming the Application](#). Read more about the Design Tools in Construction Workbench in the *Development Tools Reference Guide*.

## Construct

After creating a data model for your application, use the AppBuilder Construction Workbench to create the application business logic. This includes building the hierarchy, creating the user interface, and writing the rules that define the application logic. The primary construction tools are described in the following table.

### AppBuilder tools for creating application business logic

Tool	Usage
Hierarchy Window	A window in the Construction Workbench that can be used to create application objects and their relationships. It shows existing objects and their relationships in a repository.
Window Painter	A Construction Workbench tool for designing and developing an application's end-user interface.
Window Flow Diagram	A Construction Workbench tool for modeling window flow and user interaction during the prototyping phase.
Rule Painter	A Construction Workbench tool for writing Rules Language source code. This source code defines the logic of the application.
HTML Editor	A Construction Workbench option allowing you to select an HTML editor when the current project's interface is generated in HTML.

Refer to the *Development Tools Reference Guide* for more information about these tools. The sections of this manual dealing with [Designing the Application](#), [Creating Event-Driven and Converse Applications](#), [Handling Display Rules](#), [Working with User Components](#), and [Working with System Components](#) provide information about developing applications with these objects.

During this step you can use or reuse system components delivered with AppBuilder or user components. For information, refer to the *System Components Reference Guide*.

## Prepare

After designing and constructing the application, you must transform the rules into an executable program. Preparing the application creates the runtime interface which includes menus, icons, and windows that the end-user sees. AppBuilder has a number of preparation tools to assist in this process. See the *Deploying Applications Guide* for more detailed information about preparing and deploying an application.

### Generate Rules

AppBuilder can generate platform-specific code (C and C# for PC, and COBOL for mainframe), or it can generate Java code. The generated code must be compiled in a similar environment to which it executes (for example, a remote iSeries, UNIX, or Windows machine). Even when compilation takes place on another machine ( *remote preparation* ), it is initiated from your local Construction Workbench. The system transmits the preparation job to the remote machine where it is compiled.

Refer to [Designing the Application](#) for details about creating rules. Refer to the *Rules Language Reference Guide* for complete information about the Rules Language.

### Preparation

When you prepare your project, the system performs the following actions:

- Transforms the rules written in the Construction Workbench into an executable program.
- Compiles the source code of any user components - third-generation language routines - your rules use.
- Prepares any files your rules access and creates their corresponding database tables.
- Prepares any sets that the application uses.
- Makes available to the runtime environment the menus, icons, windows, and workstation reports that comprise your application end-user interface.

You can use the Construction Workbench to prepare mainframe rules for test execution on the workstation, and to check the syntax of mainframe reports. You can also prepare to remote servers, whether mainframe, UNIX, or Windows. The Status window of the Construction Workbench provides feedback information about the status of each object during preparation and execution. Refer to the *Development Tools Reference Guide* for information about the Status window. For additional information about creating and specifying deployment configurations, refer to the *Deploying Applications Guide*.

## Debug

After preparing the rules and windows, you can debug rules locally or remotely. AppBuilder offers the RuleView Debugger tool for debugging the application in Java or C. For information about debugging, refer to the *Debugging Applications Guide*.

For stand-alone applications, you can debug rules locally with the RuleView. RuleView has the following debugging functionality:

- Starts the execution of a rule
- Breaks on most lines of rule source code
- Skips a step during the execution of a rule (Windows C only)
- Examines and changes the contents of any field within a view
- Reviews the source code for the active rule or any rule in its data universe

For Java applications, AppBuilder includes a full-featured source debugger that is built on top of the Java Platform Debugger Architecture (JPDA) from Sun Microsystems. This debugger supports viewing and modifying AppBuilder data, setting break points, and other standard debugger functionality. For C Language applications, the RuleView debugger is available. Refer to the *Debugging Applications Guide* for more information about the RuleView debuggers for C, C# (.NET) and for Java.

## Deploy

After testing and debugging your project, you must specify an application configuration that identifies the specific target environment for your project. You can prepare client or server rules for execution on diverse operating systems and machines, against different database management tools. Refer to the *Deploying Applications Guide* for more information about packaging and deploying distributed applications.

### Rules Generated to C

When client or server rules are generated to C, the following files and resources are created:

- An executable module (dll) for each rule and user component in the application
- A display file (panel file) for each window
- Resources used during execution

### Rules Generated to Java

Event-driven applications (for example, Java server with Java client or thin-HTML client) can be deployed in a variety of ways. They can be downloaded from a Web server as needed or in the form of an entire JAR file. Traditional deployment by way of a software distribution system is also possible. Mixed deployment is possible so that a JAR file containing the main executables is downloaded while other files, such as debug files, are left on a Web server.

### Rules Generated to C#

When client or server rules are generated to C#, an executable module (exe) for each rule and user component is created.

## Execute

Depending on the application configuration, you can start the application from a program group, from the command line, or from a pull-down menu option.

When you prepare a C client and specify the function's start from a desktop object, the function name is listed under the *Programs* choice on the Windows *Start* menu. When you select the function, a cascaded menu appears listing each child process. Select a process to run it.

When you prepare the function to start as a menu bar choice, the function name appears in the *Execution Client* menu bar along with the names of any other functions you have prepared (if that option is set). Each child process is displayed below it as a menu item. To start the execution client, run *Start > AppBuilder > Execution Clients* and select the appropriate Java or C client. At this point, the menu comes up with the function names. After selecting the menu item, the client can select and run the particular process.

## Customizing the Environment

Before developing an application in the Construction Workbench, you can customize the development environment with optional configuration settings.

### Workbench Options

Use the Workbench Options window to specify the various Construction Workbench options for each tool (for example, display options, defaults, or confirmations). To access these options, select *Tools > Workbench Options* from the Construction Workbench menu bar. For a complete explanation of the Workbench Options, refer to the *Development Tools Reference Guide*.



## Initialization Settings (INI Files)

Most settings for the Construction Workbench are set within the Workbench Options dialog. A few of the settings can be modified by editing the system initialization file (hps.ini) before running the Construction Workbench. You can edit and set configuration or initialization parameters that affect the development environment. To access these parameters, edit the hps.ini file using the INI File Editor available from the Management Console. Refer to the *Communications Guide* for an explanation of how to run the INI File Editor and the *INI Settings Reference Guide* for explanations of the settings in hps.ini.

## Window Arrangement

One way to customize the development environment involves arranging the window areas in the Construction Workbench. Refer to the *Development Tools Reference Guide* for information about the window areas.

# Data Modeling

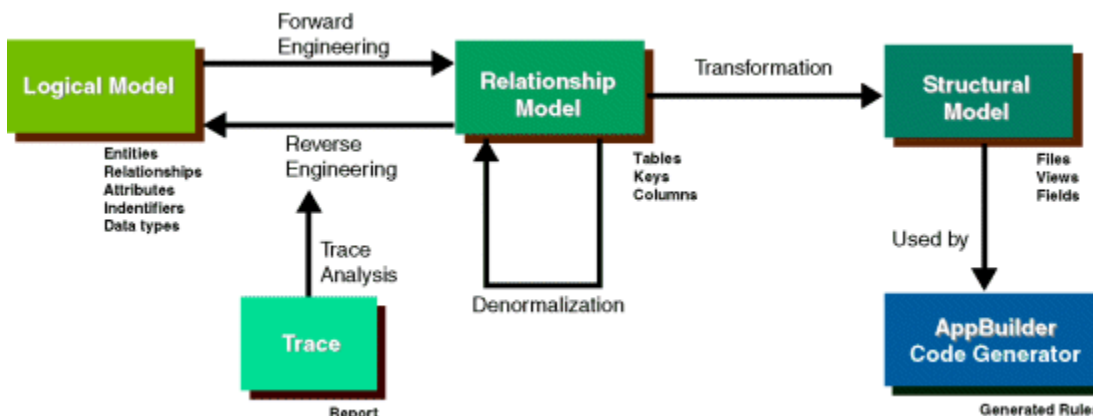
## Data Modeling

The AppBuilder environment enables a high degree of reuse in the data modeling and database design phases of software development. The AppBuilder engineering toolset can automatically:

- Forward engineer a logical model represented in an entity relationship diagram (ERD) into a relational model represented in a database diagram (DBD)
- Track through a trace analysis report to discover how the logical entities in an ERD correspond to the relational entities in the database model
- Copy a column from one table to another through a foreign key by denormalizing, thereby improving database performance
- Reverse engineer the refined relational model in a DBD back to the logical model in an ERD, so that the relational and logical models reflect each other
- Track in a trace analysis report how the relational entities in the database model map back to logical entities in an ERD
- Forward and reverse engineer the same model as many times as you want
- Transform a relational model into a structural model (the data structures that AppBuilder rules use to read from and write to database tables)

[Moving from abstract design to generated rules](#) illustrates how the different data representations interrelate, from more abstract on the left, to less abstract on the right.

### Moving from abstract design to generated rules



The topics in this chapter include:

- [Understanding Engineering Diagrams](#)
- [Understanding Entity Relationships](#)
- [AppBuilder Engineering Tools](#)

Refer to the *Development Tools Reference Guide* for information about how to use the engineering tools in AppBuilder.

## Understanding Engineering Diagrams

Each step in the engineering process begins with the representation of data; this input can be a drawing, such as an Entity Relationship Diagram (ERD), a Process Dependency Diagram (PDD), a Database Diagram (DBD), a Window Flow Diagram (WFD), or objects in a hierarchy. These diagrams provide useful representations of objects, relationships, processes, windows, etc. so that the development of an application can be methodically tracked.

While some kinds of diagrams are purely used for modeling, other kinds of diagrams provide a path for developing directly from them. For

example, you can forward engineer from either an ERD or from the hierarchy window. You can then start forward trace analysis from an ERD. You can start denormalization, reverse engineering, reverse trace analysis, and transformation from a DBD. Forward engineering and reverse engineering are tightly coupled processes that require you to start with a model representation (ERD, DBD, or hierarchy) that has all the "from" and "to" entities. Thus, you should plan to forward engineer and reverse engineer in the same repository because you can successfully reverse engineer a downloaded or migrated DBD only if the repository contains all of the corresponding logical models\* and traceability information. Your project management team should establish naming standards so that you will know how to query for the objects you need for the engineering tools.

\*The logical model can also exist in the entities in a hierarchy, even though they have not been constructed into an ERD.

The following sections provide more information about these diagrams:

- [Entity Relationship Diagram \(ERD\)](#)
- [Database Diagram \(DBD\)](#)

## Entity Relationship Diagram (ERD)

You can start with an ERD for these processes:

- Forward engineering
- Forward trace analysis

The three main components of an ERD are:

- *Entity* - person, place, or thing for which data is collected
- *Relationship* - action between the entities
- *Cardinality* - description of the type of relationship between the entities

To create an Entity Relationship Diagram:

### ***Identify the entities.***

1. Determine all significant interactions ? the relationships.
2. Analyze the nature of the interactions ? the cardinality.
3. Create the ERD.

## Database Diagram (DBD)

You start with a Database Diagram for these processes:

- Reverse engineering
- Reverse trace analysis
- Transformation
- Denormalization

Use the DBD when you have the relational database model of tables, columns, and keys, and you want to create from that the logical model of entities and relationships for your application. This process is referred to as *reverse engineering*.

The DBD is usually the end result of forward engineering an Entity Relationship Diagram (ERD), but the DBD could be imported from another source and reverse engineered to create the logical entities for your application. Reverse engineering maps the DBD's relational entities back to the logical entities in an ERD as follows:

Tables generate entities.

Columns in the primary key specify the entity type ? kernel, characteristic, associative, or intersection\*.

Foreign keys either preserve a one-to-one or generate a many-to-one relationship.

Primary and Index keys are mapped back to primary and alternate identifiers.

Columns not in a Foreign key are reverse engineered into simple attributes (unless a complex attribute existed previously).

Intersection entities are constructed for many-to-many relationships.

\*For definitions of terms, see the Product Glossary in the *Getting Started Guide*.

# Understanding Entity Relationships

This section describes the entity relationships for use with the engineering drawing tools. For specific information about these drawing tools, refer to the *Development Tools Reference Guide*.

Forward engineering transforms the 'from' and 'to' entities from an abstract, platform-independent model into a database-specific model. When forward engineering is executed (at design-time), it adds columns to the 'from' and 'to' entities so that in a relational database, they become linked to each other in a way that is efficient and minimizes the duplication of information. The "from" entity identifier, entity's primary key, is copied to the "to" entity, where it becomes a foreign key. A foreign key is one or more columns that uniquely identify rows in another table; this associates two entities through a relationship. The cardinality of a relationship determines which entities (source/"from" or target/"to") will hold a foreign key reference to the other.

### ***Relationships between entities***

---

Relationship	Description
<a href="#">One-to-One Relationship</a>	The source ("from") entity identifier is copied to the target ("to") entity. The one mandatory cardinality dictates that an entity must have one relationship with another entity. For example, an automobile entity must have one and only one engine. The one optional cardinality means that an entity may have no relationship or one relationship with another entity (for our example meaning that the automobile doesn't have to have an associated engine).
<a href="#">One-to-Many Relationship</a>	In a one-to-many relationship, the primary key for the source entity on the one side is added to the target entity table on the many side so that it can hold a reference to its parent entity. The parent does not hold references to the children because there could be an infinite number of them. For example, a customer might have one or more addresses.
<a href="#">Many-to-Many Relationship</a>	In a many-to-many relationship, a new association table is created with the foreign key of the source entity, and the foreign key of the target entity. An example should be that, to an automobile, any electrical part might be wired to any other electrical part and vice-versa.
<a href="#">Supertype-to-Subtype Relationship</a>	The primary key of the supertype is copied to the subtype, and the subtype adds additional columns as necessary.

See [Sample Entity Relationship Diagram \(ERD\)](#) for a sample Entity Relationship Diagram.

### One-to-One Relationship

Forward engineering uses the following algorithm to resolve both optional-one to optional-one and mandatory one-to-one relationships:

**Any non-kernel entity inherits from the kernel entity.**

1. Look at the volumetric information and try to inherit to the side that has fewer expected rows.
2. If steps 1 and 2 fail, try to inherit to the side that has fewer expected maximum rows.
3. If step 3 fails, try to inherit to the side that has fewer expected minimum rows.
4. If steps 1 through 4 fail, randomly select an entity to inherit to. Usually, but not always, you select the entity created first.

Steps 1 through 3 try to inherit to the entity that uses the least amount of storage. Steps 1 through 3 fail in a mandatory one-to-one relationship because both entities should have identical volumetric information. Step 4 tries to inherit to the entity that is dependent on the kernel entity if there is one.

If the volumetric information changes, the "from" and "to" entities change roles; therefore, it is safer to upload and download both sides of an optional-one to optional-one or a one-to-one relationship, regardless of which side gets inherited.

### One-to-Many Relationship

In this type of relationship, forward engineering creates two table entities, one for each entity in the original ERD. The primary identifier of each entity becomes a primary key attached to its table. Any candidate or associate identifiers become index keys, and a foreign key is created to represent the relationship between the two entities.

### Many-to-Many Relationship

In this type of relationship, forward engineering creates one table for each entity and an additional table containing intersection data (the relationship). The name of the intersection table is based on the name of the relationship between the two entities.

### Supertype-to-Subtype Relationship

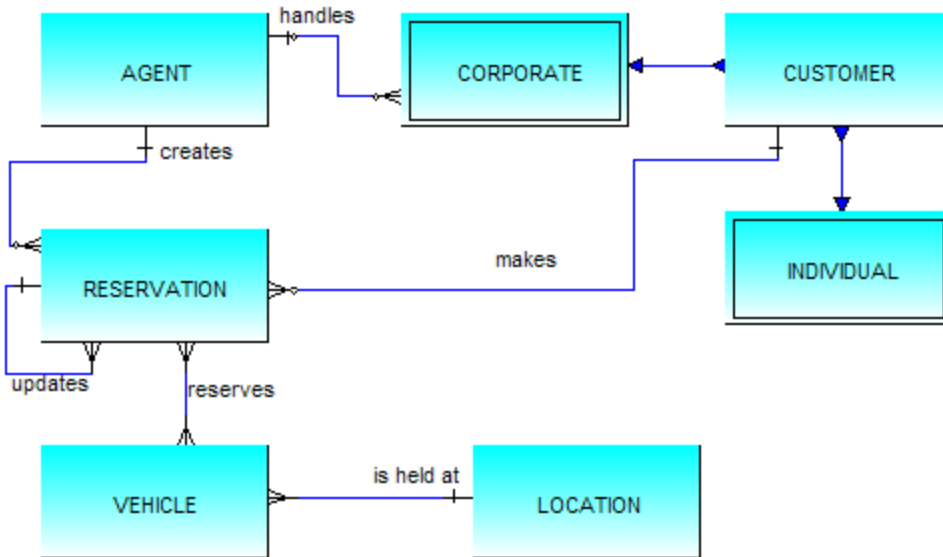
In this type of relationship, forward engineering creates one table for supertype and one table for each subtype. The supertype's table contains columns for its attributes and a primary key for its identifier. The subtype tables have columns for their own attributes, as well as columns corresponding to all the supertype's primary attributes.

### Sample Entity Relationship Diagram (ERD)

[Sample ERD illustrating "from" and "to" entities](#) shows a sample ERD with entities that have relationships with other entities that are not in the drawing but are in the repository. This example uses the different relationship types that are discussed above. The example begins with building a logical model for an automobile rental agency. Sections below illustrate building and forward engineering the ERD.

Using the rules listed above, the *AGENT* entity in [Sample ERD illustrating "from" and "to" entities](#) is a "from" entity, because it is on the "one" side of the one-to-many relationship. Because *AGENT* is the "from" entity, you do not need the "to" entity, *RESERVATION*. In contrast, *VEHICLE* is a "to" entity because it is on the "many" end of a one-to-many relationship. You need its "from" entity, *LOCATION*, before you forward engineer this drawing. The *CORPORATE* subtype entity? a "to" entity? and its "from" entity, *CUSTOMER*, are already in the drawing. You do not need the other subtype entity, *INDIVIDUAL*, because it is a "to" entity.

**Sample ERD illustrating "from" and "to" entities**



To create the ERD:

**In Construction Workbench, Select File>New. The Create New dialog opens.**

- Select Entity Relationship Diagram and click the OK button. A new Entity Relationship Diagram window appears in the Construction Workbench.

You can specify settings for engineering diagrams in the Construction Workbench by selecting Tools>Workbench Options. The settings for diagrams are on the Drawing Tools tab.

- Add the entities to the diagram. Name them: Agent, Vehicle, Customer, Reservation, Location, Individual, and Corporate.

**Click on the Entity button in the ERD toolbar or select from the menu Insert > Entity.**

1. Click inside the ERD where you want to place the Entity. The Entity is displayed as a gray box until you define the Entity.
2. Double-click the Entity. The Insert Entity dialog is displayed.
3. Click the Query button to display a list of entities in the repository or type a new entity name and click Insert.
4. Arrange them in the diagram in a similar way to the figure above. You can select any object in the diagram and drag it to a new location.
5. Add the relationships between the entities to the diagram.

**Click the Relationship button in the ERD toolbar or select from the menu Insert > Relationship.**

1. Click inside the Entity from which the relationship starts, then click inside the Entity to which the relationship ends. A line now connects the two entities.
2. Double-click the relationship line or right-click the relationship line and select Properties to display the properties window for the relationship.
3. Specify the roles of the relationship. The ERD shows the role as a label in the diagram beside the cardinality it describes. You can display "From" only, "To" only, both "From" and "To", or no labels.
4. Name the relationship. You can use a system specifying the connection (for example, CUSTOMER\_MAKES\_RESERVATION) or some other system. You may also have naming conventions specific to your company or site.
5. Save the ERD. Name the diagram RENTAL\_ERD.

While you use the ERD to place the entities and their relationships in a diagram, you use the Hierarchy window to add attributes to an Entity. Attributes must be defined for each Entity to forward engineer the ERD.

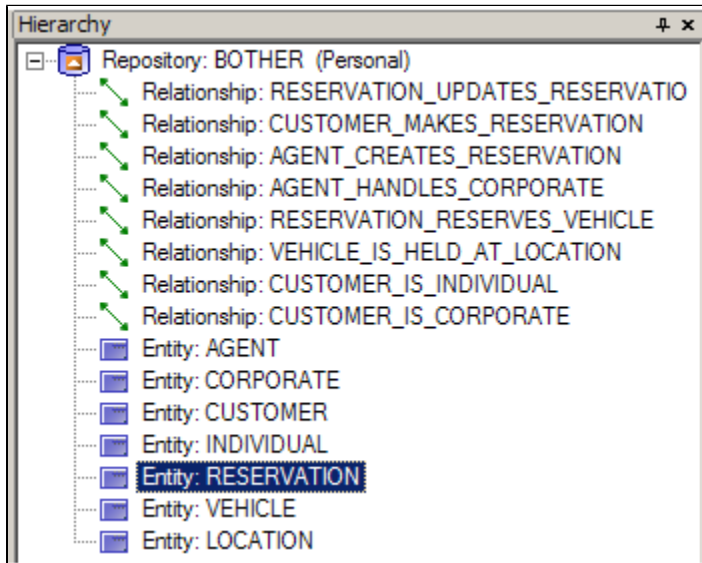
### Building the Attribute Hierarchies

To build the attribute hierarchy, do the following:

**With the ERD constructed above open, in Construction Workbench select Edit > Select All. Everything in the drawing is selected.**

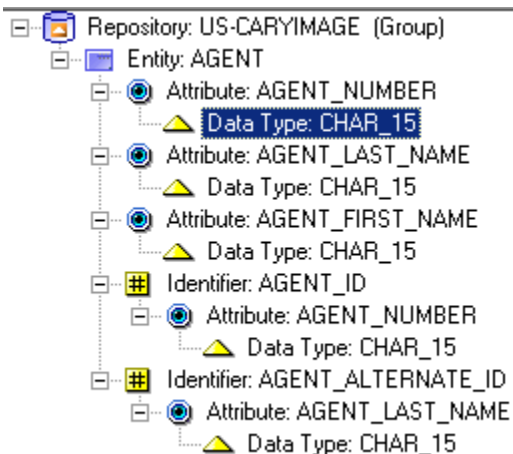
1. Select Edit > Open as Hierarchy. The entities and relationships are now available in a scoped version of the hierarchy.

## Entities and relationships in Hierarchy



1. Create the attributes, identifiers, and data types for the *AGENT* entity as in [AGENT entity hierarchy](#). For each Data type, set the format and length properties, as in [AGENT entity hierarchy](#).

### AGENT entity hierarchy




1. Right-click each *Data type* and select *Properties*. The *Data type Properties* dialog is displayed, as in [Properties dialog for the data type CHAR\\_15](#).
2. Select the correct *Data format* and *Data length* in the *General* section of the *Properties* dialog. For example, the data type entity named CHAR\_15 should have a data format of *Character* and a data length of 15, as shown in [Properties dialog for the data type CHAR\\_15](#).

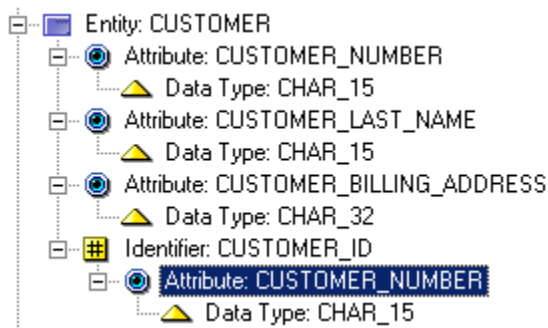
### Properties dialog for the data type CHAR\_15

Name	Value
[-] General (DATA_TYPE)	
Name	CHAR_15
Transformation Status	N/A
Preparation Status	Dirty
System ID	ZAADALE
Data Format	Character
Data Length	0015
Data Fraction	
[+] Audit	
[+] Remote Audit	
[-] Relationship (ATTRIBU...	
Relationship type	ATTRIBUTE_IS_TYPED_...
Parent name	AGENT_NUMBER
Child name	CHAR_15
Separator ID	0
Sequence number	10
[+] Relationship Audit	
[+] Relationship Remote A...	

1. Right-click the *AGENT\_ALTERNATE\_ID* identifier in the hierarchy and click *Properties* . The *Identifier Properties* dialog is displayed
2. Change the *Type* field to *Alternate* .
3. Create the hierarchies shown in [CUSTOMER Hierarchy](#) through [INDIVIDUAL Hierarchy](#) for the CUSTOMER entity and its subtypes, CORPORATE and INDIVIDUAL.

 The CORPORATE and INDIVIDUAL entities do not need identifiers - they inherit from the CUSTOMER entity.

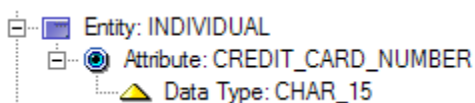
#### CUSTOMER Hierarchy



#### CORPORATE Hierarchy

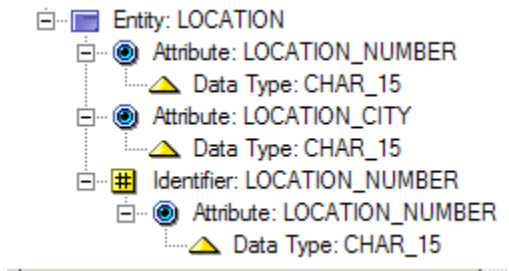


#### INDIVIDUAL Hierarchy



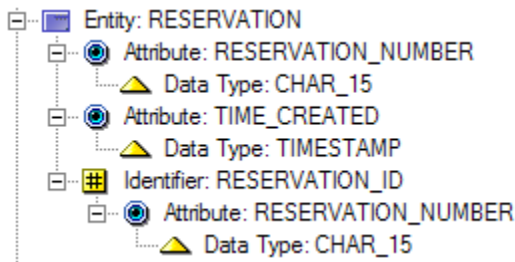
4. Create the hierarchy in [LOCATION hierarchy](#) for the LOCATION entity.

#### **LOCATION hierarchy**



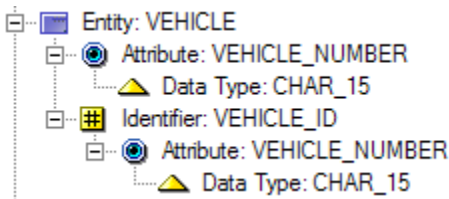
5. Create the hierarchy in [RESERVATION hierarchy](#) for the RESERVATION entity.

#### **RESERVATION hierarchy**



6. Create the hierarchy in [VEHICLE hierarchy](#) for the VEHICLE entity.

#### **VEHICLE hierarchy**



7. Select *File > Commit* from the Construction Workbench menu.

## **AppBuilder Engineering Tools**

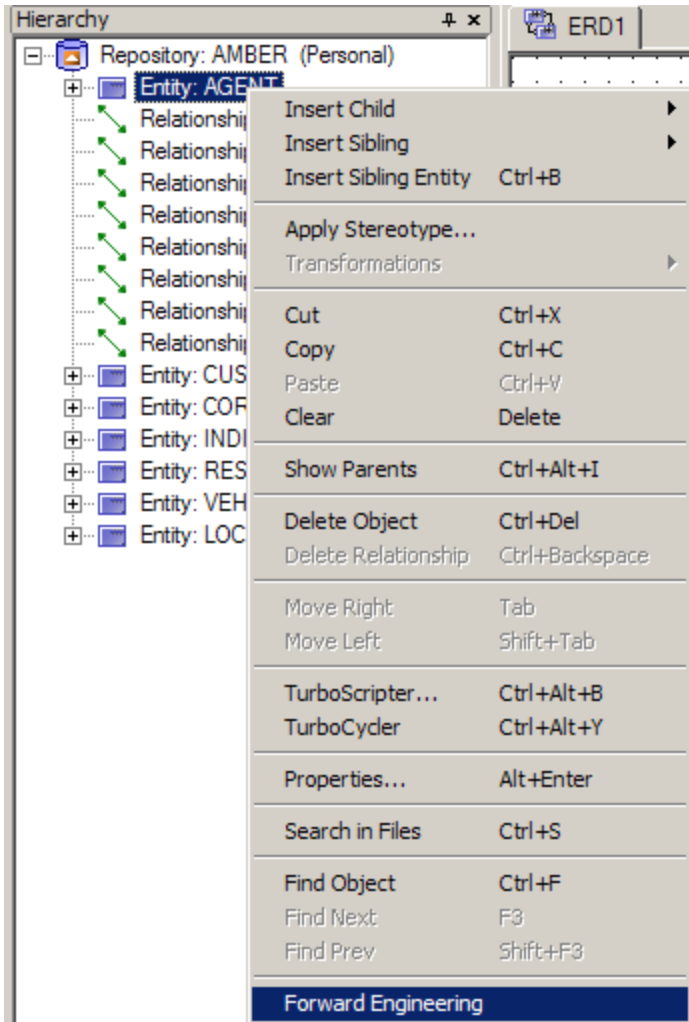
AppBuilder includes two engineering diagramming tools: the Entity Relationship Diagram (ERD) for forward engineering from a logical model to a relationship model and the Database Diagram for reverse engineering (in the opposite direction). In addition, AppBuilder provides tools for the transform process, trace analysis, and for the denormalizing process:

- [Forward Engineering an Entity Relationship Diagram](#)
- [Reverse Engineering with a Database Diagram](#)
- [Transforming a Relational Model](#)
- [Tools for Trace Analysis](#)
- [Denormalizing a Relational Model](#)

### **Forward Engineering an Entity Relationship Diagram**

The Entity Relationship Diagram is the AppBuilder source that you use to *forward engineer* your data model. Forward engineering the ERD translates logical entities, such as entities, relationships, identifiers, and attributes, into relational entities, which are tables, keys, and columns. Specifically, forward engineering creates a table for each ERD entity, keys from their identifiers and relationships, and columns from their attributes. In addition, forward engineering maintains inheritance information so that you can perform a trace analysis to see how the logical entities translated into relational entities. See [Tools for Trace Analysis](#) for information about trace analysis. It is possible to forward engineer directly one or more entities from the hierarchy window. Right-click on an entity to select Forward Engineering.

#### **Forward Engineering from the Hierarchy window**



While it is not possible to create relationships between entities via the hierarchy window, it is simple and direct to do so using an ERD. This section assumes that you have already created the ERD and the attribute hierarchy for the car rental agency example above.

**Make the Entity Relationship Diagram window (RENTAL\_ERD) active in the Construction Workbench by clicking on the window or selecting the window from the Construction Workbench Windows menu.**

1. Check each relationship in the diagram to ensure it has a cardinality symbol at each end. If any cardinality symbols are missing, place them as shown in [Sample ERD illustrating "from" and "to" entities](#).
2. Select *Edit > Show details* to show the hierarchy for each entity in the diagram. Check each entity to ensure it has the attribute structure as described in [Building the Attribute Hierarchies](#).



You can use the Analysis menu to check the ERD before moving to forward engineering. Commands include Check for Duplicates, Check for Unnamed Objects, and Verify.

#### **Output window results**



```

Verification Errors and Warnings....
Verifying AGENT
Verifying CORPORATE
Verifying CUSTOMER
Verifying CUSTOMER
Verifying INDIVIDUAL
Verifying RESERVATION
Verifying VEHICLE
Verifying LOCATION
Verifying LOCATION
No critical errors or warnings detected, Verification successful.

```

1. Select *Analysis > Forward engineer* . The system displays a message in the *Analysis* tab of the Output window showing the status of the process. If you encounter errors, repeat steps 1 - 3 of this example again.

**Forward Engineering Output window**

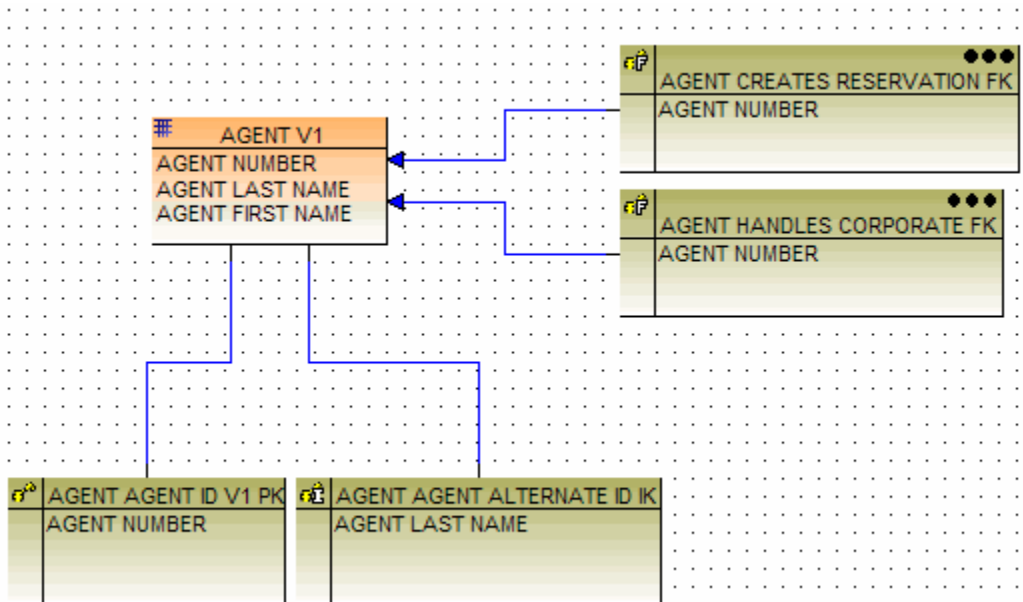
```

Forward Engineering Errors and Warnings....
Engineering AGENT
Engineering CORPORATE
Engineering CUSTOMER
Engineering INDIVIDUAL
Engineering RESERVATION
Engineering VEHICLE
Engineering LOCATION
Engineering RESERVATION_RESERVES_VEHICLE
Generating foreign key relationships.
Forward Engineer Successful.

```

1. When you have successfully created the tables from the Entity Relationship Diagram, create a new Database Diagram. From the Construction Workbench menu, select *File > New > Database Diagram* .
2. Insert a Table object from the toolbar.
3. Use the Query button to select one of the tables just created. (For example, Agent.) The selected table is inserted.
4. In the Database Diagram, you can display the tables and keys that forward engineering has created To expand the objects, use F8. This will show the entity and its keys (as in the individual database diagrams just below).

**Tables, columns, and keys for AGENT**

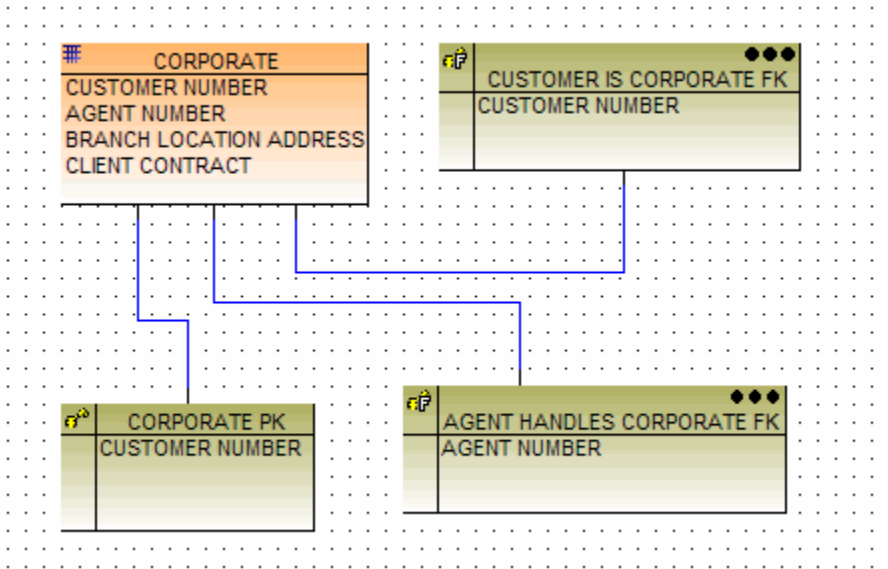




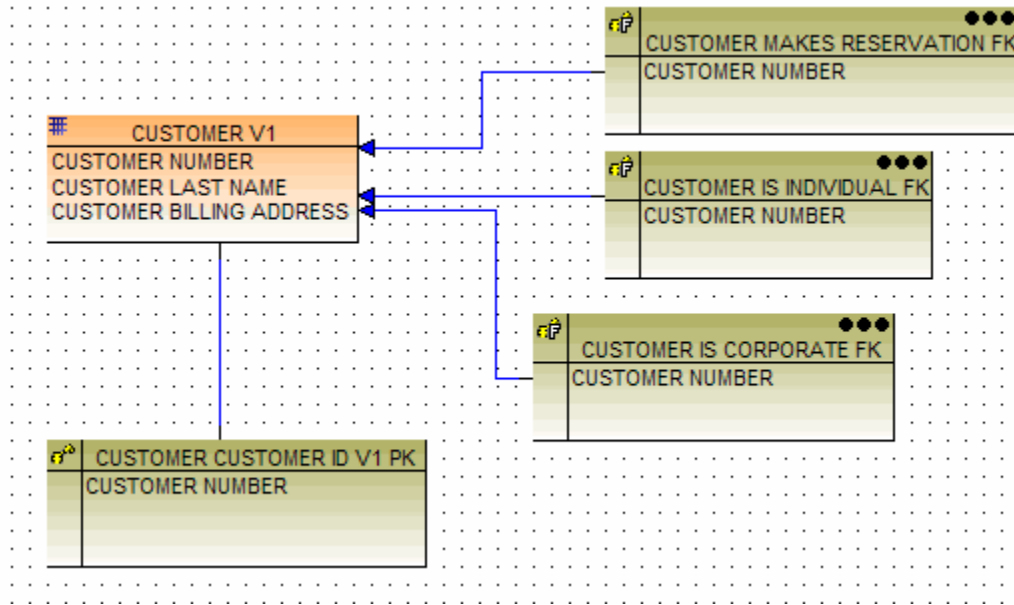
The three dots within the box for one object indicates that there are other objects tied to it. You can view other objects by selecting the object with the three dots and then used the command Edit > Explode.

7. Add each table to the Database Diagram, checking each time that the columns and keys are correct.

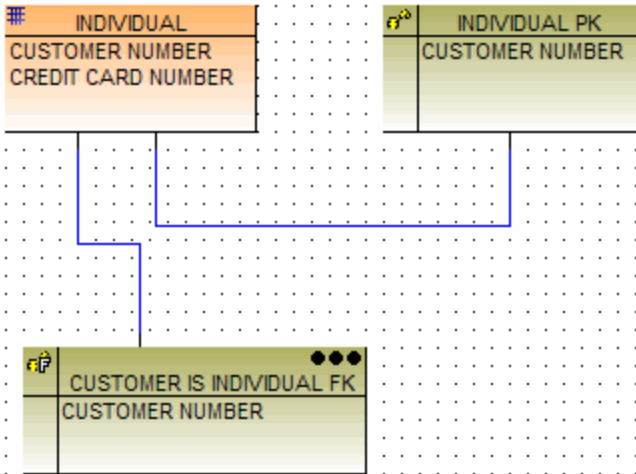
**Tables, columns, and keys for CORPORATE**



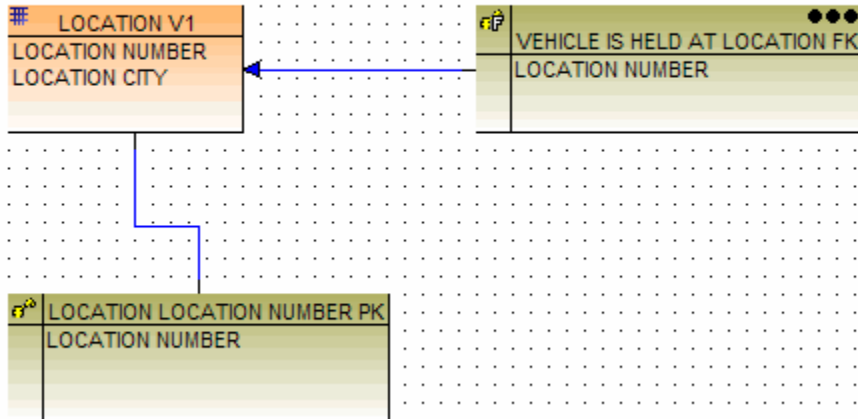
**Tables, columns, and keys for CUSTOMER**



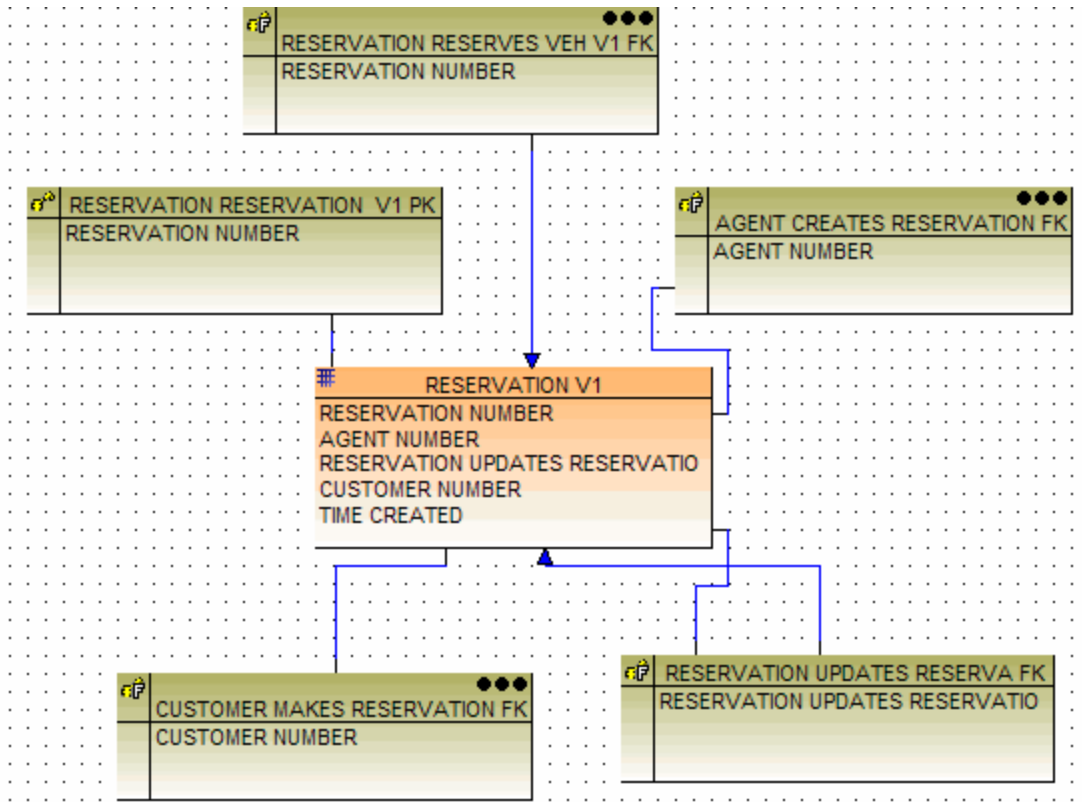
**Tables, columns, and keys for Individual**



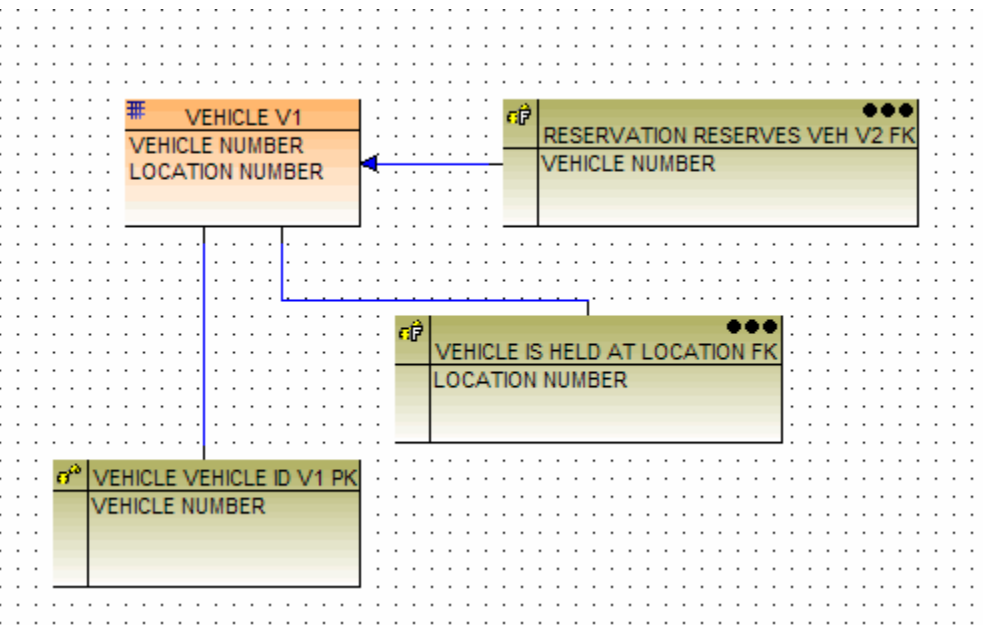
*Tables, columns, and keys for Location*



*Tables, columns, and keys for Reservation*



Tables, columns, and keys for Vehicle



1. Select *File > Commit* from the Construction Workbench menu.

### Reverse Engineering with a Database Diagram

You can also use a Database Diagram (DBD) for *reverse engineering*. The reverse engineering process converts the source Database Diagram back to an Entity Relationship Diagram (ERD). The relational entities of the DBD (tables, keys, and columns) are converted back to the logical entities of the ERD (entities, relationships, attributes, and identifiers). The resulting ERD reflects any changes you make to the relational model in a DBD, so the reverse-engineered ERD may not be the same as the forward-engineered ERD. Reverse Engineering resolves many complexities that arise when you forward engineer a logical model and then change the resulting database diagram. For example, if you forward engineer an ERD that contains a complex attribute, and then you change one of the corresponding columns in the DBD, simply changing the attribute in the ERD is not sufficient, because the attribute and the complex attribute may have other

dependencies. Depending on the dependencies, reverse engineering generates a new attribute. The verification process for reverse engineering ensures that the database model is correct and in sync. The verification process makes sure that: A table contains all columns contained in the owned keys. All foreign keys for the tables to be reverse engineered have one referred-to table. Each key is owned by only one table. Reverse engineering generates all traceability information necessary for you to forward and reverse engineer the same model as many times as you want.

## Transforming a Relational Model

The *transform* process in the Database Diagram (DBD) translates relational entities (tables, keys, and columns) into structural entities (files, views, and fields). The transformation process transforms a relational model represented in a DBD into data structures that AppBuilder rules use to read and write to a database.

By default, the transformation process uses the implementation names for the entities it converts. The advantage of using the implementation name is that the resulting fields will have the same names as the columns in the database tables. If you prefer, you can change the generation method to use long names instead of implementation names. From the Construction Workbench menu, select *Tools > Workbench Options > Engineering*, and from the File generation method drop down list, choose LONGNAME or IMPNAME (for implementation name).

The transformation process verifies the following:

No table has two columns with the same implementation name.

Every column in each table has an implementation name.

If transformation is by implementation name (the default), two columns with different formats do not have the same implementation name.

## Denormalizing a Relational Model

*Denormalization* of data is a designing concept where data is stored redundantly in a multi-dimensional model and provides fast response (but slow update) time.

The denormalizing process in the Database Diagram enables you to copy a column from one table to another through a foreign key.

Denormalization creates physical columns in a table through inheritance from other tables. Denormalizing can improve performance by reducing the number of table joins while executing an application.

## Tools for Trace Analysis

Trace analysis tools are used with both an ERD and a DBD. Traceability information in the repository enables you to:

- See the impact on the relational model if you change the logical model
- See the impact on the logical model if you change the relational model
- Forward and reverse engineer the same model as many times as you want

Use the trace analysis process in the Entity Relationship Diagram (ERD) to see how forward engineering translated logical entities into relational entities. Trace analysis tracks and reports how the logical entities in an ERD correspond to the relational entities in the database model. For example, after forward engineering, use the forward engineering report to answer questions such as:

What tables are affected when I change an attribute?

Which is the key created from an identifier?

An example of the trace analysis of an entity in the ERD is shown in [Trace analysis of an object in an ERD](#):

### Trace analysis of an object in an ERD

```
Implementation Details for Entity AGENT
This object creates table AGENT_V1.
Identifier AGENT_ID creates key AGENT_AGENT_ID_V1_PK.
Identifier AGENT_ALTERNATE_ID creates key AGENT_AGENT_ALTERNATE_ID_IK.
Identifier AGENT_ID is inherited through relationship AGENT_HANDLES_CORPORATE as
Identifier AGENT_ID is inherited through relationship AGENT_CREATES_RESERVATION a
Attribute AGENT_NUMBER creates column AGENT_NUMBERR of table AGENT_V1.
Attribute AGENT_LAST_NAME creates column AGENT_LAST_NAME of table AGENT_V1.
Attribute AGENT_FIRST_NAME creates column AGENT_FIRST_NAME of table AGENT_V1.
Attribute AGENT_NUMBER is inherited through relationship AGENT_HANDLES_CORPORATE
Attribute AGENT_NUMBER is inherited through relationship AGENT_CREATES_RESERVATIO
End of Report
```

The trace analysis process in the Database Diagram enables you to see the same information from the vantage point of relational objects. After reverse engineering, reverse trace analysis tracks and reports how the relational entities in the database model correspond to the logical entities in an ERD. You can then analyze the report to answer questions such as:

If I change a column in a table, what attributes or relationships are affected during reverse engineering, and how?

If I add a column to a key, which attributes or relationships are affected, and how?

An example of the results from reverse trace analysis is displayed in [Reverse trace analysis from a database diagram object](#):

### Reverse trace analysis from a database diagram object

```
Database Diagram Reverse Engineering Traceability Report
Drawing: VEHICLE
Report Generated: Saturday, 1/20/2007 6:47:36 PM
Implementation Details for Table VEHICLE_V1
This object is created by entity VEHICLE.
Primary key VEHICLE_VEHICLE_ID_V1_PK is created by identifier VEHICLE_ID of entit
Foreign key VEHICLE_IS_HELD_AT_LOCATION_FK is created by relationship VEHICLE_IS_
Column VEHICLE_NUMBER is created by attribute VEHICLE_NUMBER.
Column LOCATION_NUMBER is created by relationship VEHICLE_IS_HELD_AT_LOCATION fro
End of Report
```

## Diagramming the Application

When planning your application, you may want to first define and analyze the business objects that are associated with your application. The business objects are developed and modeled to present three different views of the business:

- A data view composed of an entity or set of related entities
- A control view composed of the data life cycle states
- Process views composed of the manual and automated processes

AppBuilder offers separate diagramming tools to guide you through the process of creating each of these models:

The Entity Relationship Diagram (ERD) and the Database Diagram (DBD) are used to model the data view. These tools are covered in [Data Modeling](#).

- The [State Transition Diagram](#) is used to model the control view encompassing the data life cycle states.
- The [Process Dependency Diagram](#) (PDD) is used to model the manual and automated processes used in the application.
- The [Window Flow Diagram](#) (WFD) is used to plan the flow of your application windows. The WFD is used at a later point than the other diagrams.

## State Transition Diagram

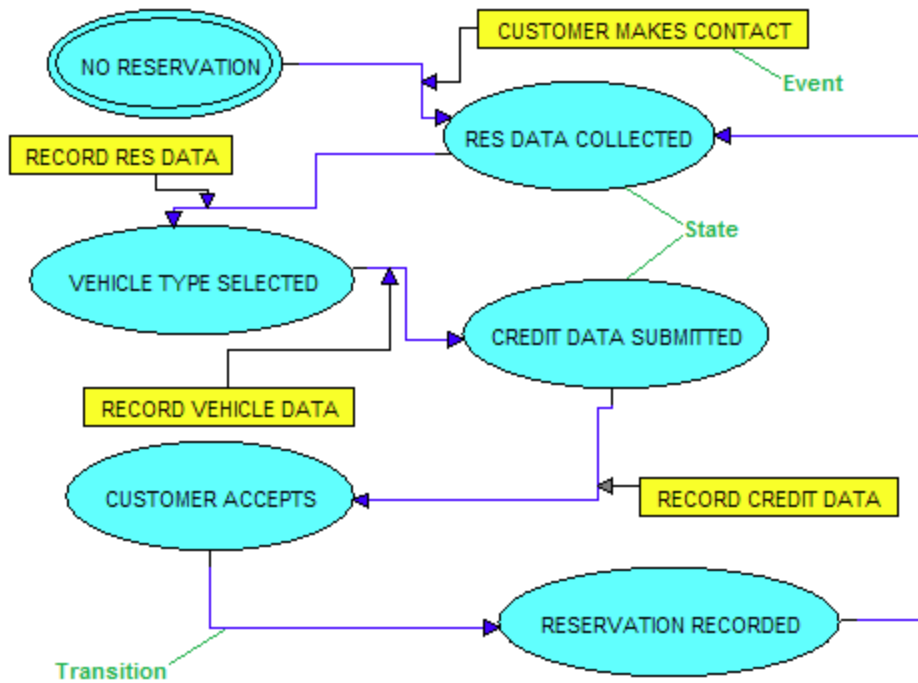
A State Transition Diagram identifies the internal and external events to which an application must respond and the various states of the resulting data. Transitions between states help outline the business rules an application needs.

For each business object defined, the processes that the business uses to manipulate the data must be identified. State Transition modeling helps identify these processes by defining the life cycle states a business object goes through in response to internal and external events and defines the control logic for the developing systems. A State Transition Diagram is a graphical representation of a business object's life cycle.

The following outlines the purpose of the State Transition Diagram:

- Describes the various life cycle states
- Depicts the transition paths between states
- Identifies the business events that affect the business object
- Indicates the control of events and associated processing

### ***State Transition Diagram***



## States

A State Transition Diagram is a graphical representation of a State model. A State model is a network of states, events, and state transitions. A State model contains three types of states:

- **Initial State** represents the business object's condition before any processing and transitions.
- **Intermediate State** is the state that an object passes through in response to an external or internal event. The business object resides in an intermediate state most of its life cycle.
- **Final State** is the last state of a business object's life cycle.

## Events

An event causes a business object to respond through a set of predefined processes. When the processing completes, the business object either remains in the same state waiting for another event to occur, or move to another state. The same event can appear on more than one transition in the same, or different State models.

There are three types of events:

- **External** – These events are driven by external agents. For example, a supplier requires the recall of a product because it is faulty or dangerous in some way.
- **Internal** – These are internal to and can be controlled by the organization within this business scope. For example, the organization rejects proposals to stock a new product.
- **Temporal** – These events are time based. For example, the product is defined as a seasonal line which means that after a certain date it is no longer kept in stock.

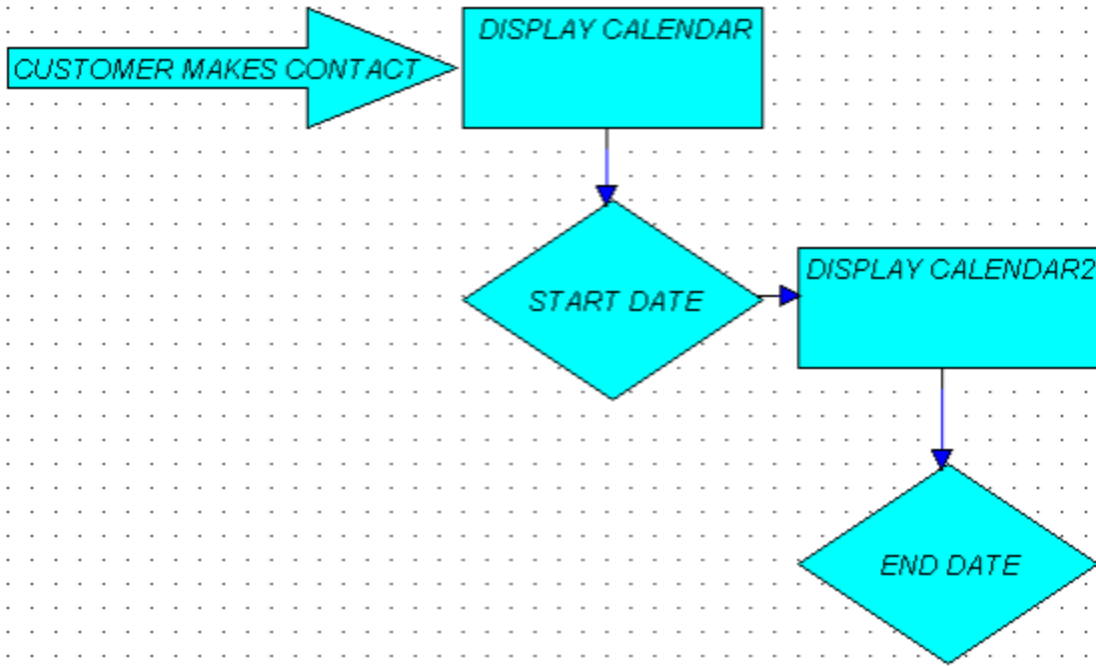
The events in the State model represent logical events to which the business must respond, rather than actions that handle the event. Actions that handle the event are the transactions. The event initiates the processing of the transaction. The business object might or might not change states as a result of the transaction.

## Process Dependency Diagram

In systems development, a process is one or more tasks that a system performs. Processes must often be performed in a certain order because the output of one process is the input for another. This is described as dependency. Use the Process Dependency Diagram (PDD) to specify the logical structure of the processes in an application. The procedure for developing a PDD is as follows:

1. Identify the processes and show how they relate to each other.
2. Specify when in the application each process is performed and what are the prerequisite tasks for that process.
3. Identify the data that will be used.

### *Process Dependency Diagram*



More information on both State Transition Diagrams and Process Dependency Diagrams and their constituent elements is found in the *Development Tools Reference Guide*.

PDDs can show three types of dependencies:

- **Sequential** – The processes execute sequentially, one after the other.
- **Parallel** – The process executes and then triggers the execution of multiple processes.
- **Selective** – The process executes and is followed by multiple other processes, of which only one will execute.

Additional facts about a PDD:

- A PDD can use any combination of sequence, parallel and selective in defining the processing dependency.
- The event is received by the State model, which causes a set of processes to be initiated and depicted on the PDD as an event pointing to a process from nowhere. The event is named the same as the State model.
- If the process within the PDD generates an event, that event is shown as an event trigger directed away from the process.
- Every event in a business object State Transition Diagram must have a corresponding PDD that defines the processes that must occur in response to it.
- It is common to find the same processes used in several PDDs, either within a single State model or across several State models. To verify that two processes are similar, check whether both processes do the following:
  - Carry out the same function
  - Read or write the same attributes from or to the same entities
  - Accept the same attributes from sources other than entities (that is event data or data produced by other processes)
  - Produce the same attributes as outputs
  - Produce the same events as outputs

If similar processes are used in exactly the same way, these are known as reusable processes and are classified into four types:

- **Accessor Process** – accesses data in a single object. This is the most reused type of process, both within the actions of a single State and across the actions of different State models.
- **Event Generator Process** – produce exactly one event as output. An event generator does not access any data. This type of process is likely to be reused in several PDDs.
- **Transformer Process** – compute or transform data from input data to output data. Transformers are typically not reusable outside the particular business object.
- **Tester Process** – tests a condition and makes a conditional control output. Testers are typically not reusable and are only initiated within the particular business object.

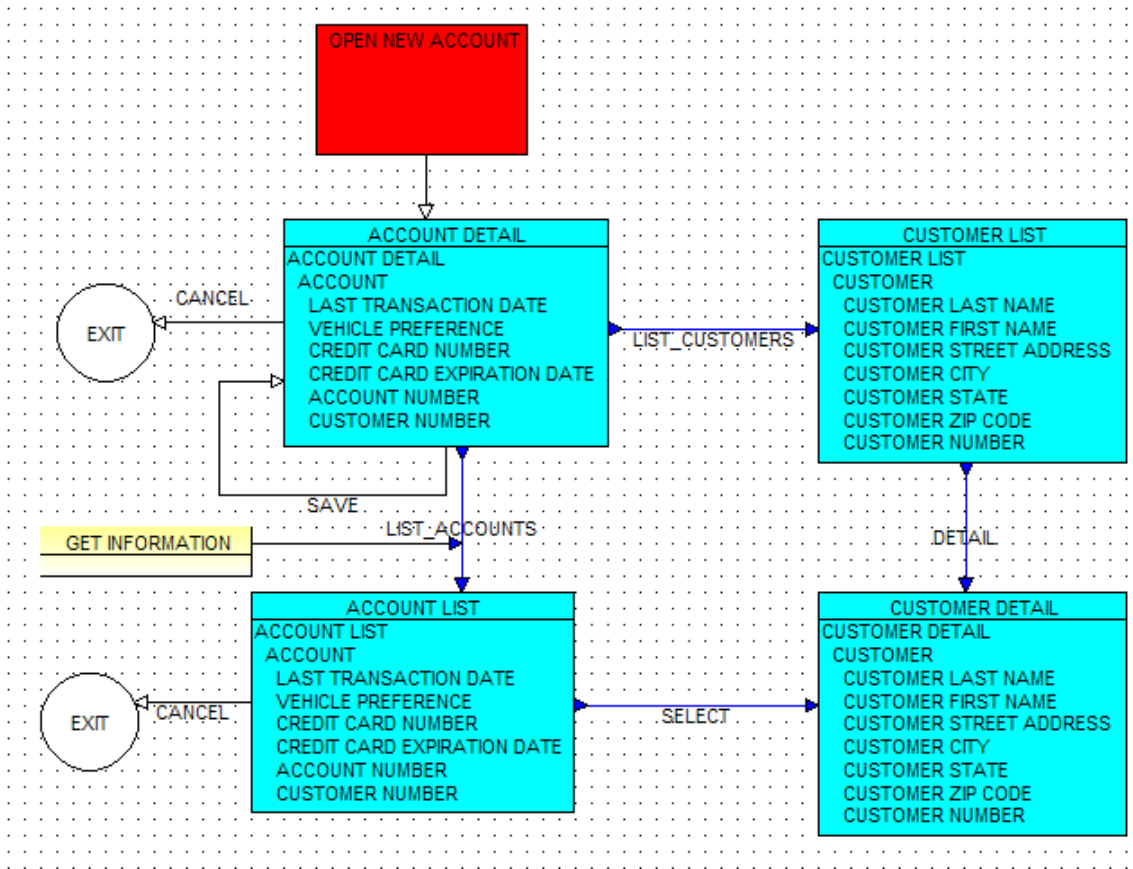
## Window Flow Diagram

A Window Flow Diagram is used to model window flow, user interaction, and event handling during the application design phase. With the Window Flow Diagram, you can show end users how their actions trigger navigation between these windows. The Window Flow Diagram enables you to do the following:



- Build a basic flow model to describe the interaction between windows
- Test the model in simulation mode for user approval
- Determine how to handle events

**Window Flow Diagram example**



**Providing Input for the Window Flow Diagram**

Before you begin using the Window Flow Diagram, you may well have completed several other models in the process of analyzing and modeling your business application, such as:

- a logical model – consisting of entities, identifiers, and attributes – in an entity relationship diagram (ERD). Refer to [Entity Relationship Diagram \(ERD\)](#) for more information about the ERD.

✔ You can take the logical model in an ERD and, using forward engineering, create a Database Diagram (DBD) and the associated tables, columns, and keys. See [Forward Engineering an Entity Relationship Diagram](#). In addition, you can use the ERD with TurboCycler to generate automatically rules, windows, and fields for building the application. With the ERD in the Construction Workbench workspace, select Analysis > TurboCycler. A full example is provided in the *Scripting Tools Reference Guide*.

- a relational model in a database diagram (DBD), produced by forward engineering the ERD's logical entities into relational entities – tables, columns, and key. Refer to [Database Diagram \(DBD\)](#) for more information about the DBD.
- a structural model produced by transforming the DBD, which converts the table, keys, and columns into AppBuilder data structures – files, views, and fields. Refer to [Transforming a Relational Model](#) for more information about creating a structural model.
- a set of State Transition Diagrams. Refer to [State Transition Diagram](#) for more information.
- a set of Process Dependency Diagrams (PDDs). Refer to [Process Dependency Diagram](#) for more information.

If you have built these models and then automatically generated rules, views, windows, and other objects using TurboCycler, you already have windows within a hierarchy. You can modify the hierarchy and rules as you wish to specify the window flow you desire. You can base a Window Flow Diagram on the window hierarchies you have already created. You can also use the Window Flow Diagram to generate the windows without having generated objects automatically.

**Using the Window Flow Diagram**

Developing an application using the Window Flow Diagram consists of a series of tasks in three phases. Building a user interface is an iterative process; therefore, it might be necessary to repeat some of these steps until you are completely satisfied with the results.

- [Phase 1 - Create the Flow Model and Simulate the Prototype](#)
- [Phase 2 - Attach Rules to Handle Events and Decision Logic](#)
- [Phase 3 - Generate and Run a Working Prototype](#)

### Phase 1 - Create the Flow Model and Simulate the Prototype

Phase 1 encompasses 9 of the 12 steps involved in building a user interface and running a prototype. The decisions resulting from the first six steps produce the underlying hierarchies and how each window looks.

- [Step 1. Create or Identify an Initial Process](#)
- [Step 2. Create the Primary Window](#)
- [Step 3. Create the Hierarchy of the Primary Window](#)
- [Step 4. Create the Secondary Windows](#)
- [Step 5. Create the Hierarchies of the Secondary Windows](#)
- [Step 6. Paint the Windows](#)

The remaining steps in Phase 1 complete the flow model and simulate the window interaction, preparing it for the test phase.

- [Step 7. Identify and Create the Flow Pattern](#)
- [Step 8. Create an Exit Terminal](#)
- [Step 9. Simulate Window Flow](#)

Common simulation errors are discussed in the section [Possible Simulation Errors](#).

#### Step 1. Create or Identify an Initial Process

A process is the starting point for a window flow model – the anchor to which all application structures and code attach. The first step in producing a window flow model is to identify or create the physical process for which the window flow is needed.

The Window Flow Diagram tool displays a process as a rectangle. Although the physical process in Window Flow Diagram is not the same as the logical process in the Process Dependency Diagram, they might have the same name when modeling the business system.

#### Step 2. Create the Primary Window

A thread is a logical sequence of windows that the user of an application can open. The first window in a thread is the *primary* window, and the additional windows in a thread are *secondary* windows.

A primary window is the starting point of the user's activity. The primary window presents the application's objects and actions to the user, and is usually defined by a data object. It should contain all the information relevant to the object being viewed, or a set of menu choices leading to more detailed secondary windows. When the user opens the primary window, all other open windows close.

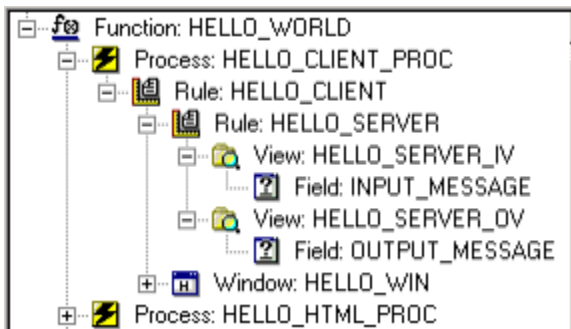
#### Step 3. Create the Hierarchy of the Primary Window

The hierarchy of view and field entities of the primary window defines the information that you want to present in that window, and how the information is presented in the final, painted window.

If you are developing your application based on a data model to change the views or fields, change the information in the ERD or the hierarchy window and forward engineer it again. Then, use the transformation process again on the resulting database diagram to produce a structural model of files, views, and fields. Refer to [Forward Engineering an Entity Relationship Diagram](#) and [Transforming a Relational Model](#) for more information.

If you are only using Construction Workbench objects, you can make your changes directly in the Construction Workbench hierarchy.

#### Workbench hierarchy



Each window's hierarchy must be able to share needed information with any window connected to it by a *flow*. You usually create common views and fields in the hierarchies associated with each window. Another way is to write rules to pass information from one window to another. In addition, each window's hierarchy must have a view for each entity whose information is available in that window. The verification process, simulation process, and generation process will all produce errors if you do not include all necessary views in the hierarchy of the window.

#### Step 4. Create the Secondary Windows

Secondary windows have all of the features of primary windows, but are displayed as children of the primary window – the result of a user action in a primary window or another secondary window, instead of from a process. There are several options for the flow of the primary window and the secondary windows.

##### Window relationships

Window flow	Description
Normal flow	The original window disappears before the second window appears. The first window re-appears when the second window is closed.
Nested flow	The original window remains on the screen and is disabled while the second window appears in front of it. The user cannot return to the original window until the second window is closed.
Detached flow	The second window appears on the screen. The user can use either window at any time. Note: If you're going to choose this style of window flow then you'll need to understand the concept of event and data posting. See <a href="#">Creating Event-Driven and Converse Applications</a> in this manual and the <i>ObjectSpeak Reference Guide</i> .

The child status is usually, but not always, indicated by nesting the secondary window. For example, users might need three sets of information when they create a new account because they might want to do any of the following:

- Duplicate the information from another account if this is the second account for a customer
- Look at a list of customers to select the correct customer number
- Change customer information, such as adding a new account or customer number

These information sets are placed in a secondary window because it is too much information to fit in the primary window or the information is not needed often.

#### Step 5. Create the Hierarchies of the Secondary Windows

The supporting hierarchy for each secondary window shares the basic format of primary window hierarchies. Follow the same steps as in [Step 3. Create the Hierarchy of the Primary Window](#) to create a hierarchy for each secondary window.

#### Step 6. Paint the Windows

Use Construction Workbench's Window Painter to design each window. In this step, decide the activation conditions – the pushbuttons and menu choices that open or close windows and pass information. In the Window Flow Diagram (WFD), these activation conditions determine when flows and rules are used.

As you paint each window (create buttons or menu choices that enables the user to open other windows and close the active window), the HPSID of each activation condition must match the event label on the flow line representing the transition between that window and the previous window.

#### Step 7. Identify and Create the Flow Pattern

The flow line leading to the window or away from it indicates when and how a user's actions open or close a window. The event label on the flow indicates the activation condition that triggers the flow, based on the HPSID of the activation condition push button or menu choice in the painted window.

Normal flow lines indicate a one-way path. In contrast, nested flows indicate a two-way path because closing a nested window returns control to the parent window. Normal flows end in white arrowheads, while nested flows start and end with blue arrowheads.

#### Step 8. Create an Exit Terminal

Exit terminals indicate closure of the window. A flow from a non-nested window to any exit terminal triggers the termination of the application. A flow from a nested window to an exit terminal closes the nested window and returns control to the parent window. You can use a single exit terminal to close windows or you can use individual exit terminals for each window.

#### Step 9. Simulate Window Flow

After verifying the diagram, you can simulate the flow of windows in the application. Follow the steps below:

1. Select a window to start the simulation, such as the primary window.
2. From the Construction Workbench menu, click **Analysis > Simulate Window Flow**.

The windows look and act as they would in the application. If an activation condition does not function correctly, check the event label and HPSID of each flow.



In a Window Flow Diagram, if you modify the text for labels from an HPSID, the simulation will not work. The label text must be changed back to an HPSID for a simulation to work.

## Possible Simulation Errors

The following table shows common simulation errors:

### Simulation errors

Simulation Error	Description
<b>No process defined in diagram</b>	Every Window Flow Diagram (WFD) must begin with an initial process, which should be connected to the primary window in the window thread. See <a href="#">Step 1. Create or Identify an Initial Process</a> for more information.
<b>Window &lt;window name&gt; has no data</b>	Each window in a WFD must have an underlying window hierarchy. See <a href="#">Step 3. Create the Hierarchy of the Primary Window</a> for more information.
<b>Window &lt;window name&gt; has no panel</b>	Each window in a WFD must have a painted panel to be successfully simulated or generated. See <a href="#">Step 6. Paint the Windows</a> for more information.
<b>Process &lt;process name&gt; has no main window</b>	Each WFD should begin with an initial process, which is connected to the primary window of the thread with a normal flow. See <a href="#">Step 2. Create the Primary Window</a> for more information.
<b>Flow has no label</b>	You must label a flow to indicate the activation condition that triggers it. The labels on the flow must match the HPS IDs of the objects that activate the flows. See <a href="#">Step 7. Identify and Create the Flow Pattern</a> for more information.
<b>Window &lt;window name&gt; and window &lt;window name&gt; do not share data</b>	Windows that pass data to each other must share the underlying hierarchy elements for the information that they want to pass. See <a href="#">Step 3. Create the Hierarchy of the Primary Window</a> and <a href="#">Step 5. Create the Hierarchies of the Secondary Windows</a> for more information.

## Phase 2 - Attach Rules to Handle Events and Decision Logic

Defining actions to be taken when a given event or set of events occurs is called event handling. Define event handling in window flow diagrams by attaching rules to flows between windows, or from a window to itself. User rules are the entry points or "hooks" by which you can add business specific logic and other code to the application. These rules handle data transformations (including database queries and calculations) and logic for customizing window displays (such as updating status areas).

The second phase entails adding the data transfer rules and the decision logic that determines how users interact with the windows of the application, and how they respond to the user's actions. Attaching rules to handle events and decision logic consists of the following two steps:

- [Step 10. Add Decision Logic](#)
- [Step 11. Build the Application](#)

### Step 10. Add Decision Logic

Use the decision object to add conditional logic to a Window Flow Diagram (WFD) to show the various paths available depending on user input to a window.

A diamond with a question mark (?) inside represents decisions in the WFD.

You can direct one or more flows from the decision to other windows in the diagram, including the source window. These flows leaving the decision can be normal or nested, and can have user rules attached. Their event labels should be the user-defined condition codes set by the input flow to the decision. Name these codes so that other developers can easily understand them.

### Step 11. Build the Application

Use the Rule Painter to create the application logic. Or, if you are building the application based on artifacts created in other diagramming tools, use the Scripting Tools.

- For information about the Scripting Tools, refer to the *Scripting Tools Reference Guide*.
- For information about the Rule Painter, refer to the *Development Tools Reference Guide*.
- For information about Rules code, refer to the *Rules Language Reference Guide*.

## Phase 3 - Generate and Run a Working Prototype

### Step 12. Prepare and Run the Application

When all of the elements of the application are complete in the Construction Workbench, prepare them for execution. Follow the steps below:

1. Right-click the Function at the top of the application hierarchy.

2. Select **Super Prepare**. This prepares all of the necessary objects for execution.

## Window Flow Diagram Tutorial

The AppBuilder Window Flow Diagram tool enables you to plan the flow of your windows. This process then helps in the construction of the windows in Window Painter. In addition, you can use the Window Flow Diagram tool to plan how to handle events. Ultimately, you can use the WFD to produce and test a simulation of the window flow.

This tutorial covers the following tasks:

- [Creating a Window Flow Diagram](#)
- [Linking to Another Diagram](#)
- [Building the Second Diagram](#)
- [Simulating the Window Flow](#)

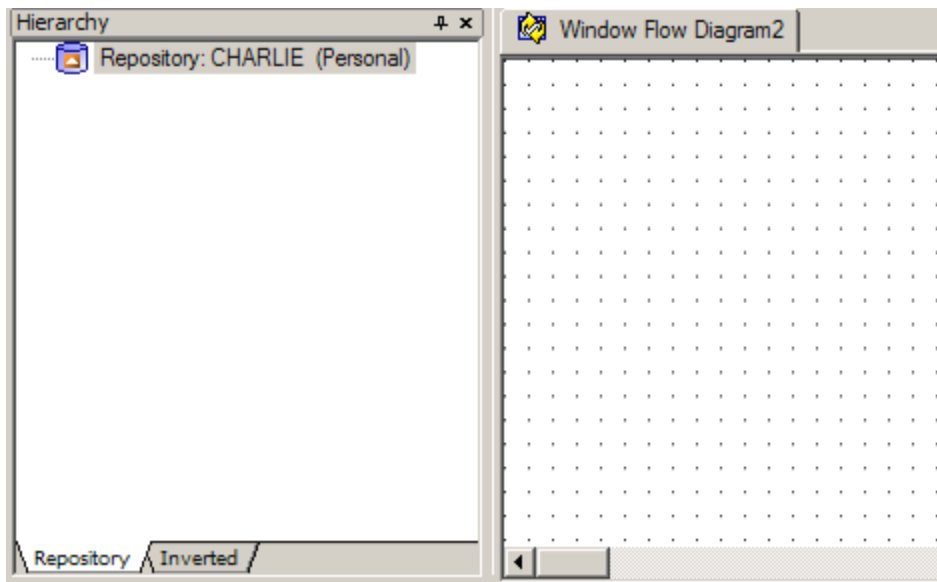
### Creating a Window Flow Diagram

To design your window flow diagram, you must have a repository set up and the AppBuilder Construction Workbench open. See the *Repository Administration Guide for Workgroup and Personal Repositories* for information on the repository and *Development Tools Reference Guide* for information on the Workbench.

1. In the Construction Workbench, select **File > New**. The Create New dialog box is displayed.
2. Select **Window Flow Diagram** from the list and click **OK**. A blank WFD window is displayed in the Work Area.

Your Construction Workbench should now look similar to this:

#### Sample new WFD



### Placing Objects in the Diagram

The following objects are available in the Window Flow Diagram toolbar:

#### WFD objects



1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	----	----

#### Window Flow Diagram toolbar objects

1. Process	7. Rule
2. Window	8. Flow

3. Terminal	9. Nested Flow
4. Decision	10. Detached Flow
5. Entry Point	11. Note
6. Dialog Unit	

For this tutorial, you will use the icons in the toolbar to add objects to your diagram. However, you can also use the **Insert** menu to add the objects.

### Adding the Objects

The first drawing for this Window Flow Diagram shows the flow of a process, 2 windows, a dialog unit and a terminal. It demonstrates how to attach flows and then to simulate or preview the application.

In AppBuilder, a process is an object type that describes business activities that comprise a logical unit of work. Each process represents a single application (leaf process), or a set of applications. Windows define the interactive user interface to an application. A dialog unit represents a way of executing a set of windows that performs a specific functionality. A dialog unit lets you modularize an application so that you can reuse the same windows and flows in different drawings. A terminal provides an exit from the application or process.

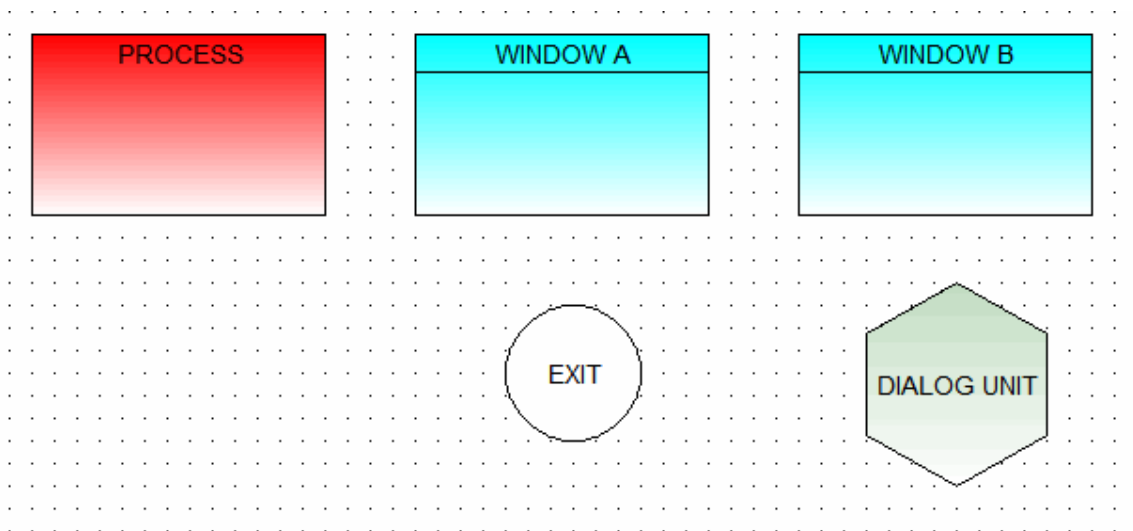
1. Click the **Process** icon in the toolbar.
2. Click inside the WFD work area where you want to place the object. Until you define the object, it is displayed as a gray box.
3. Double-click the object. The Insert PROCESS dialog is displayed.
4. Type PROCESS in the Name field and click **Insert**. The Insert PROCESS dialog is closed and the focus is returned to the Construction Workbench. The Process object in the WFD is no longer gray.
5. Repeat steps 1-4 using the **Window** and **Dialog Unit** icons instead of the **Process** icon. Use the table below for object types and names.

#### WFD objects

Object Type	Text to enter in Name field
Window	WINDOW_A
Window	WINDOW_B
Dialog unit	DIALOG_UNIT

6. Click the **Terminal** icon in the toolbar.
7. Click inside the WFD where you want to place the object. The terminal is displayed as a circle labeled "EXIT". No definition is necessary for the terminal.
8. Arrange the objects in your WFD to look similar to the following. You can click and drag objects, using the grid for alignment.

#### WFD in process



9. Select **File > Commit** or click the **Commit** icon. The Save drawing dialog is displayed.
10. Type a name for your drawing in the Drawing name field and click **Save**.



Save your file to the Repository frequently during the tutorial.

### Building a Hierarchy

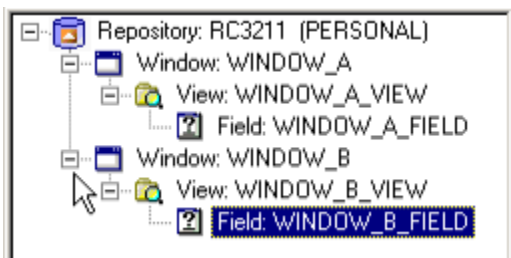
Each window in a Window Flow Diagram must have an underlying hierarchy. Hierarchies show the structure that defines the relationships of objects within an application.

To generate the hierarchy for this sample Window Flow Diagram, follow these steps:

1. Press the Shift key while you click both windows.
2. With both windows selected, right-click and select **Open as Hierarchy**. The window names are displayed in the Hierarchy Window.
3. Right-click **WINDOW\_A** and select **Insert Child > View**. The Insert VIEW dialog is displayed.
4. Type WINDOW\_A\_VIEW in the Name field and click **Insert**. A view is added to the hierarchy under WINDOW\_A.
5. Right-click the view and select **Insert Child > Field**. The Insert FIELD dialog is displayed.
6. Type WINDOW\_A\_FIELD in the Name field and click **Insert**. A field is added to the hierarchy under the view.
7. Repeat steps 3-6 to create a view and field for **WINDOW\_B**. Name these WINDOW\_B\_VIEW and WINDOW\_B\_FIELD.

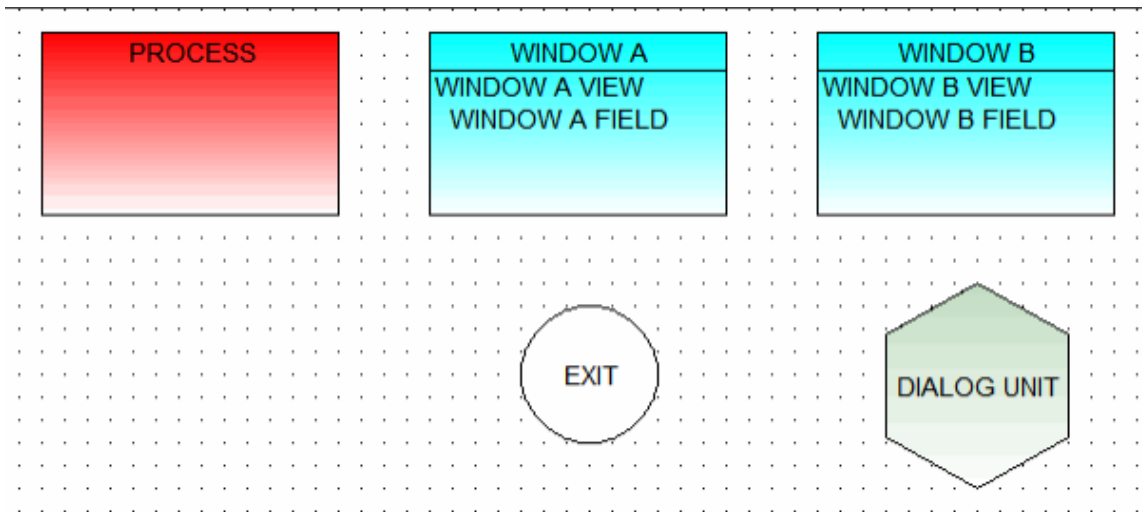
Your Hierarchy Window should look similar to this:

### Hierarchy Window



In addition, the fields and views that have been added via the hierarchy have been added to the objects in the Window Flow Diagram.

### Window Flow Diagram with fields and views



### Designing the Windows in Window Painter

1. In the Construction Workbench Hierarchy Window, double-click **WINDOW\_A**. Optionally, you can right-click the window name and select **Open Window**.  
The window and its Properties dialog box are displayed in the Window Painter.
2. In the Window Painter toolbar, click the **Push Button** icon.
3. In the window, click where you want the push button to be.  
The push button is created.
4. Drag the corner dots to adjust the size of the push button. Optionally, change the height and width in the push button's Properties dialog box.

✔ Double-click an object to display its Properties dialog box.

5. With the push button selected, enter the following text in the appropriate field to define the push button's properties:

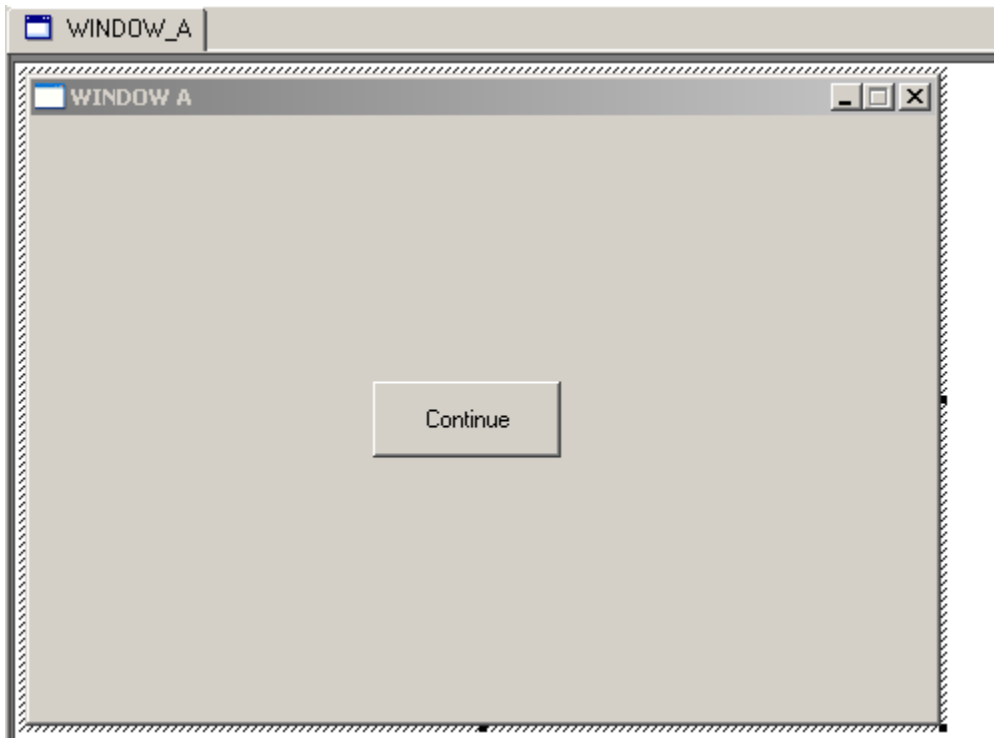
**WINDOW\_A push button properties**

Field	Text to Enter
Text Field	&CONTINUE
HPSID field	CONTINUE

✔ You may want to name the window to more easily follow the window flow simulation process. To do so, in the window's properties dialog enter the window's name in the Text field.

WINDOW\_A should now look similar to this:

**Sample window and push button**



6. Repeat steps 1-4 to add 2 push buttons to WINDOW\_B.

7. Select each push button individually and enter the following text in the appropriate field to define the push buttons' properties:

**WINDOW\_B push button properties**

	Appropriate Field	Text to Enter
Push Button 1	Text Field	&EXIT
	HPSID Field	EXIT
Push Button 2	Text Field	&NEXT DRAWING
	HPSID Field	NEXT DRAWING





Add text to objects as needed using the Static Text icon.

### Adding Flows and Events

The flow is represented by a line indicating the relationship among windows. An “event label” denotes the action that triggers a flow and is based on the HPSID (system identifier) assigned to the push button or menu choice.

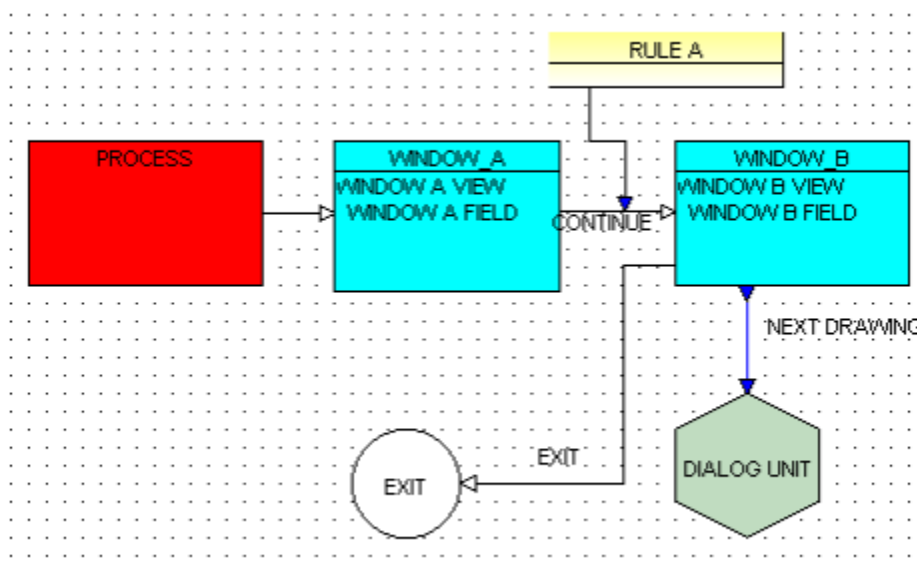
1. Select **Window <name of diagram>** to return to the WFD, where **<name of diagram>** is the name you chose when you saved your diagram in [Adding the Objects](#).
2. Click the **Flow** icon in the WFD toolbar.
3. Click **PROCESS** and then **WINDOW\_A**. An arrow is displayed connecting the two.
4. Repeat steps 2 and 3 to place a flow line from **WINDOW\_A** to **WINDOW\_B** and from **WINDOW\_B** to **EXIT**.
5. Click the **Nested Flow** icon in the WFD toolbar.
6. Click **WINDOW\_B** and then **DIALOG UNIT**.  
A nested arrow is displayed between the 2.
7. Double-click the flow line from **WINDOW\_A** to **WINDOW\_B**.  
The Select Action dialog box is displayed.
8. Select **CONTINUE** and click **OK**.
9. Double-click the flow line from **WINDOW\_B** to **EXIT**.  
The Select Action dialog box is displayed.
10. Select **EXIT** and click **OK**.
11. Double-click the flow line from **WINDOW\_B** to **DIALOG UNIT**.  
The Select Action dialog box is displayed.
12. Select **NEXT DRAWING** and click **OK**.

### Adding a Rule

The rule object in the Window Flow Diagram marks the execution of a rule along with a flow. In this case, the rule is a graphical representation only and does not affect the simulation. Only one rule can be associated with a flow.

1. Click the **Rule** icon.
2. Click in the Workbench Work Area just above the flow line between **WINDOW A** and **WINDOW B**.  
The rule box and its line are displayed.
3. Click immediately on the **CONTINUE** flow line to connect the rule line.  
A blue arrow connects the rule to the **CONTINUE** flow line.
4. Double-click the rule box.  
The Insert Rule dialog box is displayed.
5. Type **RULE\_A** in the Name field and click **Insert**.  
Your drawing should look similar to this:

**Diagram with rule object and line**



### Linking to Another Diagram

In the second drawing, you will diagram the flow from the dialog unit to another diagram. You can use an existing diagram, or you can build a new one. For this tutorial, you will build a new one. See [Building the Second Diagram](#). The Dialog Unit object connects one WFD to another and thus enables the reuse of windows and flows in different drawings. In the simulation, the dialog unit loads the drawing associated with it and starts that drawing's simulation. See [Simulating the Window Flow](#). An entry point marks the transition from the Dialog Unit to the next drawing.

1. In the first drawing, right-click the DIALOG UNIT and select **Navigate**. The Insert Drawing dialog box is displayed.
2. In the Name field, type a name for the second drawing. Click **Insert**. A new drawing is displayed, with a yellow arrow serving as an entry point.

## Building the Second Diagram

### Adding Objects

In this section, you will add the following objects to the second drawing: a decision point, 3 types of windows, and a terminal.

1. Click the **Decision** icon in the toolbar.
2. Click inside the WFD Work Area where you want to place the object. No definition is necessary for the decision object.
3. Click the **Window** icon in the toolbar.
4. Click inside the WFD Work Area where you want to place the object. Until you define the window, it is displayed as a gray box.
5. Double-click the window. The **Insert Window** dialog is displayed.
6. Type WINDOW\_C in the Name field and click **Insert**. The Insert dialog box is closed and the focus is returned to the Construction Workbench. The window is no longer gray.
7. Repeat steps 3-6 to add 2 more windows to the diagram. In Step 8, name the windows WINDOW\_D and WINDOW\_E.
8. Click the **Terminal** icon in the toolbar.
9. Click inside the WFD where you want to place the object. The terminal is displayed as a circle labeled "EXIT". No definition is necessary for the terminal.

### Adding to the Hierarchy

1. Press the Shift key while you click all 3 windows.
2. With all 3 windows selected, right-click and choose **Open as Hierarchy**. The window names are displayed in the Hierarchy Window.
3. Right-click **WINDOW\_C** and select **Insert Child > View**. The Insert VIEW dialog is displayed.
4. Type WINDOW\_C\_VIEW in the Name field and click **Insert**. A view is added to the hierarchy under WINDOW\_C.
5. Right-click the view and select **Insert Child > Field**. The Insert FIELD dialog is displayed.
6. Type WINDOW\_C\_FIELD in the Name field and click **Insert**. A field is added to the hierarchy under the view.
7. Repeat steps 3-6 to add a view and field for both WINDOW D and WINDOW E. Name these WINDOW\_D\_VIEW and WINDOW\_D\_FIELD, WINDOW\_E\_VIEW and WINDOW\_E\_FIELD, based on the name of the parent window.

### Using the Window Painter

1. In the Hierarchy Window, double-click **WINDOW\_C**. Optionally, you can right-click the window name and select **Open Window**. The window opens in the Window Painter.
2. Using the Window Painter toolbar, click the **Push Button** icon.
3. Place the cursor in the window and click where you want the push button to be. The push button is created.
4. Repeat steps 2 and 3 to add a second and third push button to this window.
5. Drag the corner dots to adjust the size of the push buttons. Optionally, change the height and width in the Properties dialog boxes for the push buttons.



Double-click an object to display its Properties dialog box.

6. Select each push button individually, and define the properties in the Properties dialog box as indicated in the table below:

#### WINDOW\_C push button properties

	Field	Text to Enter
<b>Push Button 1</b>	Text Field	&NESTED FLOW
	HPSID Field	NEST

<b>Push Button 2</b>	Text Field	&DETACHED FLOW
	HPSID Field	DETACH
<b>Push Button 3</b>	Text Field	&EXIT
	HPSID Field	EXIT

7. Repeat steps 1-3 and 5-6 to add one push button each to WINDOW\_D and WINDOW\_E. Use the tables below to define the properties in the appropriate Properties dialog box:

**WINDOW\_D properties**

Field	Text to Enter
Text Field	&BACK
HPSID Field	BACK

**WINDOW\_E properties**

Field	Text to Enter
Text Field	&BACK
HPSID Field	BACK

**Adding More Flows and Events**

1. Select **Window > <name of diagram>** to return to the WFD, where **<name of diagram>** is the name you chose when you inserted a new drawing in [Linking to Another Diagram](#).
2. Click the **Flow** icon in the WFD toolbar.
3. Click the DIALOG UNIT arrow and then the **Decision** object. An arrow is displayed connecting the two.
4. Repeat steps 2 and 3 to display the arrows between objects as indicated in the table below:

**Flow connections to objects**

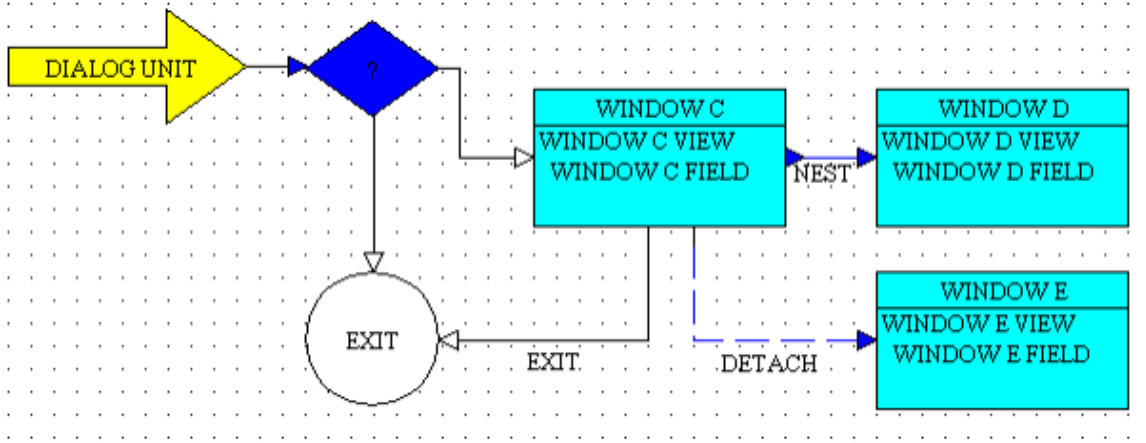
Type of Flow to Insert	Objects to Connect	
	From	To
Flow	Decision object	WINDOW C
Flow	Decision object	Terminal object
Nested	WINDOW C	WINDOW D
Detached	WINDOW C	WINDOW E
Flow	WINDOW C	Terminal object

5. Double-click the nested arrow. The Select Action dialog box is displayed.
6. Select **NEST** and click **OK**.
7. Repeat steps 5 and 6 to select actions for the other 2 flows. Use the chart below:

Flow	Action to Select
from WINDOW C to WINDOW E	DETACH
from WINDOW C to terminal object	EXIT

Your diagram should look similar to the following:

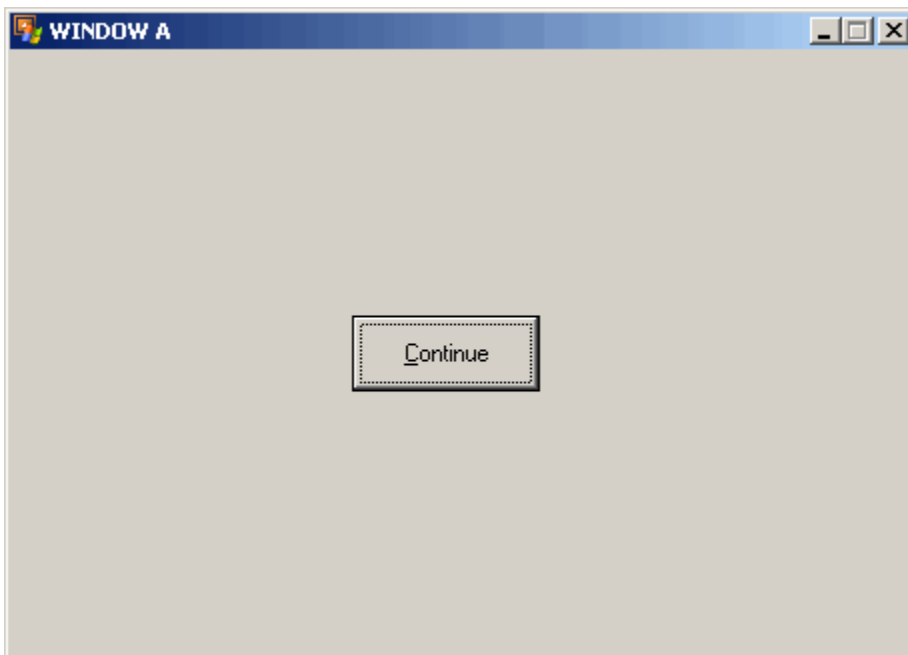
**Sample of second diagram**



### Simulating the Window Flow

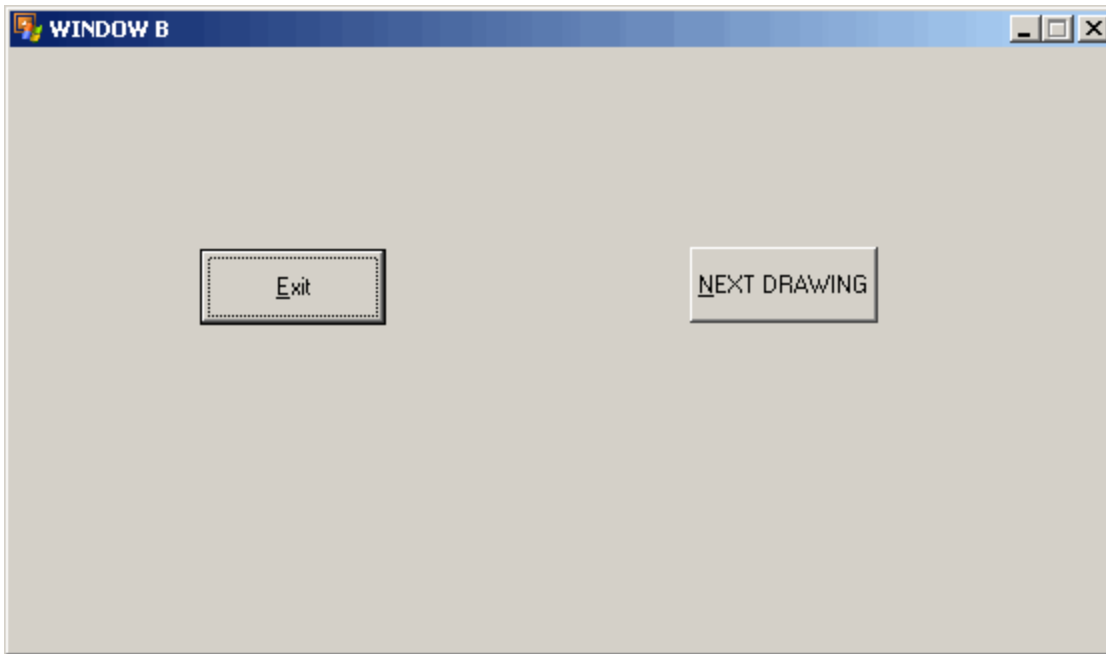
1. Select **Window > <name of diagram>** to return to the WFD, where **<name of diagram>** is the name you chose when you saved your diagram in [Placing Objects in the Diagram](#).
2. In this first drawing, click PROCESS.
3. Select **Analysis > Simulate Window Flow**.  
WINDOW\_A is displayed.

*Window A in simulation*



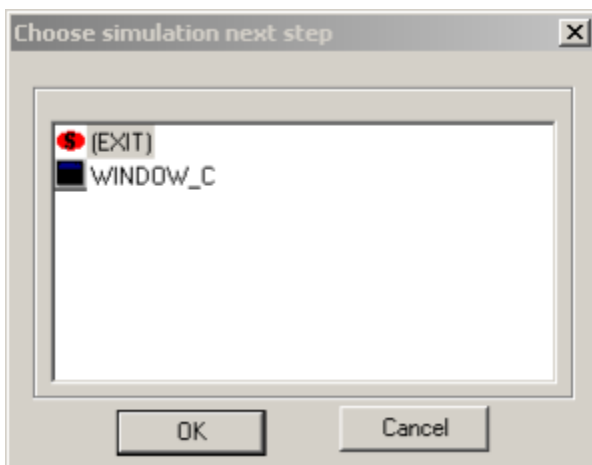
4. Click **CONTINUE**.  
WINDOW\_B is displayed.

*Window B in simulation*



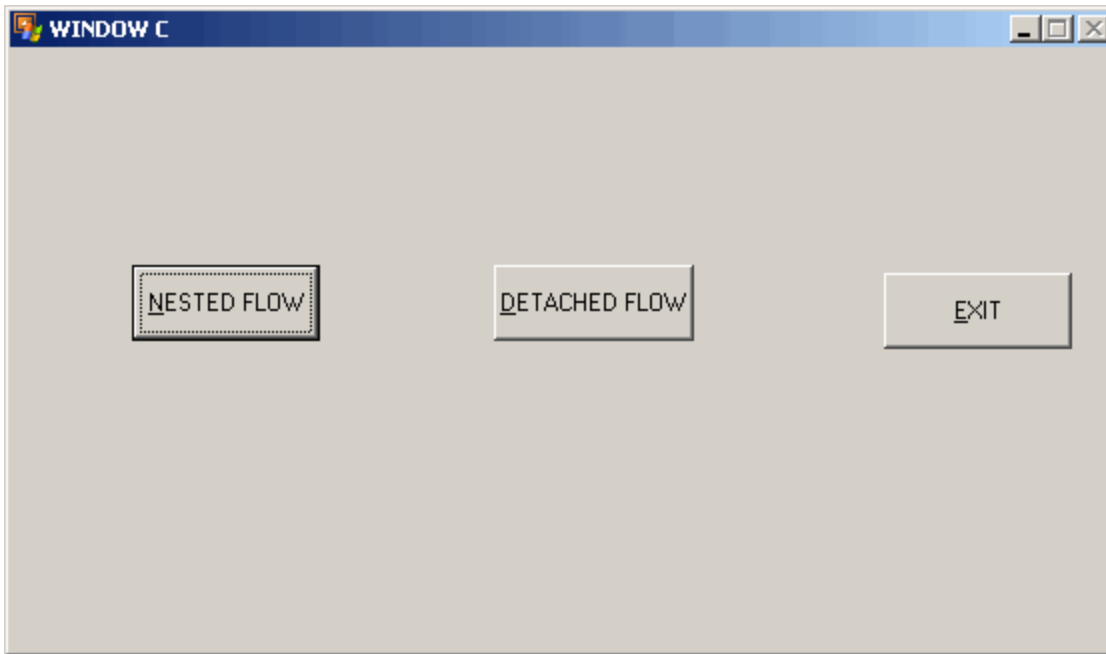
5. Click **NEXT DRAWING**.  
The Choose Simulation Next Step dialog box is displayed.

*Choose simulation next step dialog*



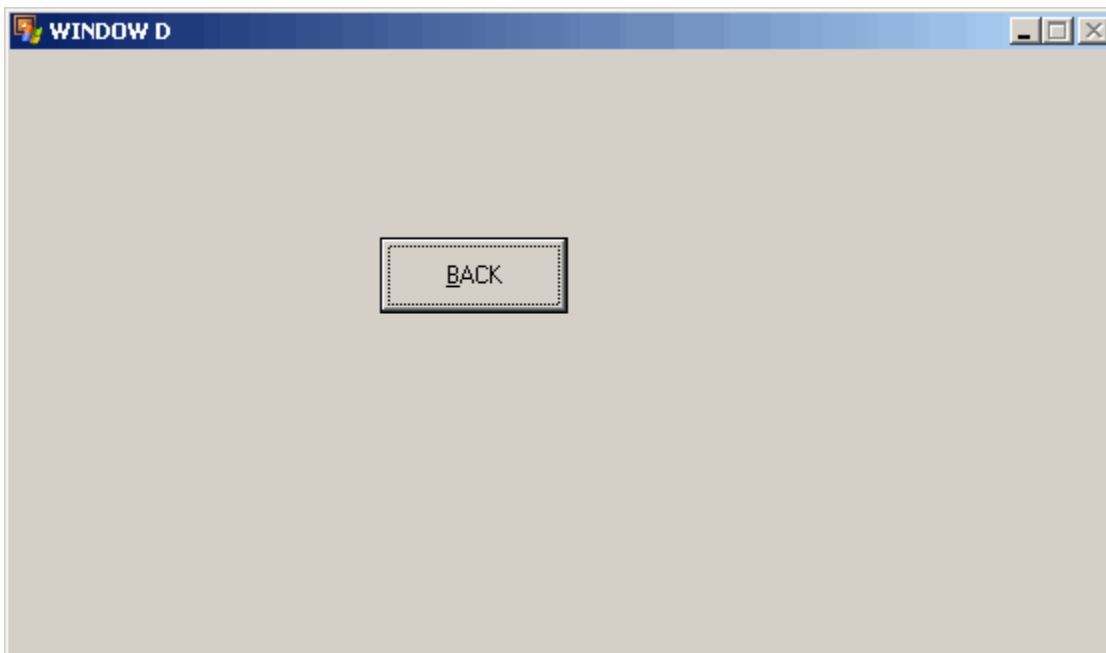
6. Select **WINDOW\_C** and click **OK**.  
WINDOW\_C is displayed. [Window B remains open and loses focus.]

*Window C in simulation*



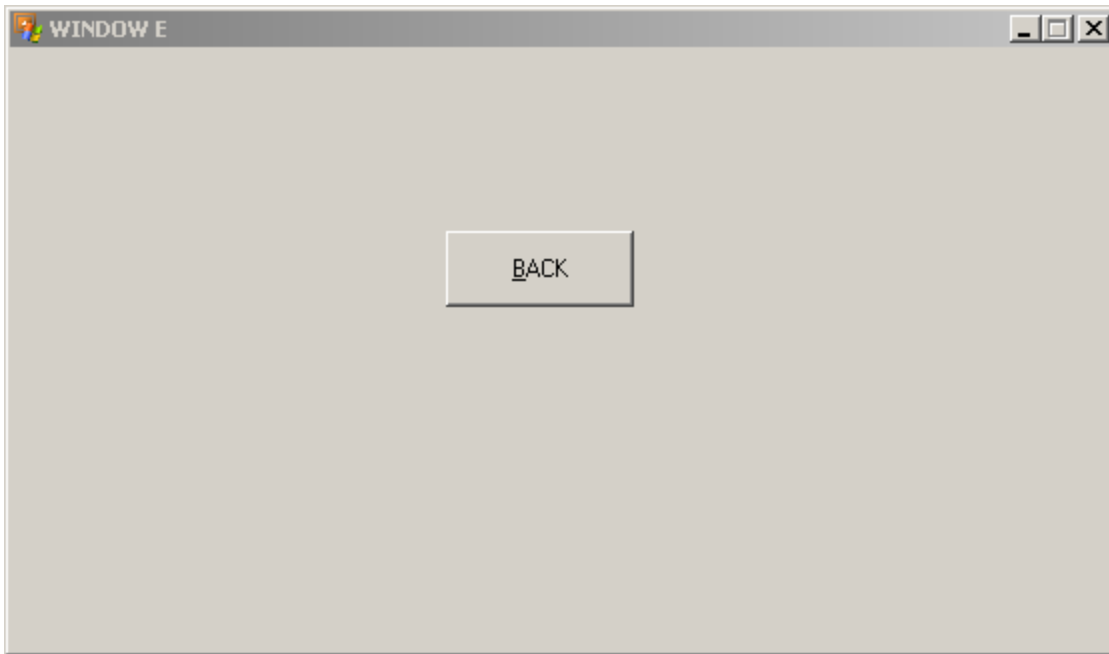
7. Click **NESTED FLOW**.  
WINDOW\_D is displayed and has the focus. Window\_B and Window\_C remain open.

*Window D in simulation*



8. Click **BACK**. Window\_D closes and WINDOW\_C has the focus.
9. Click **DETACHED FLOW**.  
WINDOW\_E is displayed, but WINDOW\_C still has the focus.

*Window E in simulation*

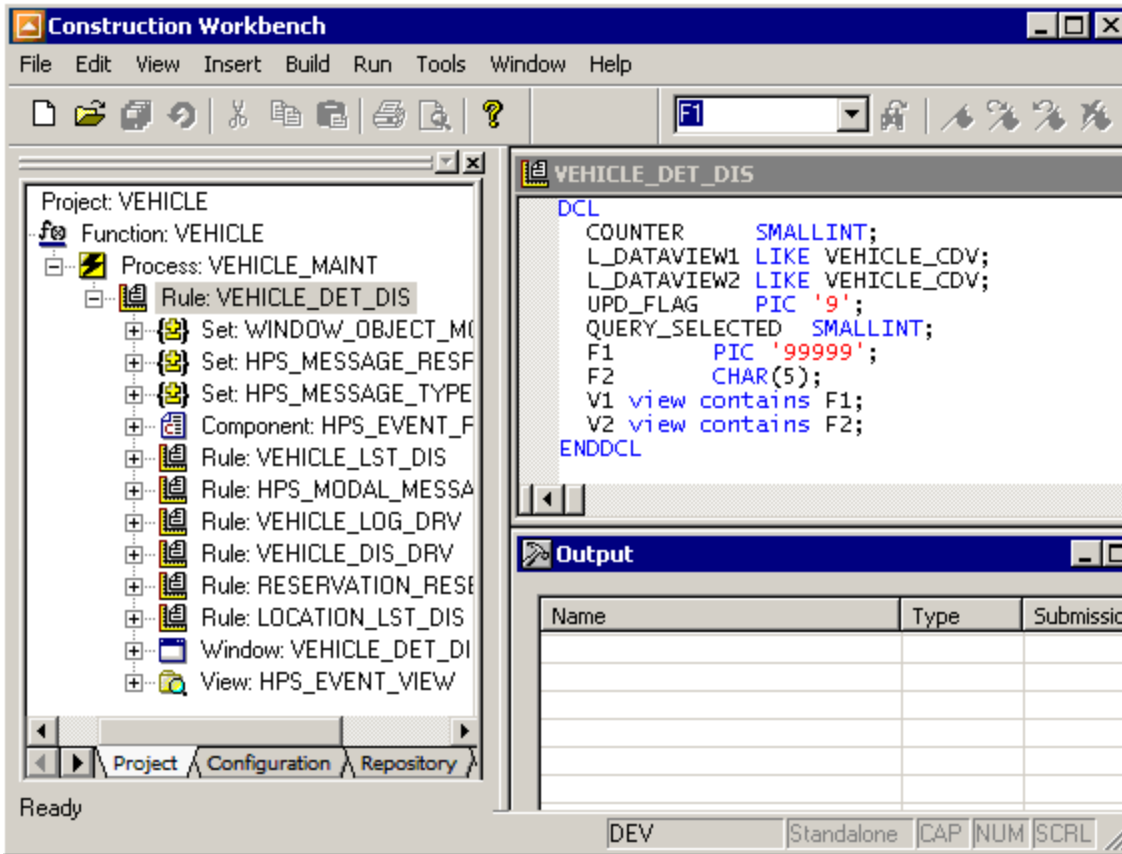


10. Click **BACK**.  
Window\_E closes and WINDOW\_C again has the focus.
11. Click **EXIT**.  
Window\_C closes. WINDOW\_B is displayed.
12. Click **EXIT**.  
The simulation is ended.

## Designing the Application

Whether you have already created the objects for an application through forward engineering or not, you will have to design and create the rules using AppBuilder's Rules Language. AppBuilder's Construction Workbench includes the Hierarchy window and Rule Painter to create and edit rules which are then saved as entities in the repository and encapsulate the business logic at a functional level.

### AppBuilder Construction Workbench



From the Rule source code, application code can be generated and user interfaces can be created. The Construction Workbench's Hierarchy window provides a convenient means of accessing objects in the repository as well as establishing the relationships between objects. Creating an application involves understanding both the business logic used to write the rules governing the application and the Hierarchy which establishes the objects and relationships in the application. AppBuilder utilizes a data model which provides entities and relationships. Your application's design must conform to that model in order to be transformed from Rules Language to a deployed application. Some basic entities in the model are included in the following table:

#### Common AppBuilder entities

Object	Description
Field	This is the simplest object type, representing a variable. It is the smallest subdivision of data (such as a column in a DB2 table), or part of the input or output definition of other object types (such as the view a window owns). Fields can have numeric formats, character formats, or date and time formats. Their only possible parent is a view.
View	Views define input and output data structures of a rule, a component, a window, a file, and a section. Views include other views and fields, which can create very sophisticated data structures. Possible parents are view, rule, window, and component.
Window	Windows are the interactive user interface to an application. You can paint objects on windows to let end users view data in different formats, modify data, or control application execution. These objects include check boxes, combo boxes, edit fields, list boxes, multicolumn list boxes, push buttons, radio buttons, menus, and static text. In order to display data, a window can have only one view attached to it. The only possible parent object for a window is a rule.
Rule	A rule is, in essence, a program (or subprogram, routine) written in Rules Language. It defines the logic of a process, controls the execution of other rules and components, converses windows and reports, and accesses files. A Rule can be called by other Rules. The first Rule in an application hierarchy is known as the Root Rule, and is normally attached to a Process object.
Component	A Component is a program, like a Rule, but it is written in a normal 3GL, such as Java, Cobol, C, PL/1 or Assembler. Components can be used to do things the AppBuilder Rules Language either cannot do or cannot do quickly enough. This might be a complicated arithmetic algorithm (such as exponentiation), non-SQL data access logic (such as an IMS database interface), or hardware-specific functions (such as time/date stamping). Because components are written in a language specific to one environment, they are not portable between environments. AppBuilder provides some system components as an extension of the Rules Language. The possible parents for a component include rules or other components.
Set	A set is a collection of symbols and their values. There are four different types of sets in AppBuilder. Possible parents of a set include rules, windows, and components.

Step-by-step instructions on how to perform tasks in the Hierarchy window and the Rule Painter window of Construction Workbench are discussed in the *Development Tools Reference Guide*. The *Rules Language Reference Guide* provides the rule syntax and additional information



about how to construct statements with Rules Language elements. This section provides an overview of the following topics:

- [Rules and Rules Language](#)
- [Designing Rules](#)
- [Writing AppBuilder Rules](#)
- [Other Considerations for Rules](#)

## Rules and Rules Language

Coding the rules is one of the most common and yet important tasks when creating an AppBuilder application. A rule is a series of programming statements that define the logic of the application - how the entities that comprise the application act and interact. In AppBuilder, you use Rule Painter and the Hierarchy window to code rules and develop the structure of the application.

A rule is roughly analogous to a separate programming procedure in a computer language such as C or COBOL. Just as you use C or COBOL language constructs to build a procedure, you use AppBuilder's Rules Language to build rules. The Rules Language provides the statements, syntax, keywords, and arguments required. For full details on the AppBuilder Rules Language, see the *Rules Language Reference Guide*.

### Sample Rule

The AppBuilder Rules Language excludes some platform-specific features of third-generation programming languages because it concerns itself mainly with program logic. In this respect, it resembles a more traditional programming tool: pseudocode. You can usually translate a pseudocode statement to a corresponding Rules Language statement.

For example, a simple series of actions in a typical business application might include the following:

1. Copy a customer's detail information from a database to a window.
2. Display the window to show the data.
3. If the user selects Update, invoke another module to write the data to a file and then return a code to the initiating module to indicate the user's changes.
4. If the user selects Cancel, return a code to indicate that the user did not make any changes.

In pseudocode, the above tasks might look like this:

```
PASS Customer detail information TO Customer detail window data
DRAW Window
IF User selects 'UPDATE' then
    PASS Customer detail window data TO Update module
    CALL Update module
    IF Update module fails
        PASS Set of customer detail error messages TO Window message module
        PASS Appropriate message IN Set of customer detail error messages TO Window message module
        PASS Customer detail window data TO Window message module
        CALL Window message module
    ELSE
        PASS Update code TO Invoking module
    ENDIF
ELSE
    PASS No change code TO Invoking module
ENDIF
```

A sample excerpt from a rule that accomplishes the same tasks follows. The words connected with underscores are specific instances of AppBuilder object types.

```

MAP DISPLAY_CUSTOMER_DETAIL_I TO CUSTOMER_DETAIL
CONVERSE WINDOW CUSTOMER_DETAIL
CASEOF EVENT_NAME OF HPS_EVENT_VIEW
CASE 'HPS_MENU_SELECT'
  CASEOF EVENT_SOURCE OF HPS_EVENT_VIEW
  CASE 'UPDATE' MAP CUSTOMER_DETAIL TO UPDATE_CUSTOMER_DETAIL_I
    MAP CUSTOMER_DETAIL TO UPDATE_CUSTOMER_DETAIL_I
    USE RULE UPDATE_CUSTOMER_DETAIL
    IF RETURN_CODE OF UPDATE_CUSTOMER_DETAIL_O = 'FAILURE'
      MAP CUSTOMER_DETAIL_FILE_MESSAGES TO MESSAGE_SET_NAME OF
        SET_WINDOW_MESSAGE_I
      MAP UPDATE_FAILED IN CUSTOMER_DETAIL_FILE_MESSAGES TO
        TEXT_CODE OF SET_WINDOW_MESSAGE_I
      MAP CUSTOMER_DETAIL TO WINDOW_LONG_NAME OF
        SET_WINDOW_MESSAGE_I
      USE COMPONENT SET_WINDOW_MESSAGE
    ELSE
      MAP 'UPDATE' TO RETURN_CODE OF DISPLAY_CUSTOMER_DETAIL_O
    ENDIF
  CASE OTHER
    MAP 'NO_CHANGE' TO RETURN_CODE OF DISPLAY_CUSTOMER_DETAIL_O
  ENDCASE
ENDCASE

```

## Rule Syntax

Rules can be written in either of two formats. You can use MAP notation or SET notation. The distinction is:

```

MAP field OF view TO destination-field OF destination-view
SET destination-view.destination-field:=view.field

```

In general, both are valid. For more information on rules language, see the *Rules Language Reference Guide*.

## Designing Rules

Consider the following before writing your rules:

The properties of the rule affect a rule during preparation and execution.

There are practical limitations on the scope and size of a rule. See [Understanding the Scope of a Rule](#).

The physical hierarchy of an application significantly affects how you code a rule and the entities on which it can act. Thus, it is important to design and construct the hierarchy before writing the rules in it. Refer to [Using an Application Functional Hierarchy](#).

Verifying the rule is an important first step before executing the application. The execution environment of the application also affects the rule; refer to the *Deploying Applications Guide* for information about preparing the rule for the particular execution environment.

The following topics are discussed in this section:

- [Understanding Types of Rules](#)
- [Using an Application Functional Hierarchy](#)
- [Understanding the Scope of a Rule](#)
- [Embedding SQL Code in a Rule](#)
- [Passing Data in a Rule](#)
- [Mapping Data to a View](#)
- [Verifying Rules](#)

### Understanding Types of Rules

Generally, there are three types of rules:

- [Event-Driven Rules](#)
- [Converse Rules](#)
- [Non-Display Rules](#)

## Event-Driven Rules

Event-driven applications can only be prepared as Java applications. The most significant difference between display rules for the Java client and display rules for other platforms is that Java client display rules use event procedures rather than a converse loop to respond to user actions. Referring to a rule as event-driven means that end-user actions, such as pressing a mouse button or selecting a menu item, generate events. These events are sent to the rule where they are handled by special procedures called event procedures. While event-driven rules are generally display rules, it is possible for non-display rules to have event handlers for error events.

Event procedures are defined within the rule itself using a special syntax and contain the logic needed to respond to the event. These events and event procedures play a central role in event-driven programming. In fact, writing event-driven applications is largely a process of determining which events you need to respond to and writing event procedures that provide the appropriate response.

Event-driven rules also allow the use of ObjectSpeak statements. ObjectSpeak is an object-based Rules Language syntax, used to interact with window objects (such as edit fields and push buttons) for Java clients at execution time. ObjectSpeak is a set of extensions for the Rules Language that supports industry standard "dot" notation for accessing object properties and methods. Refer to the *ObjectSpeak Reference Guide* and *Rules Language Reference Guide* for more information.

## Converse Rules

Converse rules are used for developing C runtime applications with AppBuilder. In a traditional AppBuilder application, display rules are typically structured as loops containing a converse statement. The loop executes until an exit condition is met, for example, a close event on the window. Actions or events on the window are handled within the loop or rules called from the loop. Converse rules cannot be prepared for Java.

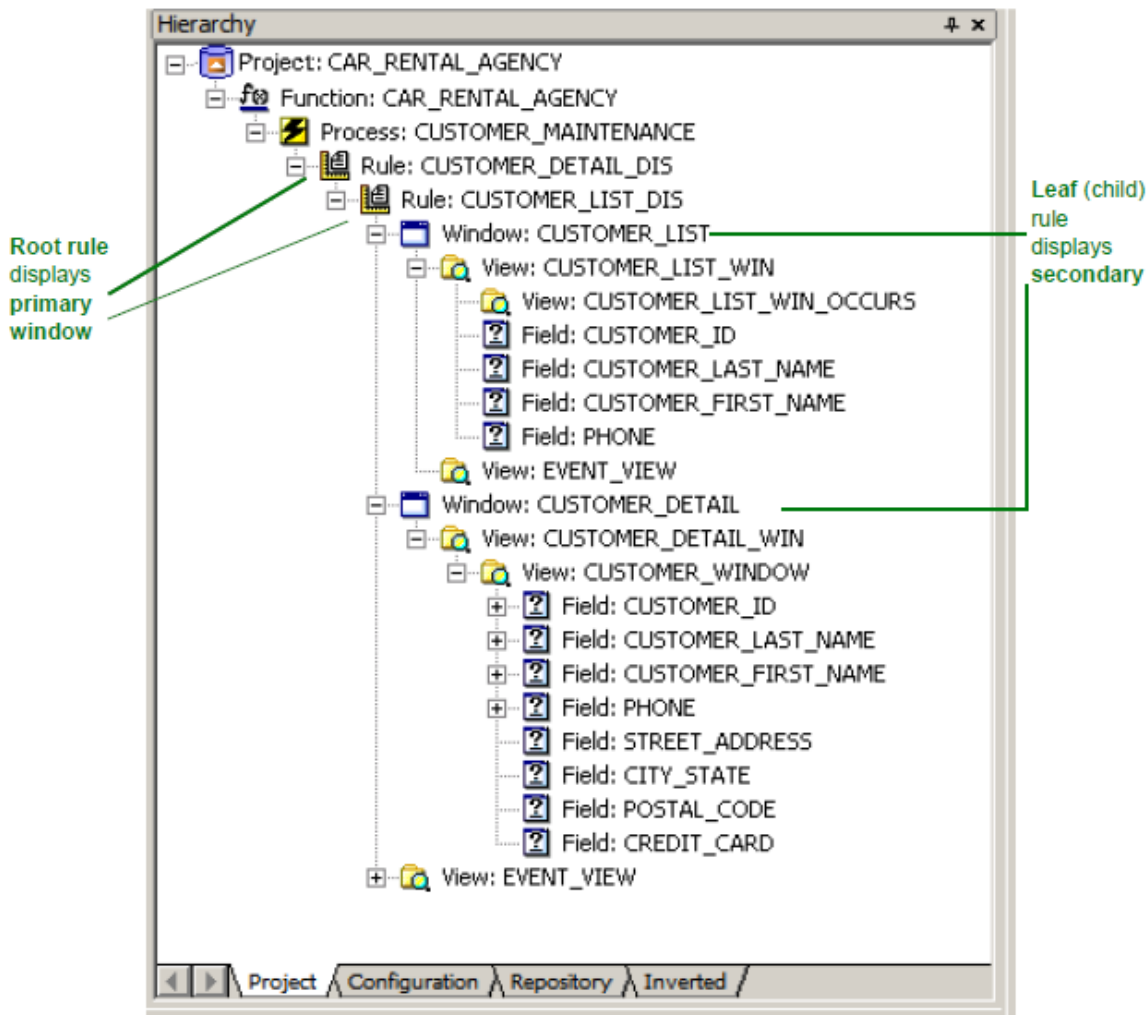
## Non-Display Rules

Event-driven rules and converse rules are generally considered display rules. All other rules not included in the event-driven or converse categories are referred to as non-display rules. They encompass all rules below the presentation layer of an application.

## Using an Application Functional Hierarchy

An application hierarchy depicts the functional structure of an application and the available data structures. The hierarchy contains a function, the processes of a function, and the rules for each process. How a rule relates to views and other objects in an application hierarchy determines how you can use a rule.

### *A typical application hierarchy*



A process with a relationship to a rule is a leaf-level process. Each leaf-level process is a separate application. The rule to which a leaf-level process relates is called a root rule because it is the root of the rest of the application hierarchy. All other objects in an application are descendants of the root rule. Only one root rule is allowed in each process. Although AppBuilder does not stop you from creating more than one root rule for one process in the hierarchy, this generates an error during the upload or the function prepare.

In a well-designed application, each rule performs a specific action, such as routing the logic flow to another rule, displaying a window, accessing a database, or performing a calculation.

A rule can be a parent or a child, depending on its position in the hierarchy. The root rule is always a parent rule. From that point on in the hierarchy, a rule is both child *and* parent. The exception is the bottom rule of a hierarchy which is only a child rule. Rules at the same level can be referred to as siblings. The leaf rules are children only, even if they have windows and views attached to them. A leaf rule can decompose into a window, views, sets, bitmaps, components, files, physical events, reports, or component folders. [A typical application hierarchy](#) shows a typical hierarchy.

Additional considerations for creating a functional hierarchy are as follows:

- There must be a display rule for every window in the application. A display rule can only have one window as a child.
- There should be a database rule for each database action. For example, a separate rule should be created for each insert, update, delete, select, fetch, count action taken on each database table. Some of these rules can be generated by running the 'CRUD Rules' Turbocycler template.
- For transactions that access more than one table, a driver rule can be created that groups together multiple rules and commits changes only after each rule is successful. There are some exceptions to this approach. For example, fetches may need to join together multiple DB2 tables in one query to bring back the appropriate data.

Application architecture should be considered as well. The number of cross-platform calls should be minimized for performance.

Consider the following scenario. There is a PC rule that needs to select data from three separate tables on a server. If the PC rule calls three separate DB2 select rules one at a time, there will be three remote calls. Instead, a server transaction rule can be created that calls the three select rules. Then the PC rule calls the transaction rule once, which gathers the selected data together and returns in one call. This eliminates two server calls.

### Understanding the Scope of a Rule

To write a rule effectively, you need to know about its scope or data universe — the collection of data structures that it can access.

A rule does not know about all the data that exists in an application hierarchy. Referring to such a large amount of data takes up too much

memory, processing time, and forces you to qualify references in your rule to views and fields to make those references unique. For a rule to prepare correctly, it must reference only entities that it knows about, which can include a view attached to the rule or other input/output views linked to immediate child rules and components. Refer to the following example.

#### [Example Code for Display\\_Customer\\_Detail](#)

```
map DISPLAY_CUSTOMER_DETAIL_I to CUSTOMER_DETAIL
converse WINDOW CUSTOMER_DETAIL
if EVENT_SOURCE of HPS_EVENT_VIEW = 'UPDATE'
    map CUSTOMER_DETAIL to UPDATE_CUSTOMER_DETAIL_I
    use RULE UPDATE_CUSTOMER_DETAIL
else
    map 'NO_CHANGE' to RETURN_CODE of DISPLAY_CUSTOMER_DETAIL_O
endif

//The rule accesses the following data items:

DISPLAY_CUSTOMER_DETAIL_I
CUSTOMER_DETAIL
EVENT_SOURCE
UPDATE_CUSTOMER_DETAIL_I
RETURN_CODE
DISPLAY_CUSTOMER_DETAIL_O
```

The rule must have these data items associated with it for the rule to prepare. In addition, although the UPDATE\_CUSTOMER\_DETAIL rule is not a data item, it contains data that the parent rule must know about, and therefore must be a child of DISPLAY\_CUSTOMER\_DETAIL. The same is true for components and windows that the rule uses.

#### **Embedding SQL Code in a Rule**

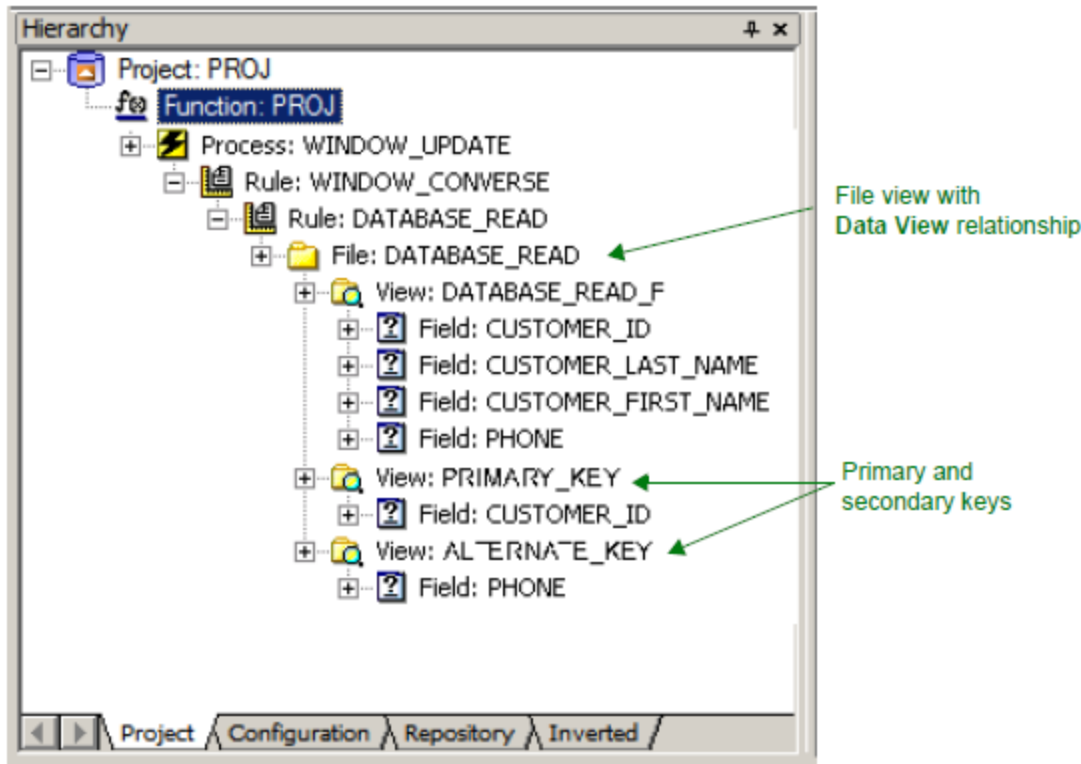
You can insert SQL code (query language commands for database access) into a rule using the SQL Builder within Rule Painter. The SQL Builder allows you to customize the generated SQL code before adding it to the rule. For step-by-step instructions to insert the SQL code, see the discussion of using SQL Builder in the *Development Tools Reference Guide*.

#### **Adding a Database Access Hierarchy**

To make it possible for an application to access a database, add SQL code to your Rule object and then add the necessary File objects and Field objects to the hierarchy for a rule that accesses a database file.

To allow access to a database, use the Project tab in the Hierarchy Window to model the hierarchy for a rule that accesses a file. [Database access hierarchy](#) shows an example of this hierarchy.

#### ***Database access hierarchy***



The File object in the hierarchy corresponds to a table in a database; the fields within the file View object correspond to the columns in the table. There can be only one File object attached to a View object, but a File object can have multiple View objects, as shown in the diagram above. Make sure you set the following properties of objects in the hierarchy for the database.

- In the database Rule object, set the property *DBMS Usage* to **DB2**. This setting does not set the type of database that the rule accesses; it simply specifies that SQL database access is used in this rule.
- In the File object, set the *Implementation name* property to the actual name of the database table.
- In each Field object of the first view object under the File object (called the file view), set the *Implementation name* to the actual name of the column in the database table.

There are several kinds of database views (defined by their relationship properties):

- Data view - is the database view
- Primary index view - is the primary key view
- Alternate view - is the alternate key view

When you build the file hierarchy, remember that:

1. The first view under the file object is the *file view*. The relationship between the file and the view must be of type **Data View**. This view must contain at least one field.
  2. Files can have a *primary key view*. The fields of this view must be a subset of the fields of this file view. These fields form unique indices.
  3. Files can have an *alternate key view*. The fields of this view must be a subset of the fields of this file view. These fields may or may not form non-unique indices.
- For more information about rule views, refer to [Passing Data in a Rule](#). For information about configuring database access in a distributed environment (not just stand-alone mode), refer to the *Deploying Applications Guide*.

## Passing Data in a Rule

An AppBuilder application passes data in fields; the fields are collected into views. Rules use views to pass data to each other and to the other entities connected to them. Once you have created a view and linked it to a rule, you must define its type. That is, you define the type on the relationship. Unless a view has a relationship to a rule, you cannot define its type.

To define a view's type, follow the steps below:

1. From the Project tab of the Hierarchy window, right-click the **View** object.
2. Select **Relationship Properties** from the pop-up menu. A Properties window appears.
3. From the View Usage drop-down list box, select the appropriate value. Any rule can have views, but root rules should have only a work view.

The data scope of a rule includes all views and sets linked to it, views attached to Window, File and Physical Event objects linked to it, and input/output views linked to immediate child rules and components. It also includes internally defined views and fields (within DCL..ENDDCL section) and an SQLCA view that is automatically available to rules defined as accessing a database.

This section discusses the following types of views:

- [Input View](#)
- [Output View](#)
- [Input & Output View](#)
- [Work View](#)
- [Global View](#)

When you prepare a rule, the AppBuilder environment automatically updates the data universe of a rule (incorporating any changes you made to relationships in the rule hierarchy) before checking the syntax of your rule and generating code. The rule source is scanned for references to rules, components, windows, and sets. If any of these elements do not match the relationships in the rule hierarchy in the repository, the preparation of that rule fails.



Application preparation fails if the leaf-level (lowest level) view does not include at least one field. That is, although a view can refer to any number of subordinate views, eventually the hierarchy must end in at least one field for all superior views to be valid.

### **Input View**

An input view passes data into a rule. A rule can have only one input view.

### **Output View**

An output view passes data from a rule to its parent rule. A rule can have only one output view.

### **Input & Output View**

A rule view is an input and output view when the rule can change its input data. For example, if you write a rule to change all lowercase letters to uppercase, you can define the single view of a rule as **Input & Output**.

A rule can have only one Input & Output view. If a rule has an Input & Output view, it cannot also have a separate input view or output view.

### **Work View**

A work view is local to its parent rule, which means that a rule cannot see the work view of any other rule. A rule can read from or write to its own work view. A rule can have more than one work view, but it is not necessary for a rule to have any work view. A work view is important for the support of events. For instance, you must attach the predefined system view HPS\_EVENT\_VIEW as a work view to any rule that captures events. You also use a work view to pass data between detached rules. See [Creating Event-Driven and Converse Applications](#) for more information.

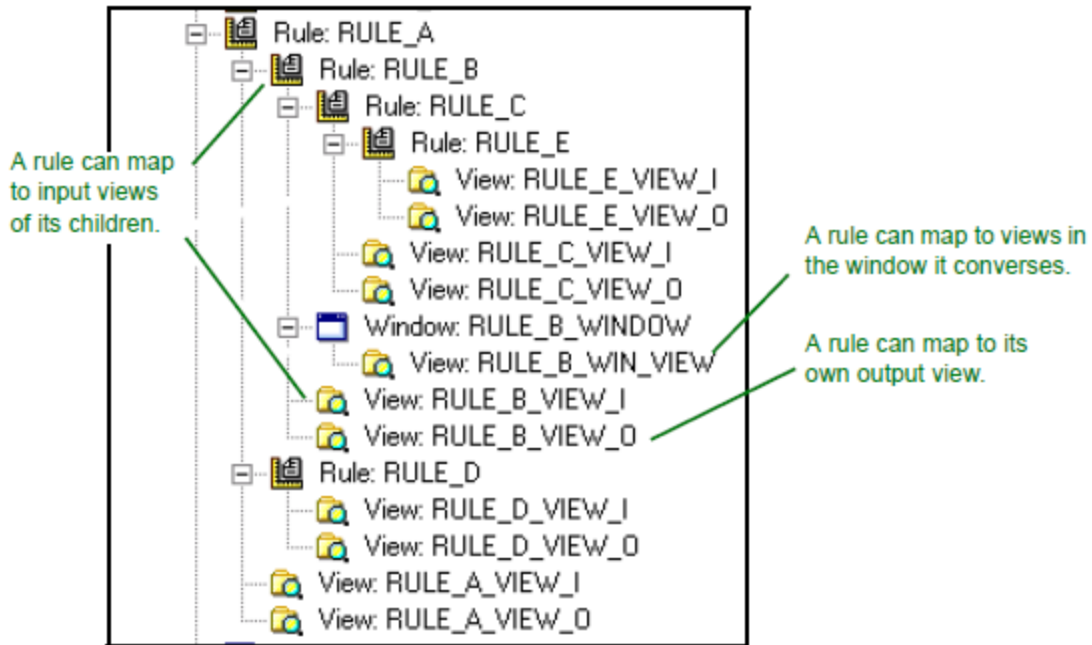
### **Global View**

A global view is a way for multiple rules to share the same data. That is, all rules in the application attached to a global view share the same data. Global views attach to rules, not to windows. The view must be attached to all rules that share its data.

### **Mapping Data to a View**

A rule can map data to or from any view in its data universe (scope). While technically permitted, it is a bad practice for a rule to write data to its own input view or to output views of its child rules or components. [What a rule can map to](#) illustrates the views to which RULE\_B can map.

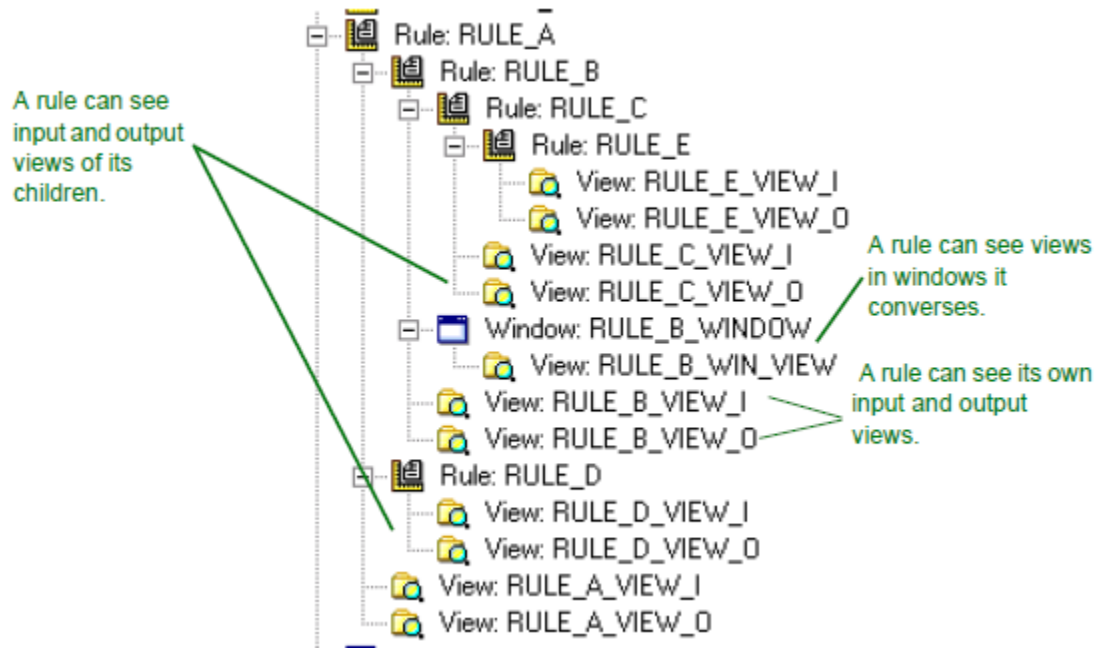
#### ***What a rule can map to***



**Mapping Scenario**

Given the entities and relationships in [What a rule can see](#), you can write code for RULE\_B to map data from its own input view, output view, or local work view (if it had one). In addition, you can map data from the input or output views (or both) of any rule that RULE\_B uses, as well as any window it converses.

**What a rule can see**



Note that RULE\_B cannot see the input or output views of RULE\_A that uses it. All data that RULE\_B rule needs to execute must be available to it in its own input view (to which RULE\_A maps data before using RULE\_B).

Also, RULE\_B cannot see the input or output views of its "grandchild," RULE\_E, which RULE\_C uses. All you need to know about a rule is what input to give it to get its output. That it might use other rules or gain access to files is irrelevant. Thus RULE\_B should not need to "see" the views of its grandchildren, nor should it be able to.





Be careful when reusing views. A root view does not have a view as its parent. The AppBuilder environment considers all root views with the same name within the same data scope to be the same view. Mapping information to one such view maps the information to all root views with the same name.

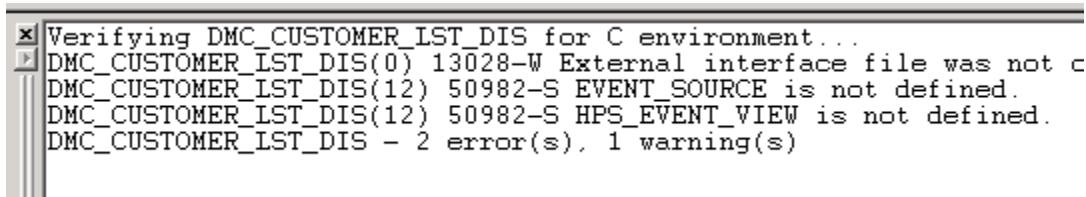
## Verifying Rules

You can verify rules to ensure that they are syntactically and logically correct. Before verifying, you must specify the programming language in which the rule will be generated so that AppBuilder knows the language syntax against which to verify. After selecting the language, you can choose to Verify, which verifies the rule syntax, the structure, and validity of the hierarchy. Follow the steps below to verify a rule:

1. Open the rule to verify by:
  - Right-clicking on the Rule object in the Hierarchy window and selecting **Open Rule**, or
  - Double-clicking on the Rule object in the Hierarchy window, or
  - Highlighting the rule name in the Hierarchy window and pressing **Enter**.  
A Rule Painter window opens in the Work Area.
2. Choose the verification language:
  - From the Construction Workbench menu bar, select **Build > Verification Language**, and select the language, or
  - Right-click in the Rule Painter window of the open rule and select **Verification Language**; then, select the language.
3. Verify the rule:
  - From the Construction Workbench menu bar, select **Build > Verify** or
  - Right-click in the Rule Painter window of the rule and select **Verify** from the pop-up menu, or
  - With the Rule Painter window in focus, to Verify press **Alt+ F7**.

The **Verify** is the best way to find errors in your rule source. This validates the rule source against the hierarchy. The system checks the rule and displays the results on the **Verify** tab of the Status window, as in [Verify tab of Status window](#). For details on error and warning messages, refer to the *Messages Reference Guide*.

### Verify tab of Status window



```
Verifying DMC_CUSTOMER_LST_DIS for C environment...
DMC_CUSTOMER_LST_DIS(0) 13028-W External interface file was not c
DMC_CUSTOMER_LST_DIS(12) 50982-S EVENT_SOURCE is not defined.
DMC_CUSTOMER_LST_DIS(12) 50982-S HPS_EVENT_VIEW is not defined.
DMC_CUSTOMER_LST_DIS - 2 error(s), 1 warning(s)
```

For more information about debugging an application, refer to the *Debugging Applications Guide*.

For more information about the Output window (and the Verify tab) refer to the *Development Tools Reference Guide*.

### Keeping Within Limits

In theory, AppBuilder does not limit the number of rules or components a single rule can use; however, in practice, the following limitations apply:

- Each rule in an online application can converse only one window for communication with the end user. When one of your rules converses a window, you need Rules Language code to interpret and respond to the end-user input.
- As a practical guideline, a single rule should not invoke more than five to nine objects. If your rule calls more than nine other objects, you may be putting too much functionality into one rule. It is more efficient to break it into logical subunits.
- Even though there is no maximum size for a rule, factors such as the operating system, the compiler, and available memory place a practical size limit. To make a rule easy to maintain and reuse, do not exceed 100 to 140 lines (about two printed pages). If your rule exceeds this limit, divide it into smaller rules.

## Writing AppBuilder Rules

This section discusses issues you need to consider when writing an AppBuilder rule, including working with repository objects and understanding a rule's scope. The following topics are discussed in this section:

- [Using Java Objects](#)
- [Understanding Thin Client Development](#)
- [Understanding Event Procedure Syntax in Java](#)
- [Manipulating the Interface](#)
- [Using Common Procedures](#)
- [Creating Objects at Execution Time with Java](#)
- [Using Display Rules](#)

- [Optimizing Java Server Rules](#)

## Using Java Objects

ObjectSpeak is an object-based Rules Language syntax, which is a subset of the AppBuilder Rules Language. AppBuilder uses ObjectSpeak to interact with window objects (such as edit fields and push buttons) at execution time. For Java, rules and windows are objects. This means that a rule and a window may have properties, methods, and events that can manipulate and interact with that rule or window.

- **Properties** represent the state of the object. The properties of an object are its attributes, such as height, foreground color, and visibility.
- A property consists of a single *value* of a specified *type*. For example, the type of the **Height** property is **integer**, the type of the **Foreground** property is **color**, and the type of the **Visible** property is **boolean**.
- You set properties in the design phase using Window Painter, and you can change or modify property values at execution time. For more details, see [Changing Property Values at Execution](#).
- **Methods** are the input to the object. The method of an object is its function or procedure that can be called on an object. For example, the **MessageBox** object has a **Show** method that is used to display the message box. For more information, see [Using Methods](#).
- **Events** are output from the object. The event is the actions that are triggered by objects, often in response to a user action. For example, push buttons have a **Click** event that is triggered when the user clicks on the button with a mouse. For more information, see [Event-Driven Rules in Java](#).

Refer to the *ObjectSpeak Reference Guide* for a description of properties, methods, and events in ObjectSpeak.

### Notes on Some Objects in Java Environment

In Java, the Chart and Hot Spot objects are not supported.

In Java code for thick client and servlet, you must define unique system identifiers (HPSIDs) for all window objects. If you have existing C applications that can use duplicates, be aware that changing to Java requires creating unique system identifiers.

### Objects for HTML Clients

In an HTML client application, rules are executed in a Web or application server environment. The application sends information about the windows to the client browser in HTML format. Some events, such as **FocusGained**, **FocusLost**, or **Click on Edit Field** must be handled by a client-side script instead of an AppBuilder rule. The object properties, such as **setFont** or **setBackgroundcolor** update the style attributes of the HTML file of a window.

There are special ObjectSpeak methods supported for thin-client applications like HTML, for example, using cookies to save user profiles. Refer to the **ObjectSpeak Reference Guide** for more information about these methods.

### Changing Property Values at Execution

Many properties for user interface objects can be set during design in Window Painter. However, it is useful to be able to manipulate the properties of an object at execution time as well. For example, you might want to disable or enable a menu item dynamically in response to the current state of the program. Or, you might want to protect an edit field, making it non-editable.

AppBuilder uses ObjectSpeak to interact with properties of objects and methods at execution time. ObjectSpeak uses a "dot" notation to indicate that a property or method of an object is being accessed. The following examples demonstrate ObjectSpeak syntax for accessing the properties and methods of an object.

#### [Example - Assigning and Determining Property Values](#)

##### **Example 1: Assigning a new value to a property**

This example demonstrates how to change the value of a property.

Use the name (HPSID) of the object (**QueryButton**), followed by a period, followed by the name of the property (**Text**).

```
Set QueryButton.Text := 'Query'
```

This code changes the **Text** property of the push button called QueryButton to **Query**.

##### **Example 2: Determining the current value of a property**

This example demonstrates how to determine the *current* value of a property.

Use the name of the object (**QueryButton**), followed by a period, followed by the name of the property (**Text**).

```
if QueryButton.Text = 'Query'  
    use rule QUERY_RULE  
endif
```

## Using Methods

Methods are functions or procedures that can be called on an object – they provide a way to tell an object to perform a specific task. For more information about calling methods, refer to the *ObjectSpeak Reference Guide*. The following methods are illustrated in the example below:

- Show Method tells an object to display a value.
- Set Method assigns values to properties. The Set Method achieves the same result as a Map statement.

### [Examples of using the Show and Set Methods](#)

#### **Example 1: Show Method - Displaying a message box**

In this example, Java client contains an object named **MessageBox** that displays a message on the screen. The **MessageBox** object has a method named **Show** that causes the message box to appear on the screen:

```
dcl
    MyMessageBox object type MessageBox;
enddcl
map new MessageBox to MyMessageBox
map 'Invalid Data' to MyMessageBox.Title
map 'The amount in the Interest field is too large'
    to MyMessageBox.Message
MyMessageBox.Show
```

This example:

1. Creates a **MessageBox** object and maps it to the local variable **MyMessageBox**.
2. Assigns the desired title and message strings to the **Title** and **Message** properties.
3. Calls the **Show** method to display the message box.

#### **Example 2: Set Method - Assigning a value to a property**

Like the Map statement, a Set method is used to assign values to properties. Although both mechanisms (set method and map statement) can be used to change the value of a property, the Set method provides a slightly more compact notation. In general, each property has a set method that assigns a value to the property. Set methods take exactly one parameter, whose type is the same as that of the property. By convention, the name of a property set method is simply the property name prefixed by **Set**. For example, a string property named **Text** has a set method named **SetText()** that takes a string parameter. Likewise, a boolean property named **AutoSelect** has a set method named **SetAutoSelect()** that takes one boolean parameter, as the following code illustrates:

```
NameField.SetAutoSelect(True)
```

Consider the following two lines of code; each accomplish the same action:

```
map 'Query' to QueryButton.Text
```

This line assigns (maps) a value directly to the *Text* property of the push button.

```
QueryButton.SetText('Query')
```

This line calls the **SetText** set method of the push button, passing the new text as a parameter. In this example, **SetText()** is the set method for the **Text** property.

### **Event-Driven Rules in Java**

In *AppBuilder*, all rules that display windows (display rules) are event-driven. However, for traditional *AppBuilder* applications, displaying a window and identifying and processing the events resulting from the window are handled in a procedural fashion and must be explicitly coded. For an event-driven application, displaying the window becomes an implicit process, and to handle events, the rule contains special procedures called event procedures. These procedures are defined using a particular syntax. *AppBuilder*'s Java class library identifies the event type and applicable object and invokes any event procedures for that event type or object. Writing event-driven applications is largely a process of determining which events you need to respond to and writing event procedures that process these appropriately. The following are some of the actions that generate events:

- Opening the window
- Closing the window
- Clicking a push button, radio button, or checkbox
- Selecting a menu item

- Shifting focus to a user interface object, such as an edit field
- Shifting focus away from a user interface object
- Double-clicking an object with the mouse
- Entering erroneous data into an error field
- Trying to close the window when one or more fields contain erroneous data

Refer to the *ObjectSpeak Reference Guide* for a list and description of events in ObjectSpeak.

It is **not** necessary to provide an event handler for every event that may be generated. If the program wants to respond to the event, an event procedure must be defined and the code that responds to the event must be placed within the procedure.

#### Example: Event Procedure

If a window contains a push button named CloseButton that is used to close the window when clicked, the necessary event procedure is:

```
PROC for Click object CloseButton
( e object type ClickEvent )
    MAIN_WINDOW.Terminate
ENDPROC
```

This event procedure responds to the Click event by calling the Terminate method on the window, which causes the window to close. More information about event procedure syntax is explained in [Understanding Event Procedure Syntax in Java](#).

### Understanding Thin Client Development

A thin-client is a Web browser or another HTTP client communicating with a server. You can build AppBuilder applications that can be accessed by any supported Web browser and other HTTP client. AppBuilder enables you to:

- Have your application's client rules execute on an HTTP server (that is, a Web server) or any Java version 2 Enterprise Edition (J2EE) compliant application server. This way, you can deploy your applications without requiring AppBuilder on the client machines.
- Create applications that can be accessed over the Internet, a corporate intranet, or both. Server rules access the database of an application, which can be on any server in the network.
- Convert your AppBuilder applications created for GUI clients into HTML pages for network browser clients. The presentation pieces of an application are translated into HTML, giving it a generic user interface that can run on most platforms.
- Generate a thin-client (HTML client) application.
- Create dynamic Web pages and process information contained in HTML forms.
- Add any technology that Web servers and browsers support (for example, JavaScript or Java applets) to your AppBuilder application.

Your application might require additional resource files, such as multimedia files and graphics, as part of the HTML pages. These miscellaneous files could be any of the following:

- Global resource files common to many or all Web pages (such as CSS style sheets, META tags, etc.). Store these files in a Component Folder defined as a child of the **Process** object in the hierarchy.
- Web page specific resource files, unique to specific pages. Store these files in a Component Folder defined as a child of the **Window** object in the hierarchy.

When you have created the Component Folder object in the correct location in your hierarchy, right-click the Component Folder object and select **Properties**. From the **General** Tab on the Properties window, define your Folder Type as **HPS\_WEB\_RESOURCE**. Component Folder entries with the given Folder Type are prepared with the Window object and are copied to the *images* subdirectory.

From the Properties window, select the **External** tab to add, remove, or modify the contents of the Component Folder. Refer to the *Development Tools Reference Guide* for specific instructions on working with the **External** tab options.

### Understanding Event Procedure Syntax in Java

There are two general forms for event procedures: event handling for a specific object and event handling for all objects of a specific type. Thus, it is possible to write a Java event procedure that handles **Click** events for a *specific* push button and another event procedure that handles **Click** events for *all* the push buttons on a window. [Variable descriptions for pop-up examples](#) shows the variables used in the following Java code examples with a description of each variable. For more information and to view a diagram of Event Procedure Syntax refer to the *Rules Language Reference Guide*.

#### Variable descriptions for pop-up examples

Variable	Description
<ProcedureName>	An optional, unique, user-defined name for the procedure.
<EventName>	The name of the event (for example, <b>Click</b> ).
<ObjectId>	The system identifier ( <b>HPSID</b> ) for the window object, or the long name for the rule or window object.

<ObjectType>	Type of the object (for example, <b>PushButton</b> or <b>EditField</b> ).
<ParameterName>	A user-defined name for the event parameter.
<ParameterType>	The data type of the parameter. The <ParameterType> is always the <EventName> with <b>Event</b> appended.

### [Event Procedure - Specific Object and All Objects of Specific Type](#)

#### **Event Procedure: Specific Object**

The following Java syntax illustrates an event procedure that handles events for a specific object:

```
PROC <ProcedureName> FOR <EventName> OBJECT <ObjectId>
(<ParameterName> object type <ParameterType>)
...
ENDPROC
```

#### **Event Procedure: All Objects of Specific Type**

The following Java syntax illustrates an event procedure that handles events for all objects of a specific type:

```
PROC <ProcedureName> FOR <EventName> TYPE <ObjectType>
(<ParameterName> object type <ParameterType>)
...
ENDPROC
```

See [Variable descriptions for pop-up examples](#) for descriptions of the variables used in these examples.

#### **Naming Conventions**

The code examples used in this section use a convention of generating a name for event procedures by combining the object and event names. For example, for a push button with a system identifier **HPSID** of **CloseButton** (and thus ObjectSpeak name of **CloseButton**), the event procedure for the **Click** event is named **CloseButtonClick**. Also, the parameter name is, by convention, often just a single letter, such as **e** or **p**.

AppBuilder's Construction Workbench generates procedures with these naming conventions.

#### [Example: Event Procedure Naming Conventions](#)

The following is a typical Java event procedure:

```
PROC CloseButtonClick FOR Click OBJECT CloseButton
( e object type ClickEvent )
    MAIN_WINDOW.Terminate
ENDPROC
```

### **Manipulating the Interface**

Many events provide information about the object that triggered the event. This information is contained in two properties of the event procedure parameter: the **HPSID** property and the **Source** property.

- **HPSID** is a string property that contains the system identifier (HPSID) of the object that generated the event. This is a required property for all Java root menu objects.
- **Source** is a reference to the object whose type is **GuiObject**.
- The **GuiObject** is a generic type that is used to represent interface objects, including edit fields, push buttons, menu items, or comboboxes, for example. **GuiObject** has two Boolean properties – **Enabled** and **Visible** – that can be used to manipulate the object that triggered the event.
- Because **GuiObject**, which is the type of **Source**, is a generic type that represents all graphical user interface (GUI) objects, the following code could be used to disable an edit field or any other GUI object when it is clicked.

#### [Example: Using GuiObject](#)

In this Java example, an application has a **Query** button that must be disabled temporarily when it is pressed:

```

PROC FOR Click OBJECT QueryButton
( e object type ClickEvent )
  *> disable the QueryButton <*>
  e.Source.SetEnabled( False )
  *> perform the query <*>
  .
  .
  .
  *> re-enable the QueryButton <*>
  e.Source.SetEnabled( True )
ENDPROC

```

## Using Common Procedures

In addition to event procedures described in [Understanding Event Procedure Syntax in Java](#), AppBuilder supports common procedures. You use common procedures to place frequently-used logic in a single place in the rule structure. Common procedures can also return a value so that they can be called from within expressions. These procedures can be used in both Java and C Windows. The following example is a Java procedure used to display text in a status bar (an edit field) at the bottom of the window.

For more information about common procedures, refer to the *Rules Language Reference Guide*.

### Example: Java Procedure for Status Bar Text

```

PROC UpdateStatusBar( text CHAR(100) )
  StatusBar.SetText( text )
ENDPROC

```

## Creating Objects at Execution Time with Java

Most objects in AppBuilder are created during the design phase with Window Painter. Objects are created during the design phase when the user interface is being created. For example, you can add a push button, list box, or radio button.

You can also create objects dynamically at execution time. Objects, whether created during design in Window Painter or at execution time in rules, can be passed as parameters to other rules or to common procedures within a rule. This allows a great deal of flexibility in designing applications.

Consider the following section of a rule:

```

dcl
  QueryButton object type PushButton;
enddcl
.
.
.
map new PushButton to QueryButton

//The push button also has to be added to the window for it to be displayed:
//Window_name.addChild( QueryButton)

```

This rule declares a local variable named **QueryButton** of type **PushButton**. The **map** statement uses the **new** keyword to create an instance of a **PushButton** object and assign it to the **QueryButton** local variable.

For detailed information about the objects you can create in AppBuilder, refer to the *ObjectSpeak Reference Guide*.

To receive events from objects that are created dynamically at execution time, you must include code to define the event procedure. The procedure shown in the example below dynamically creates a **Timer** object on a window named MAIN\_WINDOW when the user clicks the **UpdateButton** button. Note that the last statement in the Click event procedure for the button specifies that the event procedure (or *handler*) for the timer is the **TimerProc** procedure.

### Example: Procedure that Dynamically Creates Objects

Using the approach in this example, you can dynamically create objects and define event procedures to respond to the events they trigger.

```

dcl
    UpdateTimer object type Timer;
enddcl
*> event procedure for Timer events <*>
PROC TimerProc FOR Timer OBJECT UpdateTimer
(e object type TimerEvent)
    *> do what needs to be done when timer triggers <*>
    *> ??? <*>
ENDPROC

*> respond to Update button click by starting timer <*>
PROC FOR Click OBJECT UpdateButton
(e object type ClickEvent)
    *> create a timer <*>
    map new Timer( MAIN_WINDOW ) to UpdateTimer

    *> set timer properties and start the timer<*>
    UpdateTimer.SetDelay( 1000 )
    UpdateTimer.SetRepeats( True )
    UpdateTimer.Start

    *> dynamically add event procedure to handle
        timer events <*>
    Handler UpdateTimer( TimerProc )
ENDPROC

```

## Using Display Rules

In AppBuilder, event-driven rules that display windows are generally structured as follows:

- Declaration ( dcl...enddcl ) section for local variables
- Standard or non-event procedures
- Event procedures

Although none of these items are required, most applications consist of several sections that contain one or more of the items listed above. Typically, a declaration section is followed by the window **Initialize** and **Terminate** event procedures. These would be followed by additional event procedures to handle push button clicks, menu item selections, and other user actions. The most common event procedure is the **Initialize** event procedure, called after the window is created but before it is shown. This is a good time to initialize the application data and make any needed adjustments to the visual objects on the window.

The most significant difference between display rules for the Java client and display rules for other platforms is that Java client display rules use event procedures rather than a converse loop to respond to user actions. In the Java client, rules that do not display windows are written in the same way as in earlier versions of the product. That is, non-display rules can consist of:

- Declaration (dcl) section for local variables
- Rule body that contains standard procedural code
- Standard or non-event procedures

Thus, in general, a Java client application developed in AppBuilder can consist of display rules that use the new rule structure and non-display rules that use standard Rules Language.

## Optimizing Java Server Rules

Java server rules allow for several optimizations compared to client rules. One such optimization is stateless generation.

Traditionally, AppBuilder has generated stateful classes for rules, where all data within scope of the rule is generated as instance data of the rule class. For such generated classes, an instance of the rule class must be created for each concurrent Web Service or EJB request calling that rule within an application server.

An additional generation option now provides the ability to generate stateless classes for Java server rules, where the rule's data is local to the rule's execution method. Rules generated in this manner can support execution across concurrent server requests within an application server with a single instance of the rule class, thus reducing memory consumed. In previous versions of AppBuilder, with some instances, like when a rule contains PROCs, it was not possible to generate stateless rules. Now stateless generation supports user defined procedures.

Selection of which generation mechanism is used is made using the HPS.INI setting:

```

[CodegenParameters]
GENERATE_STATELESS_RULE=YES/NO

```

A value of YES specifies to generate stateless classes where possible, while NO, the default, specifies to generate stateful classes. Stateless class generation can also be enabled with the STLS codegen flag.

### Expanded Signature Generation

Another optimization available is generating rules with expanded signature. As stated above, AppBuilder has generated a java class for every distinct view. With the expanded signature feature on, rule input and output views, classes are not generated, but these views, fields are instead passed to a rule class execute method as parameters. This feature is designed to reduce the number of generated java classes and thus improve JVM performance.

Expanded signature generation is available for server rules only and implies stateless rule generation. In order to enable this setting, right-click the partition object, choose **Configuration**, expand the **CodegenParameters** node in the dialog that appears and set EXPAND\_RULE\_SIGNATURE to **YES**. Setting this parameter to NO disables the expanded signature option.

There are some restrictions on using this feature:

- Summary number of input and output views fields should not exceed MAX\_SUMMARY\_IO\_VIEW\_FIELDS Hps.ini parameter value. The default value of this parameter is 255. This is the maximum possible value reflecting the maximum number of method parameters allowed in Java. If the total number of input and output views, fields exceeds this limit, the rule is generated with non-expanded signature.
- Frontier rules (i.e. root rules which are called from outside the partition) are always generated with a non-expanded signature.
- If a rule input or output view is also used as a work view for some rules in the partition or there is a local view declared using a LIKE clause, then the view class is generated anyway, so the benefit of class number reduction is lost for this particular view.
- If a rule input or output view is used in the user procedure body, then all fields of this view are passed to the procedure as separate parameters and this may result in the invalid Java code because of exceeding the Java method parameters number limit (255). Currently generation submits an error in this situation. The user response is to change rule code to redesign the procedure with a large parameters number.
- If LOC function is applied to a rule input or output view, a Java class for the view is generated.

### Lazy Data Instantiation

This optimization is available for rules which have no window or report objects attached. Traditionally, when rule execution started, it performed the creation and initialization of all rule data items and corresponding Java objects living in JVM's memory, until the end of rule code execution. The Lazy Data Instantiation feature introduces a means for more efficient memory management.

With this option, Java code for a rule's scope data object creation and initialization is generated just before the first usage of this object. Also code generation detects a code point after which the data object is not used any more. At this point data is released, i.e. the corresponding Java variable is set to null and the object itself becomes available for garbage collection.

In order to enable Lazy Data Instantiation feature for the partition, right-click the partition object, choose **Configuration**, expand the **CodegenParameters** node, and set LAZY\_INSTANTIATION\_ENABLED to **YES**. Setting the parameter to NO disables the lazy instantiation option for the partition.

The feature can also be enabled or disabled for a particular rule by specifying PRAGMA PARAMETER (LAZY\_INSTANTIATION\_ENABLED, 'YES') or PRAGMA PARAMETER (LAZY\_INSTANTIATION\_ENABLED, 'NO') in the rule code.

## Other Considerations for Rules

Here are other considerations about using a rule:

- [Modeless Windows](#)
- [Execution Environment](#)
- [Security](#)

### Modeless Windows

AppBuilder supports modeless applications in C or Java, where you can work with multiple windows within an application concurrently. The DETACH qualifier on the USE RULE statement, denotes that the rule will be executed asynchronously to its caller. Control returns to the caller immediately, allowing it to continue processing while the detached rule executes. The detached rule, or its children, can display windows that the user can interact with concurrently with the current window displayed by the caller. Both local and remote rules can be detached. Upon termination, an event is triggered on the calling rule to signify that a detached rule has terminated.

In Java, messages can be posted between concurrently executing rules, using the postToChild and postToParent ObjectSpeak methods or equivalent system components. Synchronizing data access is only an issue with the use of [Global View](#).

### Execution Environment

The execution environment of the application affects the operation of the rule. There are different considerations when executing your application depending on the execution environment you choose. Refer to the *Deploying Applications Guide* for information about preparing the rule for a specific execution environment.

Refer to the *Debugging Applications Guide* for information about how to debug your application and troubleshoot any problems that may be related to the execution environment.



## Security

AppBuilder gives you the ability to implement security on an enterprise-wide application in terms of authentication, authorization and encryption. For more information about the mechanisms for security, refer to the *Configuring Communications Guide*. For information about the query authentication method, refer to the *ObjectSpeak Reference Guide*.

## Creating Event-Driven and Converse Applications

You can build event-driven applications in the AppBuilder environment. An event triggers a message that a rule can capture to determine what to do next. This message can be a request to perform some action or a notice that some action has been completed. Sending such a message is called posting an event. Event-driven processing uses posted events to control the flow of an application. There are three different ways to write event-driven rules:

- [Using CONVERSE Statement \(C Client\) Rules](#)
- [Using Event-Driven \(Java\) Rules](#)
- [Using Subscription \(POST Statement\)](#)

This section discusses interface, user and global events, setting up hierarchies for event processing, providing details about [Using the System View](#) in event-driven rules, and provides an [Event-Driven and Converse Processing Example](#).

Event-driven processing generates different types of events, depending on where you use it. The following are possibilities:

*Within a rule:* Any end-user action in a window, such as selecting a menu choice or clicking a push button, generates an interface event that is sent back to the rule that converses the window.

*Between rules within an application:* Event-driven processing is done by posting messages between a detached child rule and its parent rule using the component (HPS\_EVENT\_POST\_TO\_CHILD or HPS\_EVENT\_POST\_TO\_PARENT) or by using the rule Post() method (for Java only). A rule can detach a secondary — or child — rule, which then executes independently of the primary — or parent — rule process.

*Between applications:* Use subscription to pass messages between separate applications. The AppBuilder environment implements subscription to post and receive events between two applications on the same or different machines. Subscription on the same machine is called local eventing. Subscription on physically different machines is called global eventing.



The AppBuilder environment generates *system events* internally to applications for actions such as timing out or closing a window.

Refer to the *Rules Language Reference Guide* for detailed information about the statements used to accomplish event-driven processing. Refer to [Events Supported in C](#) for events used in event-driven processing of rules in C. For information about events in Java, refer to the *ObjectSpeak Reference Guide*.

## Using Converse Statement (C Client) Rules

Use a CONVERSE Rules statement to either display a window or block a rule until it receives an event. This is necessary for C language applications. The CONVERSE statement is not available in Java. Refer to [Converse Events for Java](#). In event processing, a CONVERSE statement on a client either displays a window or blocks a rule until it receives an event. The CONVERSE statement takes two forms for the respective purposes:

- [See CONVERSE WINDOW](#)
- [See CONVERSE \(null\)](#) – without any arguments, called a null statement

See [Using Converse Rules \(CASEOF Statements\)](#) to create a Converse rule.

### CONVERSE WINDOW

This form of the CONVERSE statement displays the panel of the named window on the screen and lets end users manipulate its control objects (such as menus and buttons) and enter data into the fields of the window. The rules code execution remains on the CONVERSE WINDOW statement until an event is returned to the rule. That is, the CONVERSE WINDOW statement waits for an event, and when it receives one, continues executing the next statement in the rule.

A user manipulating a control object of a window generates an interface event, which returns control to the rule. A system event, global event, or an event from a parent or child rule also causes a rule to leave its waiting state.

### CONVERSE (null)

A CONVERSE statement without any arguments blocks a rule from executing until it receives an event. That is, the rule remains in a wait state until an event triggers execution of the statements after the null CONVERSE.

Use the null CONVERSE for eventing to pass messages among rules on the same or different systems. When one rule that includes an event in its hierarchy posts a message, any rule with the same event in its hierarchy receives and can process the message. Hence the CONVERSE statement blocks the rule until it receives an event, placing them inside a DO loop (or converse loop), process the events on receiving results in event-driven processing.

## Using Converse Rules (CASEOF Statements)

Use nested case statements (CASEOF in Rules) to check for the type of the event in the outermost statement, for the names of specific events in the next level, and for the specific sources in the innermost statement. You can create Converse rules any of the following ways:

- [Using CASEOF Statements to Receive Events](#)
- [Additional Steps for Modeless Secondary Windows](#)

### Using CASEOF Statements to Receive Events

There are several ways to code rules to receive interface events. The easiest way to do this, to handle multiple types of interface events, is with a set of nested CASEOF statements as follows:

#### *Check for the type of event in the outermost CASEOF statement.*

1. Check for the names of specific events in the next-level CASEOF statement.
2. Check for specific sources (the System ID of the object that generates the event) in the innermost CASEOF statement.
3. Place the CASE construct immediately after a CONVERSE WINDOW statement within a DO loop.
4. By controlling the WHILE condition, you can exit from the converse loop.

In the following example, the loop will exit when the SystemID of event source becomes EXIT.

The resulting code for capturing an interface event might look similar to the following example for a C language application. For information about the pre-defined, system view HPS\_EVENT\_VIEW, refer to [Using the System View](#).

#### *Example: Using CASEOF Statements*

```
do while EVENT_SOURCE of HPS_EVENT_VIEW <> 'EXIT'
  converse WINDOW window_name
  caseof EVENT_TYPE of HPS_EVENT_VIEW
    case INTERFACE_EVENT
      caseof EVENT_NAME of HPS_EVENT_VIEW
        case 'HPS_PB_CLICK'
          caseof EVENT_SOURCE of HPS_EVENT_VIEW
            case 'Push button 1'
              ..other statements...
            case 'Push button 2'
              ..other statements...
          endcase
        case 'HPS_IMMEDIATE_RETURN'
          caseof EVENT_SOURCE of HPS_EVENT_VIEW
            case 'Radio button 1'
              ..other statements...
            case 'Radio button 2'
              ..other statements...
          endcase
        endcase
      case USER_EVENT
        ..other statements...
      endcase
    enddo
```

By simply changing line 4 from CASE INTERFACE\_EVENT to CASE SYSTEM\_EVENT, you can use the same code to capture system events. See [Event-Driven and Converse Processing Example](#) for a complete example of event-driven processing.

### Additional Steps for Modeless Secondary Windows

Capturing user events to implement modeless secondary windows requires more steps than are needed to capture interface or system events. The following are the additional steps needed to capture user events to implement modeless secondary windows:

1. [Detaching a Process](#)
2. [Communicating User Events](#)
3. [Posting the Events Between Detached Rules](#)

## Detaching a Process

If you want to spawn a modeless secondary window or an independently executing process, insert a USE RULE...DETACH statement in the primary (parent) rule. You can use this statement with or without an INSTANCE clause:

With an INSTANCE clause: *To detach multiple instances of the same rule, use an INSTANCE clause as follows:*

```
use RULE rule_name_ _DETACH INSTANCE instance_name_
```

*To post an event to a specific instance, you must provide the instance name in the EVENT\_QUALIFIER field of the HPS\_EVENT\_POST\_TO\_CHILD component input view.*

Without an INSTANCE clause: *If you omit the INSTANCE clause, you can detach the named child rule only once.*

Detached rules can retrieve information from databases. A parent rule can have up to four levels of detached rules under it, all of which can access a server or the host. Refer to the *Rules Language Reference Guide* for more information about how to use a USE RULE...DETACH statement. Refer to the *System Components Reference Guide* for details on the system components.

Each detached rule starts a separate process. Any nested rule is part of the executable process of the parent. This distinction is important because events are posted from one executable process to another, not from one rule to another. The top rule in each executable process receives the event. This can cause confusion when posting an event from a detached child (secondary) rule back to a parent (primary) rule that is itself nested.

For example, assume root rule RR\_1 detaches child rule DR\_1. This creates two executable processes, and if DR\_1 posts an event to its parent, RR\_1 receives it. Similarly, assume that DR\_1 now detaches child rule DR\_2. This creates a third process, and if DR\_2 posts an event to its parent, DR\_1 receives it. DR\_1 can still post to its parent, RR\_1.

However, assume a different hierarchy where RR\_1 calls a nested rule NR\_1, and NR\_1 detaches rule DR\_1. This creates only two executable processes, because NR\_1 is part of RR\_1 process. In this case if DR\_1 posts an event to its parent, the top rule in the first executable process, RR\_1, and not NR\_1, receives the event.

## Communicating User Events

The HPS\_EVENT\_POST\_TO\_CHILD and HPS\_EVENT\_POST\_TO\_PARENT system components enable a detached child rule and its parent process to pass events to each other. A receiving rule always receives an event from one of these two components as a user event.

A work view that you define transmits data between parent and child executable processes. The input views of both of the posting components contain an EVENT\_VIEW field. If this field specifies a work view attached to the sending rule, any data within it is passed to a similarly named work view attached to the receiving rule. Thus, you must attach this view as a work view to the hierarchies of both rules and define the fields it contains.

You cannot use the predefined system view HPS\_EVENT\_VIEW itself as this work view, because the system overwrites any data you map to it. Before calling one of the components, map the name of this view to the component EVENT\_VIEW field. Calling the component places the name of the view into the EVENT\_VIEW field of HPS\_EVENT\_VIEW. For information about the pre-defined, system view HPS\_EVENT\_VIEW, refer to [Using the System View](#).

## Posting the Events Between Detached Rules

Use the following system components to post events between detached rules:

- [HPS\\_EVENT\\_POST\\_TO\\_CHILD](#)
- [HPS\\_EVENT\\_POST\\_TO\\_PARENT](#)

Refer to the *System Components Reference Guide* for details on these system components and their input and output fields.

### HPS\_EVENT\_POST\_TO\_CHILD

This system component posts an event from a parent (primary) rule to a detached rule. Its input view has the following fields:

- EVENT\_NAME contains the name of the event, which should be meaningful to the child rule and recognizable in the rules code.
- EVENT\_DEST contains the name of the rule that is to receive the event.
- EVENT\_QUALIFIER specifies the instance name of the rule that is to receive the event. The field is empty if the child is not detached as an instance.

- EVENT\_VIEW contains the name of a work view that contains data you want to send to the child rule.
- EVENT\_PARAM sends any additional information about the event.

### HPS\_EVENT\_POST\_TO\_PARENT

This system component sends an event to the top rule in the executable process of its parent. Its input view has the following fields:

- EVENT\_NAME is the name of the event that you want to send.
- EVENT\_VIEW is the name of a work view that contains data you want to send to the parent rule.
- EVENT\_PARAM sends any additional information about the event.

## Using Event-Driven (Java) Rules

Write an event-handler procedure for an object to have the rules register for an event. When an event occurs, the procedure is called. Every Java rule generates an initialize event when the rule is being initialized and generates a terminate event when the rule is being terminated. This is in addition to any user interface events such as mouse clicks and double-clicks.

Follow the steps below to write an event-driven (Java) rule:

1. Write an initialize event procedure where the program data and local variables, if any, are initialized.
2. Write a terminate event procedure where the commit database and cleanup of the program can be handled.
3. Write any other event handler procedure that needs to be handled for any object. The term *object* here includes controls in a window, detached rules, etc., for example, a 'Click' event handler procedure for a push button with System ID 'EXIT'.

## Using Subscription (POST Statement)

There are two types of subscription. Subscription on the same machine does not need a network and is called local eventing. Subscription on physically different machines employ a client/server network and is called global eventing. Global eventing is available for C applications that communicate with a server via LU6.2 only. Local eventing is available for Java thick clients and C applications. AppBuilder applications can subscribe for an event that can be fired from the same or a different application; the event can also be from the same machine or a remote machine. All the subscribing applications use the Physical Event entity within AppBuilder to register for an event. When an event is fired using the Physical Event entity, it is delivered to all the applications that registered for this event. Eventing is a flexible approach to asynchronous messaging whereby:

A client or service makes data available to the system in response to periodic runtime events.

The system makes that data available to another program for as long as the program subscribes to the data.

Once your program posts the event to the network, the system informs rules registered for the event that the data is available to be retrieved.

After the rule receives and processes the data, it is free to resume normal functioning. Rules de-register for events on exit. Each AppBuilder site defines the event itself.

You can use subscription to pass events between different applications. Whenever a rule with a particular physical event in its hierarchy posts a message, any rule with the same event in its hierarchy can receive and process the message. Use POST Rules Language statements to post a message to another application or to a different rule in the same application. The view attached to the event contains the text of the message.

This topic explains:

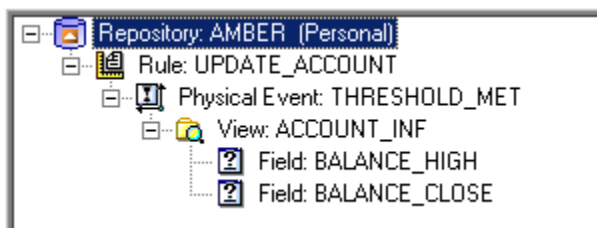
- [Setting up a Posting Rule Hierarchy](#)
- [Setting up a Receiving Rule Hierarchy](#)
- [Posting in Java](#)

Refer to the *Configuring Communications Guide* for information about configuring machines and servers on other machines.

### Setting up a Posting Rule Hierarchy

[Posting rule hierarchy](#) shows the hierarchy for the posting rule.

#### Posting rule hierarchy

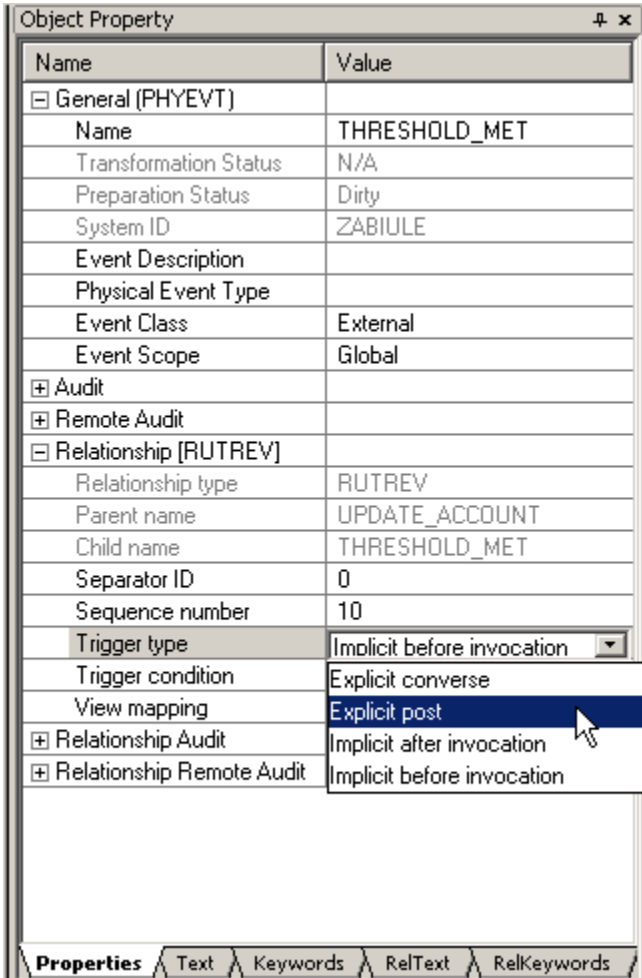


Follow these steps to set up the hierarchy:

Right-click the Rule in the hierarchy, and click *Insert > Physical Event* . Create and name the event.

1. Right-click the Physical Event, and select *Properties* from the pop-up menu.
2. In the *Properties* window, change the *Trigger Type* field to *Explicit post* and Commit the change.
3. Add a View as a child of the Physical Event. Set the *View usage* properties of the view to *Work View* .
4. Attach Fields to the work view corresponding to the data you want to pass as the message.

**Properties for Physical Event**



**Relationship Properties for a Physical Event**

Property	Description
Parent name	The name of the parent Rule
Child name	The name of the Physical Event
Sequence number	A number automatically assigned to each entity in a sequence. It is used to define the order of relationships attached to a common entity.
Separator ID	For internal purposes. Uniquely distinguishes duplicate relationships between the same parent and child entities.
Trigger type	<ul style="list-style-type: none"> <li>• Explicit Converse</li> <li>• Explicit Post</li> <li>• Implicit after invocation</li> <li>• Implicit before invocation</li> </ul>

The *Implicit after invocation* and *Implicit before invocation* properties on this relationship (Rule triggers Event) refer to [Implicit Eventing](#).

## Implicit Eventing

Where Explicit events are defined *within* the hierarchy, Implicit events are defined externally. Implicit eventing uses three types of tables:

1. A subcell table to identify the routes to the children of the local host.
2. A trigger table to identify the triggering service.
3. An event table to identify the triggered service, message, or event.

The subcell, trigger and event tables are located in AppBuilder\NT\RT.



Implicit events are *only* supported by C client and server.

Implicit eventing is a type of asynchronous processing in which eventing logic resides in text files rather than application code. Through this method, it allows you address temporary requirements of your application.

For example, suppose you wanted to monitor the activity of remote services closely in the initial stages of implementing a system, but monitor them intermittently after that – in response to unusual network demand. Or suppose you wanted to be able to react to a period of market instability by specially logging all trades affecting a particular stock. In both of these situations, you would not want to imbed a function into your application that will eventually become superfluous. By maintaining eventing logic in text files rather than program code, implicit eventing addresses these temporary requirements.

### Example: Posting Rule Code

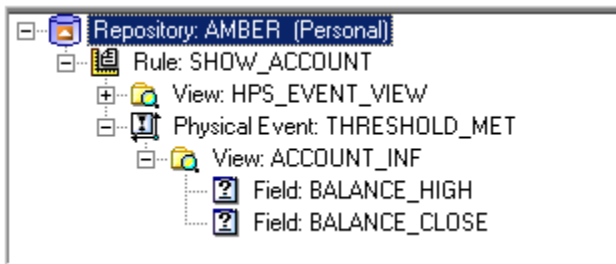
Map data to the Work View and use the *POST EVENT* statement in the posting rule to send the message:

```
map BALANCE_HIGH of BALANCE_QUERY_I to BALANCE_HIGH
  of ACCOUNT_INFO of THRESHOLD_MET
map BALANCE_CLOSE of BALANCE_QUERY_I to BALANCE_CLOSE
  of ACCOUNT_INFO of THRESHOLD_MET
post event THRESHOLD_MET
```

## Setting up a Receiving Rule Hierarchy

[Receiving rule hierarchy](#) shows the hierarchy for the receiving rule.

### Receiving rule hierarchy



Follow these steps to set up the hierarchy:

1. Right-click the Rule in the hierarchy, and click *Insert > Physical Event* . Name the event.
2. Right-click the Physical Event, and select *Relationship Properties* . Refer to [Relationship Properties for a Physical Event](#) for information about each relationship property.
3. In the *Properties – Rule triggers Event* window, change the *Trigger Type* field to *Explicit converse* and click *OK* .
4. Attach a View to the Rule and name it HPS\_EVENT\_VIEW (the predefined system view).

For information about the pre-defined, system view HPS\_EVENT\_VIEW, refer to [Using the System View](#).

### Receiving Rule Code

Use the predefined system view HPS\_EVENT\_VIEW in the receiving rule to capture the message, just as you would for an event internal to the application. Check for the event using a standard CASEOF statement. An event has an EVENT\_NAME of whatever name you gave the physical event. The other HPS\_EVENT\_VIEW fields are empty. For information about the pre-defined, system view HPS\_EVENT\_VIEW, refer to [Using the System View](#).

To handle multiple global events, include the CONVERSE statement inside a loop. You can use any condition to terminate the loop, including another event, as the following example illustrates. The system registers the rule for the event when the rule is started and deregisters it on exit.

## Example: Handling Multiple Events

```
do while EVENT_NAME of HPS_EVENT_VIEW <> 'APPLICATION_CLOSED'  
  converse  
  caseof EVENT_NAME of HPS_EVENT_VIEW  
  case 'THRESHHOLD_MET'  
    ...statements  
  endcase  
enddo
```



Although the example above uses a null *CONVERSE* statement, a *CONVERSE WINDOW* statement is also unblocked when it receives an event. That is, the statements following the *CONVERSE WINDOW* statement begin executing when an event is received.

## Posting in Java

The Java client environment includes support for local eventing within a Java virtual machine (JVM). All applications running within the same JVM can publish and subscribe for events.

Java uses the same syntax to post an event as described in [Setting up a Posting Rule Hierarchy](#). The following call sends the message to all the subscribed applications:

```
post event THRESHHOLD_MET
```

And alternatively, an ObjectSpeak method post could be used on the GlobalEvent object by:

```
THRESHHOLD_MET.post()
```

And on the subscribing end, the application can use either an event procedure or a Converse Event to handle the event. If the pre-defined system view, HPS\_EVENT\_VIEW, is attached to the subscribing rule, an event listener is automatically added to the rule and a ConverseEvent is triggered on receiving this event. The EVENT\_NAME of the HPS\_EVENT\_VIEW will have the name of the event.

For information about ObjectSpeak syntax, refer to the *ObjectSpeak Reference Guide*. For information about the pre-defined system view, HPS\_EVENT\_VIEW, refer to [Using the System View](#).

Alternatively, an event procedure can be used to process the event. It can be defined explicitly for a particular physical event.

## Examples: Posting in Java

### Example 1: Specific Event

```
proc p for post object THRESHHOLD_MET  
  ( o object type PostEvent )  
  ...statements  
endproc
```

The GlobalEvent type can be used in the event procedure definition to handle all the physical events defined under that rule.

### Example 2: Global Event

```
proc p for post object type GlobalEvent  
  ( o object type PostEvent )  
  ...statements  
endproc
```

The event procedure also can be defined for certain selected physical events defined under the rule.

### Example 3: Selected Event

```

dcl
//forward declaration
p proc for post object THRESHOLD_MET
( o object type PostEvent );
enddcl
proc p
...statements
endproc

```

## Using the System View

The predefined system view, HPS\_EVENT\_VIEW, is a work view for building an application to handle events. This view implements event-driven processing by capturing information about an event, including its type, name, and source.

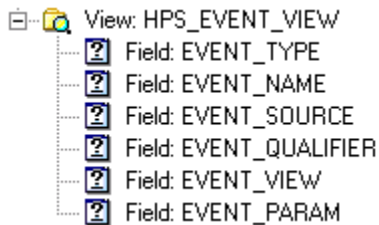
This topic includes:

- [Understanding the Fields of the System View](#)
- [Attaching the System View to a Rule Hierarchy](#)

### Understanding the Fields of the System View

The hierarchy of the predefined system view, HPS\_EVENT\_VIEW, is shown in [System view hierarchy](#).

#### System view hierarchy



This view contains the fields that describe an event and these are summarized in [Fields that describe an event](#).

#### Fields that describe an event

Field Name	Description
EVENT_TYPE	<p>The EVENT_TYPE is a SMALLINT (small integer) field denoting the type of event generated. The types of events to which the SMALLINT values in the HPS_EVENT_TYPE set correspond are:</p> <ul style="list-style-type: none"> <li>• 0 - System event</li> <li>• 1 - Interface event</li> <li>• 2 - User event</li> </ul>
EVENT_NAME	<p>The EVENT_NAME field contains the event name. Interface event names describe the action taken. For example, the EVENT_NAME field is HPS_PB_CLICK if a user clicks a push button. System events have a similar nomenclature.</p>
EVENT_SOURCE	<p>The EVENT_SOURCE field contains values for interface and user events:</p> <p>For interface events, EVENT_SOURCE contains the system identifier (HPSID) of the window object that is the event focus. For example, if a user clicks a push button, the EVENT_SOURCE field contains the push button HPSID. The system identifier (HPSID) is case sensitive. The Rules Language code for checking the EVENT_SOURCE must match the system identifier (HPSID) exactly as it is entered in Window Painter for the window object.</p> <p>For user events, the EVENT_SOURCE field contains the name of the child (secondary) rule that sent the event. For example, if a child rule uses the HPS_EVENT_POST_TO_PARENT component to send an event to its parent, EVENT_SOURCE contains the name of the child rule.</p> <p>The EVENT_SOURCE field is empty when a parent posts an event to a child:</p> <ul style="list-style-type: none"> <li>• For system events</li> <li>• For global events</li> </ul>



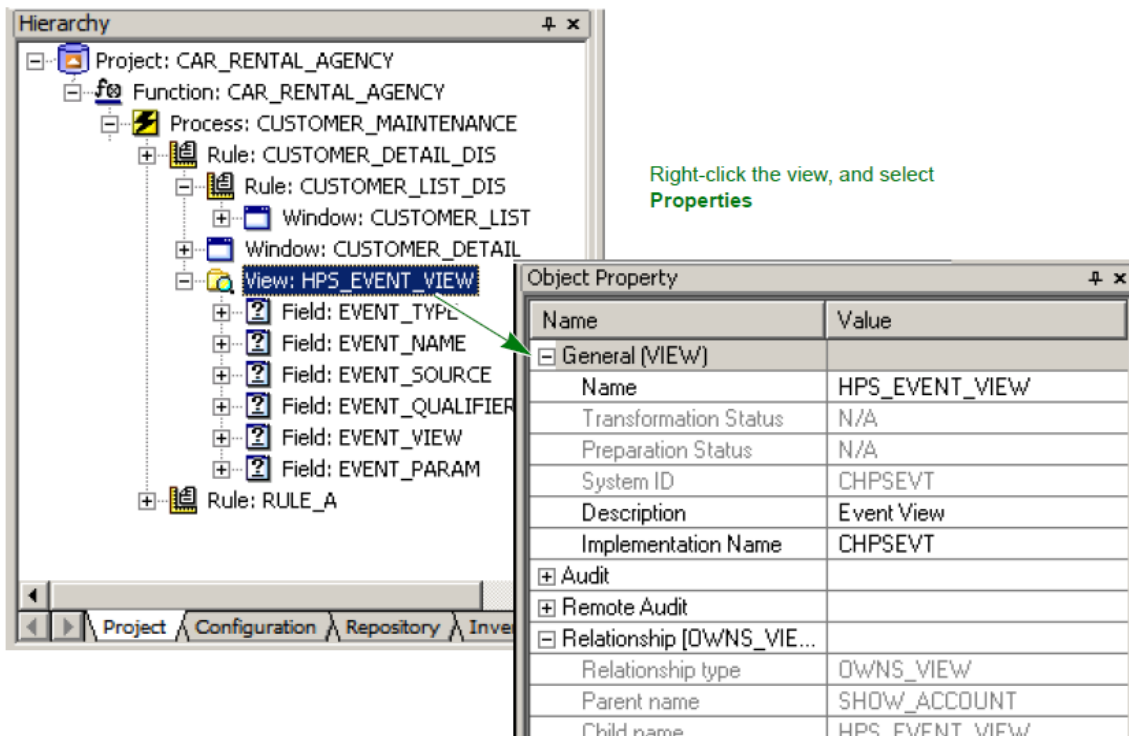
EVENT_QUALIFIER	The EVENT_QUALIFIER field qualifies the event source when a parent posts a user event to a child. This field contains the instance of the rule (if applicable) specified in the EVENT_SOURCE field. Thus, this field is empty when a parent posts an event to a child detached without an instance name.
EVENT_VIEW	The EVENT_VIEW field contains the name of the view that received data for user events only.
EVENT_PARAM	The EVENT_PARAM field contains any additional description of the event, which can be useful if you have detached multiple instances of the same rule.

### Attaching the System View to a Rule Hierarchy

You must attach the predefined system view HPS\_EVENT\_VIEW to the correct place in your application hierarchy. Attach the view as a child of any rule that receives events. For example, attach this view to any rule that converses a window. When the HPS\_EVENT\_VIEW is added as a child of a window, it sets the usage to *Work View* for you.

[System view in hierarchy and relationship properties](#) illustrates the correct placing of this view in the hierarchy and opening the relationship properties window to set the view usage.

#### System view in hierarchy and relationship properties



## Event-Driven and Converse Processing Example

This section provides examples of Event-Driven Processing in C, Java, and a mixed mode:

- [Event-Driven Processing Examples in C](#)
- [Java Examples](#)
- [Mixed Mode Example](#)

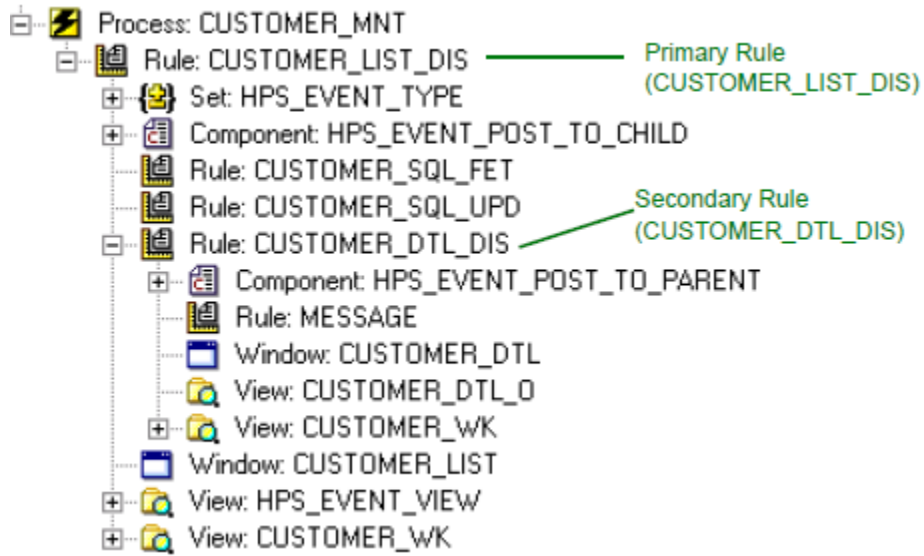
### Event-Driven Processing Examples in C

In the hierarchy in [Hierarchy of example](#), the CUSTOMER\_LIST rule is the primary rule or parent rule. Its window contains a list box of current customers. Each time a user selects a customer, either by double-clicking a row in the list box or by clicking a Select push button, the CUSTOMER\_LIST rule detaches an instance of the CUSTOMER\_DTL rule for that particular customer.

This makes the CUSTOMER\_DTL rule the secondary or detached rule to the CUSTOMER\_LIST rule, so several different customer detail windows or instances can be displayed at the same time. The CUSTOMER\_WK view transmits data between the parent rule, CUSTOMER\_LIST, and the child executable process, CUSTOMER\_DTL. The CUSTOMER\_DTL window contains a series of editable fields. Users can change the fields with an Update menu choice. A nested, modal window (not shown here) displays the results of the update action.

[Hierarchy of example](#) shows the hierarchy for the primary rule in the example program. The Rules code for the primary (parent) is listed and partially explained below. A partial explanation of the code for the secondary (child) rules follows the primary rule. This is only an example of how to structure your application hierarchy; the full hierarchy is not provided.

**Hierarchy of example**



[Examples: Event-Driven Processing in C](#)

*Example 1: Primary Rule*

The following example illustrates Rules Language code for the Primary rule. This is an example of code using the Customer\_List rule.

```

1 //CUSTOMER_LIST Rule
1 dcl
2   TEMP_PIC_FIELD PIC'999';
3   TEMP_PIC_VIEW VIEW CONTAINS temp_pic_field;
4   TEMP_INDEX INTEGER;
5 enddcl
6 do while EVENT_SOURCE of HPS_EVENT_VIEW <> 'EXIT'
7   use RULE CUSTOMER_SQL_FET
8   map CUSTOMER_OCC of CUSTOMER_SQL_FET_O to CUSTOMER_OCC
9     of CUSTOMER_LIST_W
10  converse WINDOW CUSTOMER_LIST
11  caseof EVENT_TYPE of HPS_EVENT_VIEW
12    case INTERFACE_EVENT
13      caseof EVENT_NAME of HPS_EVENT_VIEW
14        case 'HPS_IMMEDIATE_RETURN'
15          MAP int(EVENT_PARAM of HPS_EVENT_VIEW) to
16            TEMP_INDEX
17          // this line is empty
18          map CUSTOMER_ID of CUSTOMER_OCC of
19            CUSTOMER_LIST_W(TEMP_INDEX) to
20              CUSTOMER_ID of CUSTOMER_SQL_SEL_I
21          use RULE CUSTOMER_SQL_SEL
22          map CUSTOMER of CUSTOMER_SQL_O to
23            CUSTOMER of CUSTOMER_DTL_I
24          use RULE CUSTOMER_DTL DETACH INSTANCE
25        endcase
26      case USER_EVENT
27        caseof EVENT_NAME of HPS_EVENT_VIEW
28          case 'UPDATE CUSTOMER RECORD'
29            map CUSTOMER of CUSTOMER_WK to CUSTOMER of
30              CUSTOMER_SQL_UPD_I
31            use RULE CUSTOMER_SQL_UPD
32            map 'UPDATE RESULTS' to EVENT_NAME of
33              HPS_EVENT_POST_TO_CHILD_I
34            map 'CUSTOMER_DTL' to EVENT_DEST of
35              HPS_EVENT_POST_TO_CHILD_I
36            map CUSTOMER_ID of CUSTOMER of CUSTOMER_WK
37              to EVENT_QUALIFIER of
38                HPS_EVENT_POST_TO_CHILD_I
39            if SQL_RETURN_CODE of CUSTOMER_SQL_UPD_O = 0
40              map 'SUCCESSFUL' to EVENT_PARAM of
41                HPS_EVENT_POST_TO_CHILD_I
42            else
43              map 'UNSUCCESSFUL' to EVENT_PARAM of
44                HPS_EVENT_POST_TO_CHILD_I
45            endif
46            use COMPONENT HPS_EVENT_POST_TO_CHILD
47          endcase
48        endcase
49 enddo

```

#### Explanation of Primary Rule example

The primary rule captures interface and user events by checking almost every field in the predefined system view HPS\_EVENT\_VIEW with a series of nested CASEOF statements (refer to lines 11 – 14 and lines 22 – 24):

```

11 caseof EVENT_TYPE of HPS_EVENT_VIEW
12   case INTERFACE_EVENT
13     caseof EVENT_NAME of HPS_EVENT_VIEW
14       case 'HPS_IMMEDIATE_RETURN'
15         ...other statements...
16       ..other statements...
22   case USER_EVENT
23     caseof EVENT_NAME of HPS_EVENT_VIEW
24       case 'UPDATE CUSTOMER RECORD'
25         ...other statements...

```

(You can reduce the number of nested CASEOF statements by checking only the EVENT\_SOURCE and EVENT\_NAME fields.)

When the primary (parent) rule receives a user event, it copies the data in the specified child work view to the parent work view of the same name (line 25).

```
25 map CUSTOMER of CUSTOMER_WK to CUSTOMER of CUSTOMER_SQL_UPD_I
```

The primary rule then calls the appropriate SQL update rule (line 26).

```
26 use RULE CUSTOMER_SQL_UPD
```

The CUSTOMER\_ID of the CUSTOMER work view that was just received is used to determine which child the event is sent to (lines 27 through 29). After the update is completed, the HPS\_EVENT\_POST\_TO\_CHILD component issues an event to notify the child that sent the update event of the results (line 27).

```
27 map 'UPDATE RESULTS' to EVENT_NAME of HPS_EVENT_POST_TO_CHILD_I
```

To avoid adding another work view to the hierarchy, you can convert the return code sent to the child to a CHAR format and place it in the EVENT\_PARAM field of the component input view.

#### Example 2: Secondary Rule

The following example illustrates Rules Language code for the secondary (child) rule. Refer to [Hierarchy of example](#).

```
//CUSTOMER_DTL_DIS Rule
map CUSTOMER of CUSTOMER_DTL_I to CUSTOMER of CUSTOMER_DTL_W
do while EVENT_SOURCE of HPS_EVENT_VIEW <> 'EXIT'
  converse WINDOW CUSTOMER_DTL
  caseof EVENT_NAME of HPS_EVENT_VIEW
    case 'HPS_MENU_SELECT'
      caseof EVENT_SOURCE of HPS_EVENT_VIEW
        case 'UPDATE'
          map CUSTOMER of CUSTOMER_DTL_W to CUSTOMER
            of CUSTOMER_WK
          map 'UPDATE CUSTOMER RECORD' to EVENT_NAME of
            HPS_EVENT_POST_TO_PARENT_I
          map 'CUSTOMER_WK' to EVENT_VIEW of
            HPS_EVENT_POST_TO_PARENT_I
          use COMPONENT HPS_EVENT_POST_TO_PARENT
        endcase
      case 'UPDATE RESULTS'
        map 'The update was' ++ EVENT_PARAM of
          HPS_EVENT_VIEW ++ ' ' to TEST1 of MESSAGE_I
        use RULE MESSAGE_NEST
      endcase
  enddo
```

#### Explanation of Secondary Rule example

The secondary (child) CUSTOMER\_DTL\_DIS rule posts the user event, UPDATE CUSTOMER RECORD. The record to be updated is sent to the parent using the CUSTOMER\_WK work view (lines 5 to 18).

When a user selects the *Update* menu choice, the child rule maps the altered data to the work view (line 9) and then puts the work view name in the EVENT\_VIEW field of the HPS\_EVENT\_POST\_TO\_PARENT component input view (lines 10 and 11).

```
9 map CUSTOMER of CUSTOMER_DTL_W to CUSTOMER
  of CUSTOMER_WK
10 map 'UPDATE CUSTOMER RECORD' to EVENT_NAME of
  HPS_EVENT_POST_TO_PARENT_I
11 map 'CUSTOMER_WK' to EVENT_VIEW of
  HPS_EVENT_POST_TO_PARENT_I
```

When the child rule receives the UPDATE RESULTS event, the EVENT\_PARAM field is tested and a nested rule displays an appropriate message (lines 14 – 16).

```
14 case 'UPDATE RESULTS'  
15 map 'The update was' ++ EVENT_PARAM of  
    HPS_EVENT_VIEW ++ '.' to TEST1 of MESSAGE_I  
16 use RULE MESSAGE NEST
```

## Java Examples

The following Java examples are associated with the hierarchy that is shown in [Hierarchy of example](#). The following examples illustrates Rules Language code for the Primary rule and the Secondary rule.

### [Example: Java Code](#)

*Example 1: Primary Rule*

```

//CUSTOMER_LIST_DIS Rule
dcl
    tempstr char(20);
enddcl
//This procedure fetches data from the database and fills the list
//box with the data.
proc FillCustomerList
    use rule CUSTOMER_SQL_FETCH
    map CUSTOMER_OCC of CUSTOMER_SQL_FET_O
    to CUSTOMER_OCC of CUSTOMER_LIST_W
endproc
proc for Initialize type window(ie object pointer to InitializeEvent)
    FillCutomerList
endproc
proc for Click object EXIT(ce object pointer to ClickEvent)
    thisRule.terminate()
endproc
//The following procedure get called when double-clicked on a row of
//the multi column list
proc for DoubleClick type Table(dce object pointer to
DoubleClickEvent)
    map CUSTOMER_ID of CUSTOMER_OCC of
        CUSTOMER_LIST_W(dce.PhysicalIndex) to
        CUSTOMER_ID of CUSTOMER_SQL_SEL_I
    use RULE CUSTOMER_SQL_SEL
    map CUSTOMER of CUSTOMER_SQL_O to
        CUSTOMER of CUSTOMER_DTL_I
    use RULE CUSTOMER_DTL DETACH INSTANCE
endproc
//This procedure get called when a message received from the child
//rule
proc for Post object type rule(pe object pointer to PostEvent)
    if pe.Subject = 'UPDATE CUSTOMER RECORD'
        map CUSTOMER of CUSTOMER_WK to CUSTOMER of CUSTOMER_SQL_UPD_I
        use RULE CUSTOMER_SQL_UPD

        if SQL_RETURN_CODE of CUSTOMER_SQL_UPD_O = 0
            map 'SUCCESSFUL' to tempstr
        else
            map 'UNSUCCESSFUL' to tempstr
        endif
        //post the message "UPDATE RESULT" to the child rule
        //CUSTOMER_DTL_DIS
        thisRule.post('CUSTOMER_DTL_DIS', 'UPDATE RESULTS',
            CUSTOMER_WK, tempstr)
        FillCustomerList
    endif
endproc

```

#### *Example 2: Secondary Rule*

The following example illustrates Rules Code for Customer\_DTL Rule, the Secondary rule. See [Hierarchy of example](#).

```

proc for Initialize type window(ie object pointer to InitializeEvent)
  map CUSTOMER of CUSTOMER_DTL_I to CUSTOMER of CUSTOMER_DTL_W
endproc

proc for Click object EXIT(ce object pointer to ClickEvent)
  thisRule.Terminate()
endproc

proc for Click object UPDATE(ce object pointer to ClickEvent)
  map CUSTOMER of CUSTOMER_DTL_W to CUSTOMER
  of CUSTOMER_WK
  map 'UPDATE CUSTOMER RECORD' to EVENT_NAME of
  HPS_EVENT_POST_TO_PARENT_I
  map 'CUSTOMER_WK' to EVENT_VIEW of
  HPS_EVENT_POST_TO_PARENT_I
  use COMPONENT HPS_EVENT_POST_TO_PARENT
endproc

proc for Post object type rule(pe object pointer to PostEvent)
  if pe.Subject = 'UPDATE RESULTS'
    map 'The update was' ++ EVENT_PARAM of
      HPS_EVENT_VIEW ++ '.' to TEST1 of MESSAGE_I
    use RULE MESSAGE NEST
  endif
endproc

```

## Mixed Mode Example

The following code example shows the same Customer Rule that is shown in the [Java Examples](#). Here, the Customer Rule is converted using TurboScripter to support both Java and C source code in the same rule. Use the macro 'LANGUAGE' to conditionally compile for Java or C.

### [Example: Customer List Rule \(Mixed Mode\)](#)

```

*>CUSTOMER_LIST Rule <*>
dcl
  TEMP_PIC_FIELD PIC'999';
  TEMP_PIC_VIEW VIEW CONTAINS temp_pic_field;
  TEMP_INDEX INTEGER;
enddcl

*> ----- <*>
*> ** Booleans to control display loop - do not change/remove ** <*>
*> ** skipDisp to skip the display of the window ** <*>
*> ** exitLoop for exiting the loop ** <*>
*> ** exitRule for exiting the rule ** <*>
*> ----- <*>

dcl
  skipDisp boolean;
  exitLoop boolean;
  exitRule boolean;
enddcl

*> ----- <*>
*> ** The preLoop is executed once before the loop ** <*>
*> ----- <*>

proc preLoop
endproc *> preLoop <*>

*> ----- <*>
*> ** The preConverse is executed each loop before the converse ** <*>
*> ----- <*>

```

```

proc preConverse
  if not ( EVENT_SOURCE of HPS_EVENT_VIEW <> 'EXIT' )
    map true to exitLoop
    proc return
  endif
  use RULE CUSTOMER_SQL_FET
  map CUSTOMER_OCC of CUSTOMER_SQL_FET_O to CUSTOMER_OCC of
    CUSTOMER_LIST_W
endproc *> preConverse <*>
*> ----- <*>
*> ** The postConverse is executed each loop after the converse ** <*>
*> ----- <*>
proc postConverse
  caseof EVENT_TYPE of HPS_EVENT_VIEW
    case INTERFACE_EVENT
      caseof EVENT_NAME of HPS_EVENT_VIEW
        case 'HPS_IMMEDIATE_RETURN'
          map int(EVENT_PARAM of HPS_EVENT_VIEW) to TEMP_INDEX
          map CUSTOMER_ID of CUSTOMER_OCC of
            CUSTOMER_LIST_W(TEMP_INDEX) to
            CUSTOMER_ID of CUSTOMER_SQL_SEL_I
          use RULE CUSTOMER_SQL_SEL
          map CUSTOMER of CUSTOMER_SQL_O to
            CUSTOMER of CUSTOMER_DTL_I
          use RULE CUSTOMER_DTL DETACH INSTANCE
        endcase
      case USER_EVENT
        caseof EVENT_NAME of HPS_EVENT_VIEW
          case 'UPDATE CUSTOMER RECORD'
            map CUSTOMER of CUSTOMER_WK to CUSTOMER of
              CUSTOMER_SQL_UPD_I
            use RULE CUSTOMER_SQL_UPD
            map 'UPDATE RESULTS' to EVENT_NAME of
              HPS_EVENT_POST_TO_CHILD_I
            map 'CUSTOMER_DTL' to EVENT_DEST of
              HPS_EVENT_POST_TO_CHILD_I
            map CUSTOMER_ID of CUSTOMER of CUSTOMER_WK
              to EVENT_QUALIFIER of HPS_EVENT_POST_TO_CHILD_I
            if SQL_RETURN_CODE of CUSTOMER_SQL_UPD_O = 0
              map 'SUCCESSFUL' to EVENT_PARAM of
                HPS_EVENT_POST_TO_CHILD_I
            else
              map 'UNSUCCESSFUL' to EVENT_PARAM of
                HPS_EVENT_POST_TO_CHILD_I
            endif
            use COMPONENT HPS_EVENT_POST_TO_CHILD
          endcase
        endcase
      endcase
endproc

*> ----- <*>
*> ** The postLoop is executed once after the loop ** <*>
*> ----- <*>
proc postLoop
endproc *> postLoop <*>

*> -----<*>
*> *** THE CODE BELOW THIS COMMENT BOX SHOULD NEVER BE CHANGED ***
<*>
*> *** THE STRUCTURE IS CONSTANT FOR ALL WINDOW DISPLAY RULES ***
*> -----<*>
*> -----<*>

*> To support a 'single source dual runtime solution' the CG_IF ...
<*>
*> is used to enable the generation of both a C and a Java client.
<*>
*> -----<*>
CG_IF(LANGUAGE,C)
*> -----<*>

```



```

*> This block of code is only included by CODEGEN for C compile
<*
*> Note: HPS.INI value [CGen] MACRO=LANGUAGE=C is required
<*
*> -----<*
*> -----<*
*> Initialize logic, before the loop <*
*> -----<*
preLoop
    if exitRule
        return
    endif
*> -----<*
*> DO... WHILE ... converse window loop <*
*> -----<*
do
    preConverse
    if exitRule
        return
    endif
    if exitLoop = false
        if skipDisp = false
            converse window CUSTOMER_LIST
        endif
        postConverse
        if exitRule
            return
        endif
    endif
    while exitLoop = false
enddo *> converse window loop <*
*> -----<*
*> Terminate logic, after the loop <*
postLoop

CG_ENDIF
CG_IF(LANGUAGE,Java)
*> -----<*
*> This block of code is only included by CODEGEN for Java compile
<*
*> Note: HPS.INI value [JavaGen] MACRO=LANGUAGE=Java is required
<*
*> -----<*
*> -----<*
*> Initialize the window - procedure executed once at the start
<*
*> -----<*
proc for initialize type window
(p object type initializeEvent)
    preLoop
    if exitLoop or exitRule
        CUSTOMER_LIST.terminate
        proc return
    endif
    endif
    do
        preConverse
        if exitLoop or exitRule
            CUSTOMER_LIST.terminate
            proc return
        endif
        while skipDisp
        postConverse
        if exitLoop or exitRule
            CUSTOMER_LIST.terminate
            proc return
        endif
    endif
    enddo
endproc *> initializeEvent <*
*> -----<*

```

```

*> Converse the window - procedure executed until terminate event
<*
*> -----<*

proc for converse type window
(p object type converseEvent)
do
    postConverse
    if exitLoop or exitRule
        CUSTOMER_LIST.terminate
        proc return
    endif
    preConverse
    if exitLoop or exitRule
        CUSTOMER_LIST.terminate
        proc return
    endif
    while skipDisp
    enddo
endproc *> converseEvent <*

*> -----<*
*> Terminate the window - procedure executed once at the end
<*
*> -----<*

proc for terminate type window
(p object type terminateEvent)
    if exitRule
        proc return
    endif
    postLoop
endproc *> terminateEvent <*

CG_ENDIF
*> -----<*
*> *** THE CODE ABOVE THIS COMMENT BOX SHOULD NEVER BE CHANGED ***
<*

*> -----<*
*> Terminate the window - procedure executed once at the end
<*
*> -----<*
<*

proc for terminate type window
(p object type terminateEvent)
    if exitRule
        proc return
    endif
    postLoop
endproc *> terminateEvent <*
CG_ENDIF
*> -----<*
*> *** THE CODE ABOVE THIS COMMENT BOX SHOULD NEVER BE CHANGED ***

```

```
< *
* > ----- < *
```

## Handling Display Rules

Several types of display components can be used when developing applications in AppBuilder. These include:

- [See Standard Display Rules](#)
- [See Display Rules for Thin \(HTML\) Client](#)
- [See Non-Display Rules for Java Client](#)
- [See Display Rules with Third-Party Java Bean](#)
- [See User Events Handled by Java Bean](#)
- [See Rules Controlling Converse Window](#)
- [See Converse Events for Java](#)
- [See Domain Sets for Window Objects](#)

Use the Window Painter to create the presentation components for your AppBuilder application. For more information about the use of the Window Painter, refer to the *Development Tools Reference Guide*. To convert existing C applications to Java, you must understand how to handle [See Converse Events for Java](#).

## Standard Display Rules

For distributed applications, rules that display windows ( *display rules* ) have a special structure. Specifically, display rules consist *only* of the following:

- A declaration ( *DCL* ) section for local variables
- Event procedures
- Standard non-event procedures

The display rule typically contains a declaration section for variables used in the rule. This is typically followed by the window initialize and terminate event procedures, followed by additional event procedures to handle push button clicks, menu item selections, and other user actions. It might also contain some standard procedures, which would be called by code within the rule itself (rather than in response to user actions).

### [Example of a Display Rule](#)

```
dcl
variable1 OBJECT TYPE char
enddcl
proc for Click OBJECT CloseButton
( e object type ClickEvent )
MAIN_WINDOW.Terminate
endproc
```

## Display Rules for Thin (HTML) Client

For thin-client (servlet-based, client-side) applications, the rules execute in a Web or application server environment; therefore, certain considerations apply for using display rules. The system sends all display information to a browser in HTML format by way of the HTTP protocol. Even though this is a stateless, sessionless protocol, the AppBuilder applications can execute in the same way as the traditional GUI-based applications because the execution state is preserved in a cache or saved in the local file system.

Before sending the HTML to the Web browser, the AppBuilder rule saves or caches the context information to free up the resources on the Web server. The saved or cached information includes the view and the currently active rule stack. When the Web browser returns a request, the system restores the AppBuilder rule stack and reloads or reactivates the last display rule. After reloading the rules, the application invokes the appropriate event procedure. The caching and restoring of the state of execution between requests is completely transparent to the application. In thin-client applications, all display rules must be event-driven rules with the following default procedures or events.

Rule	Description
------	-------------

Rule or Window Initialize	Invoked when the window is loaded for the first time
Rule or Window Terminate	Must be called by the rule before exiting
RuleEnd	Must be called by the rule before exiting
ChildRuleEnd	Must be called by the parent rule to conclude processing of the child rule

The following are conventions for thin-client rules:

A root rule must be an event-driven rule.

Only an event-driven rule can call another event-driven rule using `Detach`.

An event-driven rule can call any other procedure rule.

The procedure rule can call another procedure rule including the remote rules but cannot call another event rule as in the second condition.

## Non-Display Rules for Java Client

In the Java client, rules that do *not* display windows are written in the same way as in earlier versions of the product; therefore, non-display rules can consist of:

- A declaration ( *dcl* ) section for local variables
- The rule body (which contains standard procedural code)
- Standard (non-event) procedures

In general, an AppBuilder Java client program can consist of display rules that use the new rule structure, and non-display rules that use standard Rules Language.

### [Example: Non-Display Rule in Java](#)

This is an example of a rule that uses a window with a push button. The window closes when the push button is pressed. The system identifier (HPSID) of the push button is `CloseButton` . The event procedure would be:

```
proc for Click OBJECT CloseButton
( e object type ClickEvent )
MAIN_WINDOW.Terminate
endproc
```

## Display Rules with Third-Party Java Bean

The Java client supports the use of third-party Java beans directly in the rules code. Here is an example of a display rule using a third-party Java bean.

### [Example of Display Rule with Java Bean](#)

This example demonstrates a rule that uses the Java tree control `JTree` bean.

```

dcl
java_tree OBJECT TYPE 'javax.swing.JTree' of javabeans;
enddcl
proc for Initialize OBJECT NC_BEAN_TEST_W
(e object type InitializeEvent)
*> create the Java Bean instance <*>
map new 'javax.swing.JTree' to java_tree
*> set properties on the Java Bean <*>
java_tree.SetSize(100,100)
java_tree.SetLocation(10,10)
NC_BEAN_TEST_W.AddChild(java_tree)
*> make the Java Bean visible <*>
NC_BEAN_TEST_W.SetVisible(true)
endproc

proc for Terminate OBJECT NC_BEAN_TEST_W
(e object pointer to TerminateEvent)
*> close the window <*>
NC_BEAN_TEST_W.Terminate
endproc

```

## User Events Handled by Java Bean

The following is an example of registering events for Java classes. This rule uses two different timer objects: one is the AppBuilder internal Timer and one is from the javax.swing package (Java Swing class, the set of graphical interface components).

### [Example of User Event with Java Bean](#)

This is an example of an event-handling procedure for button click events.

```

dcl
javaTimer OBJECT TYPE 'javax.swing.Timer';
nullActionListener object type 'java.awt.event.ActionListener';
enddcl

proc javaTimerProc for ActionPerformed Listener java.awt.event.ActionListener object javaTimer
(e object type 'java.awt.event.ActionEvent')
*> do what needs to be done when timer triggers <*>
map FD_INT + 1 to FD_INT
endproc

proc ExitClickProc FOR click object EXIT
(e object type ClickEvent)
WN_JAVABEAN_DEMO.terminate
endproc

proc ActivateClickProc FOR click object ACTIVATE
(e object type ClickEvent)
map new 'javax.swing.Timer'( 100, nullActionListener ) to javaTimer
*> dynamically add event procedure to handle timer events <*>
Handler javaTimer( javaTimerProc )
*> set timer properties and start the timer<*>
javaTimer.SetDelay( 1000 )
javaTimer.SetRepeats( True )
javaTimer.Start
map false to ACTIVATE.Enabled
endproc

```

# Rules Controlling Converse Window

With a traditional graphical user interface (GUI), client-side applications run the client side rules, display the GUI windows, and wait for the user inputs. After receiving user input, the client-side rules continue processing in one of the following methods:

- Through callbacks to event procedures defined within the rules
- From the statement following a *Converse Window* in a converse-driven rule

So, in traditional GUI-based AppBuilder applications, there is always a rule stack (a sequence of AppBuilder rules calling other rules), and the current execution state is maintained.

A rule might control a converse window. CONVERSE is a Rules Language verb that means "display" when applied to a user-interface window or "print" when applied to a report.

For example, a simple series of actions in a typical business application might include the following:

1. Copy the detailed information of a customer from a database to a window.
2. Display the window to show the data.
3. If the user selects *Update*, invoke another module to write the data to a file and return a code to the initiating module to indicate the changes from an end user.
4. If the user selects *Cancel*, return a code to indicate that the user did not make any changes.

## [Example: Converse Window Control](#)

### *Pseudocode*

In pseudocode, the above tasks might look like this:

```
PASS Customer detail information to Customer detail window data
DRAW Window
if User selects 'UPDATE' then
PASS Customer detail window data to Update module
CALL Update module
if Update module fails
PASS Set of customer detail error messages to Window message module
PASS Appropriate message IN Set of customer detail error messages to Window message module
PASS Customer detail window data to Window message module
CALL Window message module
else
PASS Update code to Invoking module
endif
else
PASS No change code to Invoking module
endif
```

### *Rules Language Code*

Here is a rule that accomplishes the same tasks. The words connected with underscores are specific instances of AppBuilder object types. The lines are numbered so we can explain significant elements in the Rules Language statements by stepping through the control logic a few lines at a time.

```

1 map DISPLAY_CUSTOMER_DETAIL_I to CUSTOMER_DETAIL
2 converse window CUSTOMER_DETAIL_WINDOW
3 caseof EVENT_NAME of HPS_EVENT_VIEW
4 case 'HPS_MENU_SELECT'
5 caseof EVENT_SOURCE of HPS_EVENT_VIEW
6 case 'UPDATE'
7 map CUSTOMER_DETAIL to UPDATE_CUSTOMER_DETAIL_I
8 use RULE UPDATE_CUSTOMER_DETAIL
9 if RETURN_CODE of UPDATE_CUSTOMER_DETAIL_O = 'FAILURE'
10 map CUSTOMER_DETAIL_FILE_MESSAGES to MESSAGE_SET_NAME of
11 SET_WINDOW_MESSAGE_I
12 map UPDATE_FAILED in CUSTOMER_DETAIL_FILE_MESSAGES to
13 TEXT_CODE of SET_WINDOW_MESSAGE_I
14 map CUSTOMER_DETAIL to WINDOW_LONG_NAME of
15 SET_WINDOW_MESSAGE_I
16 use COMPONENT SET_WINDOW_MESSAGE
17 else
18 map 'UPDATE' to RETURN_CODE of DISPLAY_CUSTOMER_DETAIL_O
19 endif
20 case other
21 map 'NO_CHANGE' to RETURN_CODE of DISPLAY_CUSTOMER_DETAIL_O
22 endcase
23 endcase

```

#### Explanations

The line numbers correspond to the line numbers in the Rules Language source code above.

1 map DISPLAY\_CUSTOMER\_DETAIL\_I to CUSTOMER\_DETAIL

DISPLAY\_CUSTOMER\_DETAIL is the name of the rule. The first line uses the MAP statement to copy the data in the rule input view, DISPLAY\_CUSTOMER\_DETAIL\_I, into the view of a window, CUSTOMER\_DETAIL\_WINDOW. The Rules Language assignment statement

```
map 10 to A
```

is analogous to an assignment statement in a traditional programming language, such as

```
A = 10
```

CUSTOMER\_DETAIL\_WINDOW is the name of the window in which the data—a record for one customer—is displayed to the user. CUSTOMER\_DETAIL is the name of the view of the window.

```
2 converse window CUSTOMER_DETAIL_WINDOW
```

CONVERSE is a Rules Language verb that means "display" when applied to a user-interface window or "print" when applied to a report. The rule issues the converse statement to display the window with the copied data to the user. The processing of a rule is suspended until that rule registers an event. For example, when the user clicks a button on the panel, codes identifying that button are placed in the fields of the predefined system view HPS\_EVENT\_VIEW and control returns to the rule.

The processing resumes at the line after the CONVERSE statement (line 3).

```

3 caseof EVENT_NAME of HPS_EVENT_VIEW
4 case 'HPS_MENU_SELECT'
5 caseof EVENT_SOURCE of HPS_EVENT_VIEW
6 case 'UPDATE'

```

Line 6 checks a field value of the HPS\_EVENT\_VIEW to see if the user selected the *Update* menu choice. (HPS\_EVENT\_VIEW is the work view of that rule that captures events from the end user.)

EVENT\_NAME and EVENT\_SOURCE are two fields of HPS\_EVENT\_VIEW that define an event. If the value of the EVENT\_SOURCE field is UPDATE, the rule copies the data in the window view to the input view of the rule that handles the file access (UPDATE\_CUSTOMER\_DETAIL), and then calls it with the use RULE statement.

```

7 map CUSTOMER_DETAIL to UPDATE_CUSTOMER_DETAIL_I
8 use RULE UPDATE_CUSTOMER_DETAIL
9 if RETURN_CODE of UPDATE_CUSTOMER_DETAIL_O = 'FAILURE'

```

UPDATE\_CUSTOMER\_DETAIL is a second rule that performs file updates. UPDATE\_CUSTOMER\_DETAIL\_I is its input view and UPDATE\_CUSTOMER\_DETAIL\_O is its output view.

Once processing control returns from UPDATE\_CUSTOMER\_DETAIL, the rule checks the value of the field in an output view that passes return codes (RETURN\_CODE).

The IF...ELSE...ENDIF is similar to statements from other programming languages: when the condition after IF is true, the statements between it and ELSE execute. Otherwise, the statements between ELSE and ENDIF execute. If the return code does not indicate failure, control skips to the ELSE clause on line 17 (described below).

```

10 map CUSTOMER_DETAIL_FILE_MESSAGES to MESSAGE_SET_NAME of
11 SET_WINDOW_MESSAGE_I
12 map UPDATE_FAILED in CUSTOMER_DETAIL_FILE_MESSAGES to
13 TEXT_CODE of SET_WINDOW_MESSAGE_I
14 map CUSTOMER_DETAIL to WINDOW_LONG_NAME of
15 SET_WINDOW_MESSAGE_I

```

If the return code is FAILURE, the SET\_WINDOW\_MESSAGE system component displays a customized error message for the user in a secondary window. SET\_WINDOW\_MESSAGE\_I is its input view.

CUSTOMER\_DETAIL\_FILE\_MESSAGES is a set of dialog messages and UPDATE\_FAILED is a specific error message.

```

17 else
18 map 'UPDATE' to RETURN_CODE of DISPLAY_CUSTOMER_DETAIL_O

```

If the return code does not indicate failure, control skips to the ELSE clause on line 17. The rule copies the character literal UPDATE into the RETURN\_CODE field of the output view, DISPLAY\_CUSTOMER\_DETAIL\_O.

```

20 case other
21 map 'NO_CHANGE' to RETURN_CODE of DISPLAY_CUSTOMER_DETAIL_O

```

If the value of the EVENT\_SOURCE field is anything other than UPDATE, the rule just copies the character literal NO\_CHANGE into the RETURN\_CODE field of DISPLAY\_CUSTOMER\_DETAIL\_O.

## Converse Events for Java

Java does not support the converse window statement used in display rules. To ease the conversion of an application from C to Java, a Converse event has been created in AppBuilder. The Converse event is an event on the window object that triggers exactly the same places where converse window now returns control to the calling rule. It also automatically updates the predefined system view HPS\_EVENT\_VIEW.

### [Example: Converse Event for Java](#)

An application based on the Converse Window statement is converted by moving the contents of the DO-WHILE loop surrounding the converse window into a Converse Event procedure. The exit condition that was originally coded in a DO-WHILE is now in an IF-ENDIF. The CASE statements are now inside a procedure (PROC).

*Original Converse Window Code for C:*



```

do while EVENT_SOURCE of HPS_EVENT_VIEW <> 'Exit'
converse WINDOW CUST_PD_DIS
caseof EVENT_SOURCE of HPS_EVENT_VIEW
case 'QUERY'
. . .
case 'NEW'
. . .
endcase
enddo

```

Equivalent Converse Event Code for Java:

```

proc for Converse OBJECT CUST_PD_DIS
(e object type ConverseEvent)
if EVENT_SOURCE of HPS_EVENT_VIEW=='Exit'
CUST_PD_DIS.terminate
endif
caseof EE
caseof EVENT_SOURCE of HPS_EVENT_VIEW
case 'QUERY'
. . .
case 'NEW'
. . .
endcase
endproc

```

## Domain Sets for Window Objects

For several window objects you can store a list of information in a set and then display those values in a combo box, read-only edit field, or a read-only table column. This section discusses the following:

- [See Defining Domain Sets for a Combo Box](#)
- [See Defining Domain Sets for a Read-Only Edit Field](#)
- [See Defining Domain Sets for a Read-Only Table Column](#)

### Defining Domain Sets for a Combo Box

There are two ways to specify a domain of values that can be displayed in a combo box:

- [See Define Sets with Values](#)
- [See Use Occurring Views](#)

#### Define Sets with Values

One way to define a domain of values that can be displayed in a window object is to define a set containing the possible values. For information about creating sets, refer to the section on Set Builder in the *Development Tools Reference Guide*.

#### Example set values

Coded value	Displayed value
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday

6	Saturday
7	Sunday

To define a domain for a combo box, complete the following steps:

1. Create the Set in the hierarchy.
2. Place the Set object under the Window object in the hierarchy of the window that contains the combo box.
3. Drag the set object icon from the hierarchy into the window in Window Painter and Window Painter automatically creates a combo box with the Domain property value assigned to the Set object. Or, Place a combo box on the window in Window Painter and set the Domain property value for the combo box.
4. Specify the current value (one of the values of the set) for that combo box in the Link property, using the Window Painter properties panel to enter the Link property for the combo box.

### **Use Occurring Views**

One way to define a domain of values that can be displayed in a combo box is to use occurring views with multiple fields. Each field contains a possible value that is displayed in the combo box.

1. Define the View and subordinate Fields in the Hierarchy window.
2. Place the View object under the Window object in the hierarchy of the window that contains the combo box.
3. Drag the View object icon from the hierarchy into the window in Window Painter. Window Painter automatically creates a combo box with the Domain property value assigned to the View object. Or, Place a combo box on the window in Window Painter and set the Domain property values for the combo box.
4. Specify the current value (one of the fields of the view) for that combo box in the Link property. Use the Window Painter properties panel to enter the Link property for the combo box.

### **Defining Domain Sets for a Read-Only Edit Field**

This is supported only for Java applications.

You can specify a domain for a read-only edit field in a window in Window Painter. The procedure is similar to [See Defining Domain Sets for a Combo Box](#). Follow the procedure in the [See Define Sets with Values](#). The set needs to be under a window, and the same set must be under a field. The set style must be *Lookup*.

### **Defining Domain Sets for a Read-Only Table Column**

You can specify a domain for a read-only column of a table or multicolumn list box (MCLB) in a window in Window Painter.

1. Create a Set in the hierarchy.
2. Double-click the Set object in the hierarchy to open the properties dialog for the Set. Change the Style to *Lookup* or *Error*, and click *OK* to save the change and close the dialog.
3. Place the Set object under the Window object in the hierarchy of the window that contains the MCLB.
4. Right-click the MCLB in the Window Painter, and select MCLB Editor to open the MCLB Editor dialog.
5. Select the column you want define the domain, and choose the name of the Set from the drop-down list for the Domain. Click *Set*, and then *OK* to save the change.

## **Native Files Handling**

Since the early 1960s, when people started using COBOL to process records from a file, very little about file handling has changed. Languages like C introduced structured data types, which are very much like records, and object-oriented languages like C++ and Java have classes that store data in much the same way. Programs still read records from files or databases into memory. The only differences are in the formats used to store the records on disk. There are fixed record formats, variable record formats, text and binary formats, and even XML formats, which are just another kind of variable text format. Database tables represent another kind of record format. There is no real difference between the formats, all of them being different ways of storing a record.

Programs still pass records to procedures that operate on them, records are still used to display forms on screen and to get input from users, and programs continue to store records back to files or databases.

Before Native File Handling became an AppBuilder feature, users needed to write code using built-in system components to read and write blocks of data to a file. Alternatively, they had the option of writing user components that were specific to a particular language and operating system. These custom user components had some very complex code that wasn't always reusable across operating systems. The user components would often need to be rewritten each time the application was ported to a new platform or language.

Forcing users to rewrite these user components violates the AppBuilder vision of platform neutrality because certain abstract properties of each language implementation remain basically the same, repeated in every platform-specific implementation. These properties could be moved up to the abstract application model so that generating code for different platforms might be possible.

This chapter explains the new AppBuilder Native File Handling feature. The topics in this chapter include:

- [Understanding Native Files](#)

- [Understanding File Organization Types](#)
- [Adding and Defining Records from Hierarchy](#)
- [Configuring a Rule to access a Data Source](#)
- [Writing Rules Code to Access Files](#)

## Understanding Native Files

The new Native File Handling capability was designed to provide a simple, record-based mechanism for storing AppBuilder views in files. This mechanism is now supported by extensions to the Rules Language, and is currently supported for Open Cobol generation only.

The Native File Handling feature provides a new model and language abstraction that can be used to generate the operating system-specific code. This means that the business logic is made more portable across different languages and platforms. Also, instead of being hidden and deeply embedded in rules code that invokes system components, data files are now part of the AppBuilder repository model, so they can be diagrammed, printed, and their role in the program can be better understood.

## Understanding File Organization Types

This section describes the main file organizational types:

- [Storing Records](#)
- [Fixed Format](#)
- [Variable Format](#)
- [Line-Sequential Format](#)

### Storing Records

The AppBuilder Native File Handling feature supports sequential access files, meaning that a predecessor-successor relationship among the records in a file is established by the order in which the records are placed in the file when it is created or extended.

Native Files can have different physical organizations: fixed format, variable format, and line-sequential format. This section describes these organizations in greater detail.

The following AppBuilder views are used as an example to demonstrate the meaning of each physical organization.

The RESIDENCE\_ADDR view shown below can be stored in a sequence of 77 bytes, and the BUSINESS\_ADDR view can be stored in a sequence of 97 bytes. A file can store many different kinds of records at the same time.

#### ***Native File view example***

VIEW: RESIDENCE\_ADDR

FIELD: ADDR\_TYPE

FIELD: STREET

FIELD: CITY

FIELD: ZIP

FIELD: FILL\_20

77 BYTES -- RESIDENCE\_ADDR RECORD

CHAR(2)

CHAR(40)

CHAR(30)

CHAR(5)

CHAR(20)

VIEW: BUSINESS\_ADDR

FIELD: ADDR\_TYPE

FIELD: SUITE

FIELD: STREET

FIELD: CITY

FIELD: ZIP

97 BYTES -- BUSINESS\_ADDR RECORD

CHAR(2)

CHAR(20)

CHAR(40)

CHAR(30)

CHAR(5)

**Fixed Format**

In the fixed format file, records with different sizes are stored in the same file, but the length of each record is always the same. Records smaller than the maximum length are not automatically padded, an extra field must be added to the view to make all views the same length. The additional padding wastes storage space, making this a somewhat inefficient way to store records.

**Native File fixed format sample**

01	212 Nickel Rd.	Springfield	42016	padding
02	Suite 200	4000 Crystal Pkwy	Springfield	42016

**Variable Format**

In the variable format file, records with different sizes are stored in the same file, but the length of each record varies. In order to read the correct number of bytes, the code reading the record must check the discriminant first, in order to know the number of bytes to read. In this example, the TYPE field is the discriminant, and a value of '01' tells the code to read a RESIDENCE\_ADDR record, which has 77 bytes. The size of the record is hard-coded into the code that reads the record, so each time the record is redefined to add more fields, all the corresponding code must be changed manually

**Native File variable format sample**

01	212 Nickel Rd.	Springfield	42018
----	----------------	-------------	-------

02	Suite 200	4000 Crystal Pkwy	Springfield	42018
----	-----------	-------------------	-------------	-------

## Line-Sequential Format

In the line-sequential format file, records are stored using their natural size, but are marked at the end with a special sequence of bytes called a delimiter. The code reading the record gets bytes in sequence until the delimiter is encountered. The delimiter is discarded and the record is stored in memory. This is more maintainable than the variable format since it doesn't rely on having the record length embedded in the code. When the record changes, the length does not have to be changed.

### Native File line-sequential format sample

01	212 Nickel Rd.	Springfield	42018	delimiter
----	----------------	-------------	-------	-----------

02	Suite 200	4000 Crystal Pkwy	Springfield	42018	delimiter
----	-----------	-------------------	-------------	-------	-----------

## Adding and Defining Records from Hierarchy

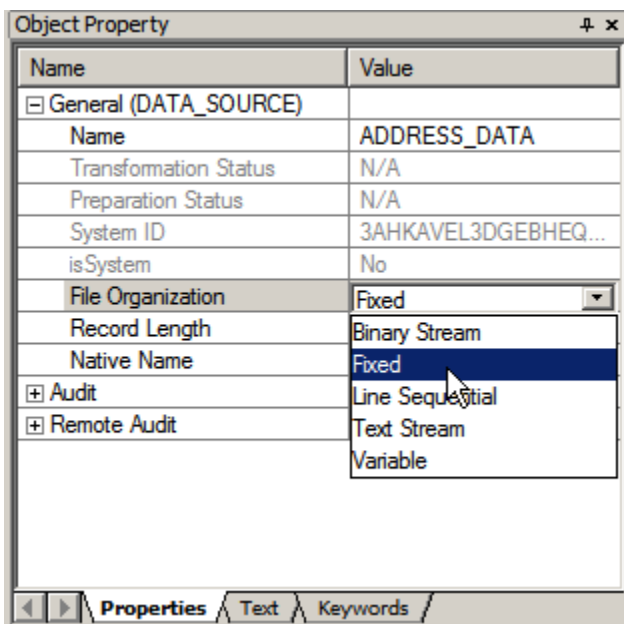
In order to begin using Native Files from your procedural AppBuilder rules code, you must start by defining a data source for each native file you want to read from or write to.

From the Repository tab, you can insert and configure a data source object.

### Creating and configuring a Data Source

1. In the Construction Workbench, on the Hierarchy Diagrammer, right-click the root of the hierarchy and choose the option **Insert > Native File > Data Source**. The Insert Data Source window appears.
2. Type **ADDRESS\_DATA** in the name field and press **Insert**.
3. In the Object Property panel, click on the **File Organization** property. From the drop-down list, choose **Fixed**.

#### Data Source object property



4. For fixed length files, you can leave the **Record Length** property as zero. This indicates the record length should be calculated using the largest child record size. For variable length files, the Record Length holds the maximum record size.

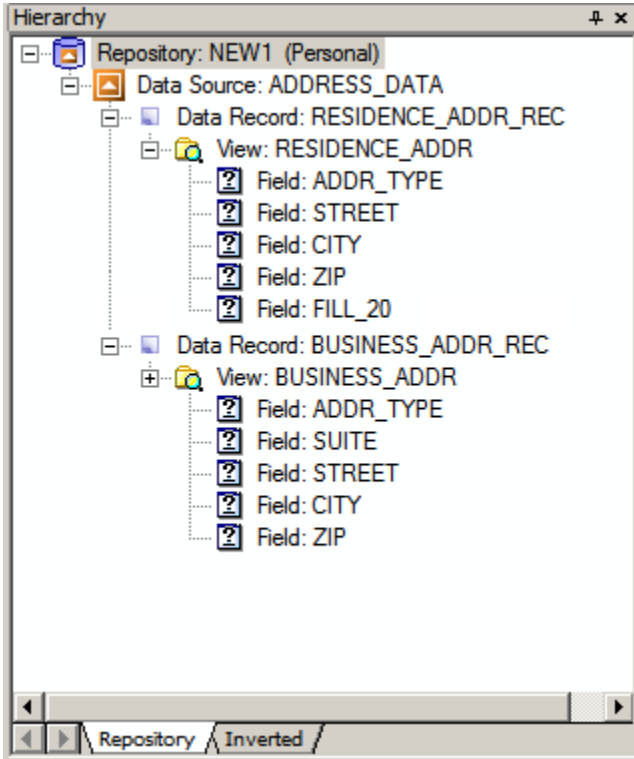
For more information on the Record types, see [Understanding File Organization Types](#).

## Creating and configuring a Data Record

Create a data record for each view you need to store in the target file. A data record is a platform-dependent mapping of the platform-independent data in a view. After the data record is shown in the hierarchy, you can insert a view as a child to the data records via DATA\_RECORD\_MAPS\_TO relationship.

1. In the Hierarchy window, right-click the data source ADDRESS\_DATA and choose **Insert Child > Native File > Data Record**. In the Insert Data Record window, type the name of the object (for example **RESIDENCE\_ADDR\_REC**), and press **Insert**.
2. Associate the new data record with an already existing view. To do this, right-click the data record, and from the context menu, choose **Maps to View**. Query for your view in the query window and press **Insert**.

### Data Records mapped to views in the hierarchy



Repeat this step to create a data record for each view that will be stored in the data source.

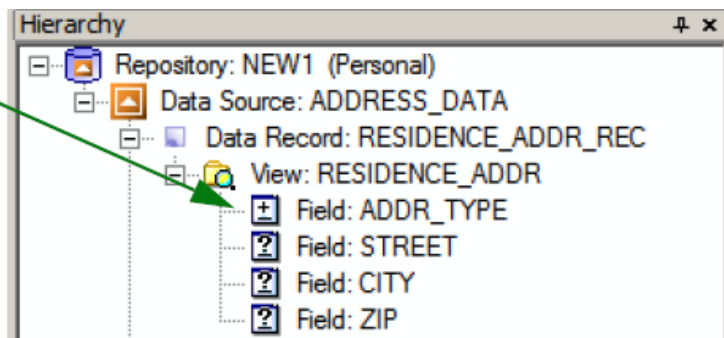
### Setting a Discriminant

Next you have to set the discriminant field. This field is used to distinguish one record type from another. Using discriminants is optional. The discriminant does not have to be the first field in the record, but it does need to be in the same place (its offset must be the same), so that it can be retrieved and tested using AppBuilder code.

1. To set the discriminant field on a view, right click the discriminant field underneath the linked view and choose **Set Discriminant** from the context menu. The field icon changes from a question mark to a '+' symbol.

#### Setting discriminant view

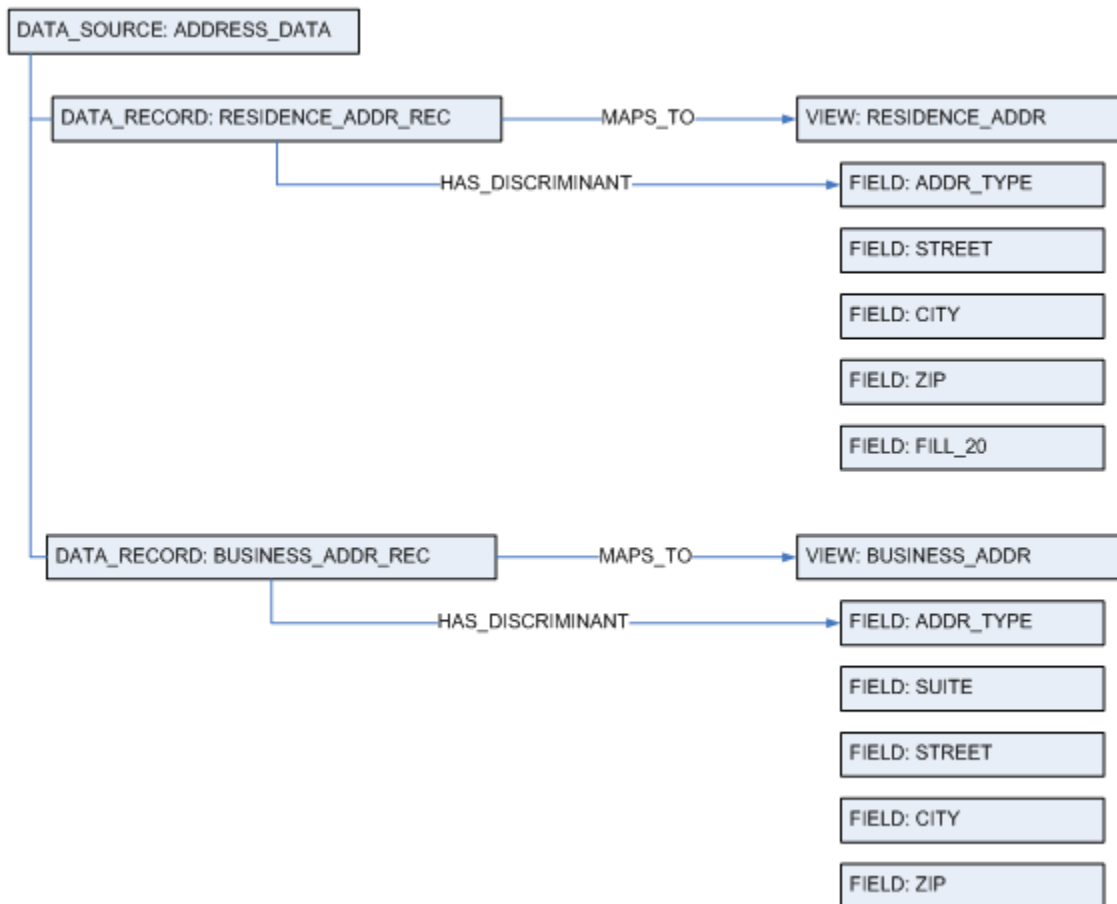
Field with discriminant



2. The field can be unset in the same way, by right-clicking the field and choosing **Clear Discriminant** from the context menu.
3. Perform this step for each Data Record in your file.

The sample model now looks, conceptually, like the following diagram:

#### Data source using Discriminant field



The diagram shows a sample repository with the two address records distinguished by the discriminant field, ADDR\_TYPE.

## Configuring a Rule to access a Data Source

To use a Data Source, a Rule must first establish a relationship to that Data Source. To do this, follow the steps.

1. In the Construction Workbench, on the Hierarchy Diagrammer, right-click the rule and choose **Insert > Native File > Data Source**. The Insert Data Source window appears.
2. Type **ADDRESS\_DATA** in the name field and press **Insert**.

## Writing Rules Code to Access Files

There are two different approaches to reading data from a Native File. The approach you choose depends on whether your code needs to make use of discriminant fields to distinguish one kind of record from another.

1. When not using discriminants, the coding is straightforward and easy to comprehend. The following code retrieves all records from the file. Before you proceed to write the rule, you need to know exactly which pattern was used to write the records to the file. In this case there is always a pair of addresses to be read from the file: a residence address followed by a business address.

```

Open (ADDRESS_DATA, 'r')
Do While (Not EOF(ADDRESS_DATA))
  ADDRESS_DATA >> RESIDENCE_ADDR
  ADDRESS_DATA >> BUSINESS_ADDR
Enddo
Close (ADDRESS_DATA)

```

- When using discriminants, you need to make use of the **Next()** function to retrieve the next record's discriminants. You may then test the discriminants as though they are members of the file, before reading them into the appropriate view.

```

Open (ADDRESS_DATA, 'r')
Do While (Next(ADDRESS_DATA) = 0 )
  if (ADDRESS_DATA.RESIDENCE_ADDR.ADDR_TYPE = "01")
    ADDRESS_DATA >> RESIDENCE_ADDR
  Endif
  if (ADDRESS_DATA.BUSINESS_ADDR.ADDR_TYPE = "02")
    ADDRESS_DATA >> BUSINESS_ADDR
  Endif
Enddo
Close (ADDRESS_DATA)

```

## Package Relationships for Native File Entities and Relationships

Data Source entities can be organized into packages to improve the organization of the application source code. A data source entity can only belong to a single package. Its membership in a package is purely optional.

### *PACKAGE\_CONTAINS relationship to Native File*

The screenshot displays a software interface with two main windows. On the left is a 'Hierarchy' window showing a tree structure under 'Repository: NEW1 (Personal)'. It contains a 'Package: pack' which includes a 'Data Source: ADDRESS\_DATA'. This data source has two 'Data Record' entries: 'RESIDENCE\_ADDR\_REC' and 'BUSINESS\_ADDR\_REC', each with a corresponding 'View'. On the right is an 'Object Property' window. A green arrow points from the 'ADDRESS\_DATA' data source in the hierarchy to the 'Relationship (PACKAGE)' property in the object property window. The 'Relationship (PACKAGE)' property is highlighted with a green box and has the value 'PACKAGE\_CONTAINS'. Other properties in the window include 'Name' (ADDRESS\_DATA), 'File Organization' (Fixed), and 'Sequence number' (10).

Name	Value
General (DATA_SOURCE)	
Name	ADDRESS_DATA
Transformation Status	N/A
Preparation Status	Dirty
System ID	3AHKAVEL3DGEBHEQ0D...
isSystem	No
File Organization	Fixed
Record Length	0
Native Name	
Audit	
Remote Audit	
Relationship (PACKAGE)	
Relationship type	PACKAGE_CONTAINS
Parent name	pack
Child name	ADDRESS_DATA
Separator ID	0
Sequence number	10
Relationship Audit	
Relationship Remote Audit	



# Working with User Components

You can write your own user components and incorporate existing procedures into your AppBuilder application. A user component is similar to a rule. It is a piece of an application you write to enable the application to do a specific unit of work. Like a rule, a component is a repository object that you can reuse within your application. Unlike a rule, which you write in the Rules Language, you code a component in the native language of the target environment. These tasks include:

- [Advantages of Using User Components](#)
- [Using User Components](#)
- [Using Subroutine Components](#)
- [Guidelines for Using User Components](#)
- [Specifying the Component Includes Directory](#)
- [Adding a User Component](#)
- [Writing a Java User Component](#)
- [Writing a C User Component](#)
- [Calling a C Component from Java](#)
- [Using Sample Component Code](#)
- [Data Type Comparison](#)

If you are writing a component to run on the host, its source can be written in C, COBOL, PL/I, or assembler. For more information about writing host user components, refer to the *Enterprise Application Guide*. PC components can be written in either C or Java.

Do not confuse user components that you write with the system components that the AppBuilder product provides in the default repository. System components perform various specialized functions, such as message display processing for your user interface. For more information about system components, see [Working with System Components](#) and refer to the *System Components Reference Guide*.

## Using Subroutine Components

A subroutine component only serves as a copybook; the source of the subroutine component is copied into the source of the parent component at preparation time.

To update the application hierarchy to use a subroutine component, perform the following:

1. Open your application hierarchy in the Hierarchy window.
2. Add a component as a child to the component to include the subroutine component, and name the component.
3. Select the correct execution environment attribute for the child component.
4. Select the Subroutine Option for the child component. To include the component as a subroutine, open the Object property window and choose **Yes** from the drop-down list of the Subroutine field.
5. Add a COPY statement as applicable to the language of your component in the parent component where you want the subroutine component to be included.



There can only be one level of subroutine components per root component for which a root component is any component other than a subroutine component.

## Specifying the Component Includes Directory

You can specify which directory AppBuilder looks in to find the user component includes, such as include files and header files. Set the directory for component includes in the [AP < platform >] section of the system initialization file, Hps.ini. The name of the section depends on the platform that you are using; it might be the [AP Windows] section. The setting is `_C_INC_DIR_`.

When you are preparing C components to be called from Java, the settings described in [INI setting descriptions](#) must be set in [AP Java] section of the Hps.ini file. See [Calling a C Component from Java](#) for additional information.

The default value is the temporary files directory under the product installation main directory such as `c:\AppBuilder\TEMP`. You may type in any path, including the drive letter and the directory structure. For example,

```
C_INC_DIR=D:\User\AppData\Comp\Includes
```

The following settings are listed in the Hps.ini file. Here is an explanation of each setting.

### INI setting descriptions

Setting	Description
C_INC_DIR	Where to place additional directories for header files if the directories are not already in the Include environment variables path. This setting affects rules and components prepared for C.
C_LIB_DIR	Where to place additional directories for libraries if the directories are not already in the LIB environment variables path. This setting affects rules and components prepared for C.

C_LIB_NAME	Where to list the names of additional libraries you are using. This setting affects rules and components prepared for C.
COMP_LINK_OBJJS	Where to place additional objects to link to your C User Components.
COMP_LINK_INCS	Where to place additional libraries to link to your C User Components.

## Advantages of Using User Components

Components can provide functionality the Rules Language lacks because they are written in the native language of the target platform. There are four major reasons to use a component:

### Component Uses

Reason	Explanation
Platform-specific functions	Because the Rules Language is portable, it does not support platform-specific functions such as transaction time stamping. Create a user component for these functions.
Native file access	This is related to platform-specific functions. The Rules Language supports file access through embedded SQL calls, but this may not work for every file. If your application accesses such a file, you need to write a user component. User components can contain embedded SQL as well.
Sophisticated mathematical functions	The Rules Language does not support calculus. If your application requires complex math functions, you need to write a user component.
Integrating existing code	If you have an existing non-AppBuilder module, you can incorporate it directly into your application as a user-written component, as opposed to recoding it in the Rules Language.

## Using User Components

Generally, you should use AppBuilder rules instead of components. Many of the advantages of working in the AppBuilder environment are not applicable to components in your application because of the following reasons:

- Unlike rules, components may not be portable from one target environment to another.
- Components are not as easy to reuse or edit as rules. You must have a developer fluent in the specific language of the component.
- You cannot use AppBuilder tools on components in your application. You need a different set of editing and debugging tools.
- You must manually update the information of a component in the repository when you re-Prepare it unless you define its input and output views to the repository.
- If a field is modified by a component outside of AppBuilder, the component must ensure that the field complies with the AppBuilder definition. See *Rules Language Reference Guide* .
- If a user component initializes or pads a field, it must perform as if within AppBuilder. Even if the field type has different characteristics in the language the user component is written in, the field must conform to what AppBuilder uses. See *Rules Language Reference Guide* .

## Guidelines for Using User Components

To write effective components, follow these two guidelines:

- [Design for Reuse](#)
- [Keep the Code Current](#)

### Design for Reuse

It is important to limit each component to one logical unit of work. This maximizes its potential for reuse. For example, suppose you have three basic tasks to perform:

1. Apply search criteria to a file to retrieve a record.
2. Perform a calculation on the record data.
3. Timestamp the results of the calculation.

If you define one component to accomplish all these tasks, you can reuse that component *only* if you need to perform *all three tasks* again in *precisely* the same sequence . If elsewhere you need only to perform a portion of the task (that is, the calculation) you cannot reuse the component. If you define a separate component for each task, you can reuse the components individually, as needed, throughout the application.

### Keep the Code Current

Because a component is really a traditional program, the pertinent data definitions must be declared in the code. Changing the data definition in the component input view (for example) does *not* automatically update the code. Generally, you must manually keep the component code and

view data definitions consistent.

## Adding a User Component

To add a user component, you must follow these steps.

1. [Creating or Updating the Application Hierarchy](#)
2. [Inserting the Component in the Calling Rule](#)
3. [Creating and Editing the Component Source Code](#)

You can also implement user components by using the Component Folder object and link external code objects through that folder. Refer to the *Development Tools Reference Guide* for a description of adding components and external files or objects in component folders.

### Creating or Updating the Application Hierarchy

To update the application hierarchy to use a component, complete the following steps:

1. Open your application hierarchy in the Hierarchy window.
2. Add a component as a child to the rule that calls the component. Name the component.
3. Select the correct execution environment attribute for the component.
4. If you are using implementation names, provide an implementation name for the component, and note it for later use.

Depending on the component function, you may also need to add views and fields to pass information to and from the component. Follow these steps to add views and fields:

1. Add any necessary views to the component, naming them as you would the views of a rule. For instance, you might have an input view, an output view, and a work view.
2. Select the relationship between the component and each view and make sure to set the view usage attribute to the correct type of view.
3. Add as many fields under the views as necessary for your application. This usually includes a return value field under the output view.
4. Add any other entities, such as files or sets, which the component accesses or refers to.
5. Commit the changes to your hierarchy.

### Inserting the Component in the Calling Rule

To insert the component in the rule that calls the component, complete the following steps:

1. Navigate to the Rule Painter.
2. In the source for the rule that calls the component, map any data the rule needs to pass into the fields of the component input view.
3. Insert a USE COMPONENT statement where you want to call the component  
`use COMPONENT < component_name >.`
4. After the USE COMPONENT statement, check any return codes that the component passes back in the return value field. If the return code indicates success, map the data from the component output view into the rule.
5. Commit your rule to save your changes.

### Creating and Editing the Component Source Code

Use a text editor to write or edit the source code of a component. If you are incorporating an existing component into the AppBuilder application, you must alter the declaration of your procedure to conform to the data types and parameters that AppBuilder applications expect.

- For C data types, refer to [AppBuilder-to-C data type comparison](#).
- For ClassicCOBOL data types, refer to [AppBuilder-to-ClassicCOBOL type comparison](#).
- For OpenCOBOL data types, refer to [AppBuilder-to-OpenCOBOL data type comparison](#).
- For Java data types, refer to [AppBuilder-to-Java data type comparison](#).

You may write a wrapper procedure to provide an entry point for the component or alter the declaration of your procedure. The latter method might be easier and perform slightly faster for very small simple procedures, but you should write a wrapper procedure for large or complex ones.

To write a wrapper procedure, complete the following steps:

1. From the Construction Workbench menu bar, click **Insert > Insert > Component**.  
The Insert Component window appears.
2. Click **Query** to display a list of available components.
3. Select your user component, and click **Insert** or double-click the component.  
The component is inserted into the hierarchy.
4. Double-click the component in the Hierarchy Window.  
A window opens in the Work Area with the standard AppBuilder editor.
5. Use the editor to type the source code for your component.  
For a C language component, the source is stored as `< AppBuilder >\TEMP\imp_name.c` if you are using implementation names and as `< AppBuilder >\TEMP\system_ID.c` otherwise.



You can configure the path names by changing the appropriate values in the **hps.ini** file. All path names given in this section are the defaults.

6. You can also use any third-party editor to code your component. Be sure to store the source in < *AppBuilder* >\TEMP.
7. From the Construction Workbench menu bar, select **File > Commit** to save your changes in the repository.

## Data Type Comparison

For more information about character field data types, refer to [Data Type Support](#).  
The following data type comparisons are described in this section:

- [AppBuilder-to-C Data Type Comparison](#)
- [AppBuilder-to-ClassicCOBOL Data Type Comparison](#)
- [AppBuilder-to-OpenCOBOL Data Type Comparison](#)
- [AppBuilder-to-Java Data Type Comparison](#)
- [See AppBuilder OO Primitive Types to Java Comparison](#)

### AppBuilder-to-C Data Type Comparison

The table below shows how AppBuilder data types map to C data types that contain values that are not zero terminated.

#### AppBuilder-to-C data type comparison

AppBuilder Data Type	C Data Type
CHAR(n)	char[n]
VARCHAR(n)	struct { short length; char value[n]; }
MIXED(n)	char[n] For more information, refer to <a href="#">Data Type Support</a> .
DBCS(n)	char[n*2] For more information, refer to <a href="#">Data Type Support</a> .
SMALLINT	short
INTEGER	long
DEC( <i>m,n</i> ), and PIC where: <i>m</i> is the total field length. <i>n</i> is the length of the fraction part.	For DEC, char[m+1], For PIC signed, char[m+1], For PIC unsigned, char[m] For more information, refer to <a href="#">Data Type Support</a> .
DATE	long containing the number of days starting January 1, 1 AD.
TIME	long containing the number of milliseconds elapsed since midnight.
TIMESTAMP	long[3] The first long is the date field as described above. The second long is the time field as described above. The third long contains the number of picoseconds (trillionths of a second) of the current millisecond.
BOOLEAN	short
IMAGE	char[256] containing file name for binary file
TEXT	char [256] containing file name for text file

### AppBuilder-to-ClassicCOBOL Data Type Comparison

The table below shows how AppBuilder data types map to ClassicCOBOL data types.

#### AppBuilder-to-ClassicCOBOL type comparison

AppBuilder Data Type	ClassicCOBOL Data Type
CHAR(n)	PIC X( n )
VARCHAR(n)	Length: PIC 9(4) COMP-4 Value: PIC X( n )
SMALLINT	PIC S9(4) COMP-4
INTEGER	PIC S9(8) COMP-4
DEC(m,n)	PIC S9( m-n )V( n ) SIGN IS LEADING AND SEPARATE
DATE	PIC S9(8) COMP-4 The number of days starting January 1, 1 AD.
TIME	PIC S9(8) COMP-4 Representing the number of milliseconds elapsed since midnight.
TIMESTAMP	Date: PIC S9(8) COMP-4 Time: PIC S9(8) COMP-4 Seconds: PIC S9(9) COMP-4 The number of picoseconds (trillionths of a second) of the current millisecond.

### AppBuilder-to-OpenCOBOL Data Type Comparison

The table below shows how AppBuilder data types map to OpenCOBOL data types.

#### AppBuilder-to-OpenCOBOL data type comparison

AppBuilder Data Type	OpenCOBOL Data Type
CHAR(n)	PIC X( n )
VARCHAR(n)	Length: PIC 9(4) COMP-4 Value: PIC X( n )
SMALLINT	PIC S9(4) COMP-4
INTEGER	PIC S9(8) COMP-4
DEC(m,n)	PIC S9( m-n )V( n ) SIGN IS LEADING AND SEPARATE
DATE	FMT-DATE1-F. YYYY PIC 9999. DATEDLM PIC X. MO PIC 99. DATEDLM PIC X. DD PIC 99. DATE1-F REDEFINES FMT-DATE1-F PIC X(10). The number of days starting January 1, 1 AD.
TIME	FMT-TIME1-F. HH PIC 99. TIMEDLM PIC X. MM PIC 99. TIMEDLM PIC X. SS PIC 99. TIMEDLM PIC X. MS PIC 999. TIME1-F REDEFINES FMT-TIME1-F PIC X(12). Representing the number of milliseconds elapsed since midnight.

TIMESTAMP	FMT-TIMESTAMP-1-F. YYYY PIC 9999. DATEDLM PIC X. MO PIC 99. DATEDLM PIC X. DD PIC 99. DATEDLM PIC X. HH PIC 99. TIMEDLM PIC X. MM PIC 99. TIMEDLM PIC X. SS PIC 99. TIMEDLM PIC X. MS PIC 999999. TIMESTAMP-1-F REDEFINES FMT-TIMESTAMP-1-F PIC X(26). The number of picoseconds (trillionths of a second) of the current millisecond.
-----------	---

### AppBuilder-to-Java Data Type Comparison

The table below shows how AppBuilder data types map to Java data types.

#### AppBuilder-to-Java data type comparison

AppBuilder Data Type	Java Data Type
CHAR(n)	AbfString(AbfString:CHAR,n)
VARCHAR(n)	AbfString(AbfString:VARCHAR,n)
MIXED(n)	AbfString(AbfString:MIXED,n)
DBCS(n)	AbfString(AbfString:DBCS,n)
SMALLINT	AbfShortInt
INTEGER	AbfLongInt
BOOLEAN	AbfBoolean
DEC(m,n)	AbfDecimal( AbfDecimal.DEC, m, n )
PIC(m,n)	PIC(m,n)AbfDecimal( AbfDecimal.UPIC, m, n ) for unsigned PIC, or AbfDecimal( AbfDecimal.SPIC, m, n ) for signed PIC
DATE	AbfDate
TIME	AbfTime
TIMESTAMP	AbfTimeStamp
IMAGE	AbfBlob( AbfBlob.IMAGE )
TEXT	AbfBlob( AbfBlob.TEXT )

### AppBuilder OO Primitive Types to Java Comparison

The table below shows how AppBuilder OO primitive data types map to Java data types.

#### AppBuilder OO Primitives to Java type comparison

AppBuilder OO Primitive Data Type	Java Data Type
blob	java.lang.String
boolean	boolean
byte	byte
char	char
clob	java.lang.String

date	ABODate
decimal	ABODecimal
double	double
fixchar	ABOFixChar
float	float
integer	int
long	long
short	short
string	java.lang.String
time	ABOTime
timestamp	ABOTimeStamp
varchar	ABOVarChar

## Writing a Java User Component

To write a user component in Java, complete the following steps:

1. Add a component and its input and output view to the rule in the hierarchy.
2. Use the component editor to edit the component. Launch the editor by selecting the component in the Hierarchy window and then right-clicking and selecting Open Component.
3. In the editor, write Java code for a public class that extends `AbfNativeModule`. The component class must be public or you get an exception at runtime. The class name must be the component's mixed case name. The package name should be the same as specified by the `COMPONENT_PACKAGE` variable in the `hps.ini` file. If `Package` object is attached to component, instead of using `COMPONENT_PACKAGE` ini setting, the package name comes from `Package` object (Long name).
4. Override the `run` (`AbfStruct` input, `AbfStruct` output) method inherited from the `AbfNativeModule` class. In this method, cast the input and output views to the actual view classes and provide whatever logic you wish.
5. Be sure to include import statements for `appbuilder.AbfNativeModule` and `appbuilder.util.AbfStruct` so the Java compiler can locate these classes.

### Naming conventions

The class name of the component must be the same as the generated name. You must understand how the system generates names for both components and fields. The naming convention for all generated files is in line with Java naming conventions. For example, if you have an entity named `CUST_DTL_DIS` (for customer detail display), the generated entity names would be:

#### CUST\_DTL\_DIS example

Object	Generated Name
Rule	<i>Cust_Dtl_Dis.class</i>
View	<i>Cust_Dtl_Dis_v.class</i>
Occurring View	<i>Cust_Dtl_Dis_va.class</i>
Component	<i>Cust_Dtl_Dis_c.class</i>
Set	<i>Cust_Dtl_Dis_s.class</i>

For field and variable names, AppBuilder converts the object long names to mixed case, with the initial letter of each component of the long name delimited by underscores becoming uppercase and the rest lowercase and the underscore is removed. These mixed case names are used as the names of any classes and variables generated for hierarchy objects. Classes get a suffix corresponding to the object type (except rules, which get no suffix) and variable names get a prefix corresponding to the object type.

However, this process of removing the underscores is not appropriate for numeric components within a long name, since removing underscores would result in merging of the numbers without any means of distinguishing the component parts. Thus, field and variable names including numbers do not have the underscores removed when they are adjacent to the underscore.

This naming convention applies for all generated entities. Follow this convention when naming components. For an example of a Java user component, see [Sample Java Component Code](#).

To write user components in Java, you need to know the applicable Java classes and the available methods. Refer to [AppBuilder-to-Java data](#)

[type comparison](#) for a list of supported Java classes and methods. AppBuilder uses certain Java classes to represent the standard AppBuilder data types. The component editor has certain methods to both set and get values.

## Writing a C User Component

Generally, code your component source just as you would code any other C language program. However, you must insert certain statements into the code to integrate the component into your application.

1. In addition to any standard C *include* statements your application needs, include a header file named after the system identifier of the component in your source:

```
#include " _SYSTEM_ID_ .HDR"
```

Internally, the AppBuilder environment refers to the component by its system identifier – not its name or implementation name. This header file (stored as `<AppBuilder>\DBG\SYSTEM_ID.HDR`) converts the component input and output views and fields to corresponding C constructs. It prefixes all view and field names in the input and output views with a `V_` (capital v and underscore). For example, if the name of the output view of a component is `TEST_INTEGER_O`, this header file changes it to `V_TEST_INTEGER_O` within the component. When you refer to a view or field from within the component, you must use the name as it appears in the header file.

The header file also builds a structure for each input and output view. It prefixes these structures with a `T_` (capital t and underscore). For example, if an input view `TEST_INTEGER_I` is defined as containing one `CHAR` (50) field called `WINDOW_RETCODE` and one `SMALLINT` field called `BASE_NUMBER`, then the generated C structure looks something like the following example.

### Example Code: C User Component

```
static struct T_TEST_INTEGER_I
{
    char V_WINDOW_RETCODE[50];
    T_B V_BASE_NUMBER;
}
```



Some data type conversions, such as `T_B` for a small integer, are contained in the `HPSDEF.H` file, which is an include file in the `SYSTEM_ID.HDR` file. The header file also relates a pointer from each structure to its corresponding view. For example:

```
static struct T_TEST_INTEGER_I FAR * V_TEST_INTEGER_I;
```

The `SYSTEM_ID.HDR` file contains `#define` statements for the input and output view names. No matter what the view names are, you can refer to them generically within the component:

```
#define INPUT_VIEW_NAME V_TEST_INTEGER_I
```

If `dna.h`, located in `appbuilder\nt\sys\inc`, is included in the source code for a user component, the component compile fails. AppBuilder defines a macro `DECIMAL` in `HPSDEFS.h`, located in `appbuilder\nt\sys\inc`. This macro is also defined as a structure in the Microsoft header `wtypes.h`. Including `dna.h` automatically includes `wtypes.h`. The work-around is to always list the user component header file `SYSTEMID.HDR` first. Then undefine the `DECIMAL` macro. Any other headers can be added after the `undefine`.

### Example Code: Work-Around



```
#include <ZADQKXH.HDR>
#undef DECIMAL
#include <dna.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

- For both client and server-side components, specify the name of the component function as follows:

```
int _SYSTEM_ID_ (DCLRULEPARAM)
```

The DCLRULEPARAM input parameter acts as a handle that must be used in all calls the component makes to the runtime system.



The name of a component operating in a UNIX environment must be all capital letters.

- You must call two access routines to tell the component where the input and output views are physically located in the storage area for the calling rule. For both client and the server sides, include the following two statements in every C component:

```
INPUT_VIEW_NAME = RULE_INPUT_VIEW_ADDRESS
OUTPUT_VIEW_NAME = RULE_OUTPUT_VIEW_ADDRESS
```

- Code the main part of the component.
- At the end of the component, return DONE\_WITH\_RULE to enable the calling rule to continue processing. The entry point in a component is called repeatedly until the component returns this value (defined in r\_str.inc) to its caller. Each call to the component corresponds to one entry into the message handling loop of the calling application. This allows a component that needs a long time to execute to break its operation into subsections, giving other applications a chance to run concurrently. If you do not explicitly return the value DONE\_WITH\_RULE at some point, the system locks up. Each time the AppBuilder environment invokes the component entry point, both global and automatic data are re-initialized. It is not safe to define a global variable in the component and rely on its value staying the same from one invocation to the next. If you need to preserve data across invocations, you must allocate space for it in the component input view and write the values there, so the calling rule preserves them and passes them back on subsequent iterations.
- If global views or work views are used in the component, then the following calls must be included in the component code before any view is used:

```
get_addresses(HPSMEM)
memory_allocation(HPSMEM)
```

For an example of a user component, see [See Sample C Component Code](#).

## Calling a C Component from Java

You can call a C user component from a Java application. AppBuilder can prepare a C user component called by a rule in the Java application. A C component is called exactly the same way as a Java component. AppBuilder marshals the input and output views and allocates the work views. Global views are not supported. Be sure to copy the following files to the <AppBuilder> \nt\sys\inc directory or set the path to these files in the system environment:

- files in the Java SDK install include directory:
  - jawt.h
  - jni.h
  - jvmdi.h
  - jvmpi.h
- files in a win32 subdirectory in the include directory:
  - jawt\_md.h
  - jni\_md.h

In order to prepare C components that can be called from a Java application, some settings in the [AP Java] section of the Hps.ini file must be set as discussed in [Specifying the Component Includes Directory](#).

Check also the following sections for any additional changes that may be necessary for Java to call C components:

- [Backward Compatibility](#)
- [Thread-Safe Components](#)
- [Work Views](#)

A component with these changes can be successfully prepared for both Java and C environments.

The AppBuilder-defined macro `C_COMPONENT_FOR_JAVA` can be used in the component code. It is defined only when a C component is prepared for Java, but not defined for C.

Be sure to put `<AppBuilder>\java\rt\bin` directory in the system path for the DLL to be loaded at runtime. If this is not in the path, an Unsatisfied Link Error occurs when the C component is called.

## Backward Compatibility

If a C component satisfies the following requirements, it can be prepared for Java and C without any changes.

1. The component function must be defined as:

```
int ZAAAB25 (DCLRULEPARAM)
{
}
```

If work views are used in the component, then the following calls must be included in the component code before any view is used:

```
get_addresses(HPSMEM)
memory_allocation(HPSMEM)
```

2. The component cannot use global views.
3. The component cannot use database access.
4. The component must include the following code:

```
INPUT_VIEW_NAME = RULE_INPUT_VIEW_ADDRESS;
OUTPUT_VIEW_NAME = RULE_OUTPUT_VIEW_ADDRESS;
```

Use that to obtain input and output views and only use the AppBuilder-defined macros `INPUT_VIEW_NAME` and `OUTPUT_VIEW_NAME` to access the input and output view fields.

## Thread-Safe Components

If a C component must be thread-safe, its code must be slightly modified. The following changes are necessary:

1. Add this as the first line of the component code:

```
#define THREAD_SAFE
```

2. In the declaration section of the component, add this:

```
DCL_LOCAL_VARS;
```

3. In the initialization section of the component, before any component code is executed, add this:

```
INIT_LOCAL_VARS;
```

### Example: Thread\_Safe Components

```

#define THREAD_SAFE
#define NEW_COMPONENT_HEADERS
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "zaaab25.hdr"

int ZAAAB25 (DCLRULEPARAM)
{
    int i;
    FILE * outf;
    DCL_LOCAL_VARS;
    INIT_LOCAL_VARS;

    memory_allocation(pThis);
    get_addresses(pThis);

    INPUT_VIEW_NAME = RULE_INPUT_VIEW_ADDRESS;
    OUTPUT_VIEW_NAME = RULE_OUTPUT_VIEW_ADDRESS;

    outf = fopen("component.log", "w");
    fprintf (outf, "C Component started\n");

    for (i =10; i < 20; i++)
    {
        OUTPUT_VIEW_NAME->V_TEST_FLD_1\[i\] =
            INPUT_VIEW_NAME->V_TEST_FLD_2 \[i-10\];
    }
    fprintf (outf, "Component ended\n");
    fclose(outf);
    return DONE_WITH_RULE;
}

```

## Work Views

For the work view, you must look at the header (HDR) file. It makes sense first to prepare an empty component with all the views attached. To save the HDR file, from the Construction Workbench menu, select **Tools > Workbench Options > Prepare** and check **"Save intermediate files"** option.

After you prepare the component, the `< system_ID >.HDR` file is in the `< AppBuilder >\DBG` directory.

Find a definition for a work view pointer as the following code illustrates:

```

static TD_TEST_WORK_VIEW_1 SEER_FAR * V_TEST_WORK_VIEW_1;

```

### Example: View Structure in the Header File

The following is an example of a view structure defined in the HDR file:

```

struct T_TEST_WORK_VIEW_1
{
    T_B V_TEST_VARCHAR_15_F_LEN;

    char V_TEST_VARCHAR_15_F[15];
};

typedef struct T_TEST_WORK_VIEW_1 TD_TEST_WORK_VIEW_1;

Then use V_TEST_WORK_VIEW_1 in the component:
V_TEST_WORK_VIEW_1 -> V_TEST_VARCHAR_15_F\[0\] = 'A';
V_TEST_WORK_VIEW_1 -> V_TEST_VARCHAR_15_F\[1\] = 'b';
V_TEST_WORK_VIEW_1 -> V_TEST_VARCHAR_15_F_LEN = 2;

```

## Using Sample Component Code

This section discusses how to use sample component code for Java and C.

- [Sample Java Component Code](#)
- [Sample C Component Code](#)

### Sample Java Component Code

This sample assumes that the names of the objects are:

#### Object names for sample Java code

Object	Long Name in Repository	Generated Name
Component	<i>MY_COMPONENT</i>	<i>My_Component_c</i>
Input View	<i>MY_COMPONENT_I</i>	<i>My_Component_I_v</i>
Output View	<i>MY_COMPONENT_O</i>	<i>My_Component_O_v</i>
Field	<i>MY_FIELD</i>	<i>fMyField</i>

Remember these guidelines:

- The long name in the repository must be all caps with underscores.
- The generated class name must be the user component name in mixed case with underscores.
- The generated instance name for a field must be mixed case with an f prefix, such as fMyField.

#### Example: Using Sample Java Component

```

package component;
import appbuilder.AbfNativeModule;
import appbuilder.util.AbfStruct;
import view.*;
public class My_Component_c extends AbfNativeModule
{
    public void run(AbfStruct input, AbfStruct output)
    {
        //cast the input view to the actual type
        My_Component_I_v inputView = (My_Component_I_v) input;
        //cast the output view to the actual type
        My_Component_O_v outputView = (My_Component_O_v) output;
        //get field values like this
        String tempSt = "" + inputView.fMyField;
    }
}

```

## Sample C Component Code

The following code is for *either* a client or a server-side component. It has a system identifier of **AAC4RV** and performs a factorial on an integer. Include the final return statement (*return (DONE\_WITH\_RULE);*).

### Example: Using Sample C Component

```
#include <stdio.h>
#include <string.h>
#include <AAC4RV.HDR>
int AAC4RV (DCLRULEPARAM)
{
  int i;
  INPUT_VIEW_NAME = RULE_INPUT_VIEW_ADDRESS;
  OUTPUT_VIEW_NAME = RULE_OUTPUT_VIEW_ADDRESS;
  OUTPUT_VIEW_NAME->V_PYR_INT_O_F = 1;

  for (i=INPUT_VIEW_NAME->V_PYR_INT_F;i>1;i--)
  OUTPUT_VIEW_NAME->V_PYR_INT_O_F =
    OUTPUT_VIEW_NAME->V_PYR_INT_O_F*i;

  return (DONE_WITH_RULE);
}
```

## Working with System Components

System components perform runtime functions such as creating pop-up windows for error and warning messages, changing the color and visibility of push buttons and fields, and accessing system information such as date and time.

There are system components that are only supported in C; others are supported in C and Java. Some system components execute immediately when a USE COMPONENT statement executes; other system components execute at the next window. Use system components with the user components that you write. For more information about user components, see [Working with User Components](#).

All the system components are described in detail, including details on the platform on which the components are supported, in the *System Components Reference Guide*.

This section covers the following topics:

- [See Hierarchy of a Component](#)
- [See Thin Client Components](#)

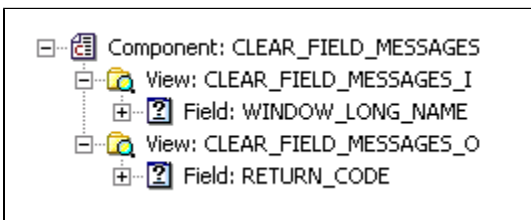
## Hierarchy of a Component

A system component typically contains one input view and one output view. Each view contains one or more fields.

- [See Input View](#)
- [See Output View](#)
- [See Deferred Components](#)

Component hierarchy shows how a system component appears in the Hierarchy window of the Construction Workbench.

### Component hierarchy



A component does not stand alone. It must be considered in context with the application and the calling rule. To make a component functional, attach the component to the appropriate rule in the hierarchy. Use MAP statements (as required) to manipulate the component input and output views.

## Input View

Use an input view to provide input data to the component. The input views of some components own a field named WINDOW\_LONG\_NAME or VIEW\_LONG\_NAME, in which you specify the target window or view on which to perform the action. For example, the component in [component hierarchy](#) returns all fields in a specified window to their non-error condition. To specify the window on which to reset these fields, add the following statements to the calling rule:

```
map 'name_of_window_to_reset_fields' to WINDOW_LONG_NAME of
CLEAR_FIELD_MESSAGES_I
use component CLEAR_FIELD_MESSAGES
```

Some components, such as GET\_USER\_WORKSTATION\_ID, do not have an input view because they do not need specific information to execute. An input view passes data into a rule. A rule can have only one input view. The standard naming convention for an input view is to append \_I to the rule name.

## Output View

Data resulting from the execution of a component is stored in the output view, typically contained in the RETURN\_CODE field and other fields. An output view passes data from a rule to its parent rule. A rule can have only one output view. The standard naming convention for an output view is to append \_O to the rule name. Consider the following example:

```
if RETURN_CODE of CLEAR_FIELD_MESSAGES_O = SUCCESS in RETURN_CODES
  *> action success <*
endif
```

## Deferred Components

Components that do not execute immediately when a USE COMPONENT statement executes are called deferred components. The system defers their execution until the next CONVERSE WINDOW or CONVERSE\_EVENT statement executes.

Deferred component functionality is for C applications only. While some deferred components are supported for Java applications, deferred components do not execute in the same way as they do for C applications. In Java, these components act on the current window and not on the next conversed window. This is because there is no deferred component functionality in Java.

Because most of these deferred components have no window name in the input view, they execute on the window that is the object of the CONVERSE statement. Thus, the fields of an input view usually cannot be validated when a USE COMPONENT statement executes. For this reason, the RETURN\_CODE field of an output view is set to 1, unless a system-level error sets it to 0. When the CONVERSE WINDOW statement executes, it checks the contents of the input view. If the CONVERSE WINDOW statement detects an error, a message window displays the error.

Several deferred components also have a WINDOW\_LONG\_NAME field. The name mapped to the WINDOW\_LONG\_NAME field must match the name of the next window conversed.

See the *System Components Reference Guide* for a complete list of deferred components.

## Thin Client Components

By displaying dynamically generated HTML, your Web application can display error messages stored in a database. See [Using Dynamically Generated HTML Components](#) for more information.

Use the following system components to display dynamically generated HTML fragments at execution time:

**HPS\_SET\_HTML\_FILE** displays dynamically generated HTML code stored in a separate HTML file. This HTML file cannot be a complete page, that is, it cannot contain <HTML>, <HEAD>, or <BODY> tags.

**HPS\_SET\_HTML\_FRAGMENT** displays dynamically generated HTML code stored as a character string within the rule that uses the component.

Before using these system components, use the Repository Administration tool or the *Migration Import* to import HPS\_SET\_HTML\_FILE and HPS\_SET\_HTML\_FRAGMENT. For information about the Repository Administration tool, see the *Repository Administration Guide for Workgroup and Personal Repositories*. For more information about these components, see the *System Components Reference Guide*.

## Using Dynamically Generated HTML Components

Use the following procedure to include dynamically generated HTML in a window at execution time.

1. Edit the HTML for the window. (Refer to the *Development Tools Reference Guide* for information about using the HTML Editor.) Insert the following HTML statement in the location to include the dynamically generated HTML:

```
<LEVEL_DYNAMIC name=" _macroname_ ">
```

where the *macroname* is the name used in the input view of the system component.



Verify that the dynamically generated HTML code is in the indicated location.

2. Save the HTML file and close the HTML editor. In AppBuilder, commit the changes to the repository.
3. To use the component, add the HPS\_SET\_HTML\_FILE or HPS\_SET\_HTML\_FRAGMENT statement to the display rule.
4. Prepare the project, including the window and rules.

At execution time, the HTML tag, LEVEL\_DYNAMIC is replaced with the HTML fragment or file corresponding to *macroname* (as entered in step 1). Call the component each time the window is displayed, because returning control to the rule resets the HTML file names.



If you make any changes in Window Painter and select *Regenerate HTML*, the system overwrites any changes you might have made in your HTML editor, including the HTML comment tags used by the component.

Make sure that your application deletes any temporary files that it creates.

### **Example: Thin-client Component**

The following are examples of how to use HTML components in your code.

#### **Example 1: HPS\_SET\_HTML\_FILE**

```
map "macroname" to HPS_HTML_MACRO of HPS_SET_HTML_FILE_I
map "filename" to HPS_HTML_FILE_NAME of HPS_SET_HTML_FILE_I
use component HPS_SET_HTML_FILE
```

where *filename* is one of the following:

- An existing file (including path) to which you refer. The existing files can be stored in the repository in a Component Folder attached to the Window and all attached files are prepared with the Window to the includes subdirectory.
- A temporary file (including path), created with user-written components, that contains the HTML fragment. Remember that your application must delete the temporary file after using it.



The HTML file must be a fragment – not a complete page. It cannot contain <HTML>, <HEAD>, or <BODY> tags.

At execution time, the HTML comment tag is replaced with the HTML fragment in the corresponding file.

#### **Example 2: HPS\_SET\_HTML\_FRAGMENT**

```

map " _macroname_ " to HPS_HTML_MACRO of HPS_SET_HTML_FRAGMENT_I
map " _This is the character string to be displayed_ "
to HPS_HTML_FRAGMENT of HPS_SET_HTML_FRAGMENT_I
map _40_ to HPS_HTML_FRAGMENT_LENGTH of HPS_SET_HTML_FRAGMENT_I
map _0_ to HPS_HTML_FRAGMENT_APPEND of HPS_SET_HTML_FRAGMENT_I
use component HPS_SET_HTML_FRAGMENT

```

At execution time, the HTML comment tag is replaced with the *character string* .

## Creating User Assistance for Applications

Applications commonly require some forms of user assistance, whether it is as a reference (how many characters can be entered in this field?) or to aid in particular operations (how do I write a rule to call a remote database?) or some other kind. User assistance can include many forms; some common forms in Windows applications are listed in table 9-1:

### Examples of User Assistance

Form	Accessed	Commonly used for	Target audience
Wizards	within the application	Complex or low-frequency tasks	Any user
Help system, including Reference Help Procedural Help Conceptual Help	Help menu command(s); sometimes F1. Help may appear in separate window(s), in an embedded window, in pop-ups, etc., depending upon a number of other factors.	Presenting all pertinent information for users of the application. <ul style="list-style-type: none"> <li>References may include specifics as to syntax, definitions, error messages and exceptions.</li> <li>Procedural frequently deals with the operation of user interfaces.</li> <li>Conceptual generally deals with how processes are designed to operate within the application.</li> </ul>	Any user
What's This Help	generally via a dedicated Help button on the application's toolbar	Explanation of screen items	All users
Status bar messages	provided as text on the status bar or in field of the application	Explanation for complex interfaces	All users
ToolTips	mouse over a control	Explanation of the user interface	Any user
Tutorials	generally external to an application or tool	Introduction to an application and a walk-through of a process	Novice users

The Help system may or may not be context-sensitive; even when it is context-sensitive, the help system normally includes information well beyond that used in context-sensitive topics. In addition to these online forms, user assistance may include traditional print (or PDF) manuals. AppBuilder supports several kinds of user assistance. In general, the forms of user assistance can most generally be categorized as Internal/External. Those listed in the AppBuilder Internal section are part of the application, stored as part of its fields in the repository, and can be viewed in several different ways. The fact that they are built and stored internally is significant. Those forms of user assistance listed in the AppBuilder External section are ones which must be created and stored externally and then linked to your AppBuilder application. In this case, the content is not stored within AppBuilder. There are numerous forms of online help available. Not all are feasible on all platforms and environments. Even when they can be created, most differ in display, performance, and other properties. There are also many external tools possible for creating and generating online help or other documentation, each with different strengths. We recommend learning as much as possible about various online formats, their limits and restrictions, the tools used to create and generate these formats, and how best to address the user assistance needs associated with the development of your projects.

## AppBuilder Internal

AppBuilder Internal refers to data that is stored by AppBuilder in its repository and can be part of the application. AppBuilder windows and window controls generally have two properties which can be used to provide forms of user assistance. The two properties are the fields named *Help* and *Short Help* . The help information entered and stored in AppBuilder produces .HLP files. These files are not Windows-based .HLP files and cannot be opened independent of AppBuilder.



## ToolTips

The AppBuilder C Client cannot display help text as tooltips.

The AppBuilder Java Client can display text as tooltips.

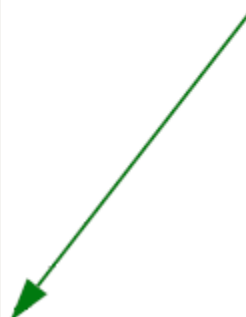
To define the tooltip (for Java applications), complete the following steps:

1. Right-click the window or an object in the window, and select *Properties* . When selecting the window itself, make sure no objects are selected when you right-click.
2. Add or edit the text in the Short Help property in the Properties window as shown in [See Short Help property editing](#). You can insert ASCII [unformatted] text up to 30,000 characters long.
3. Commit the changes before exiting Window Painter to save changes to the short help text. For information about using Window Painter, refer to the *Development Tools Reference Guide* .

### Short Help property editing

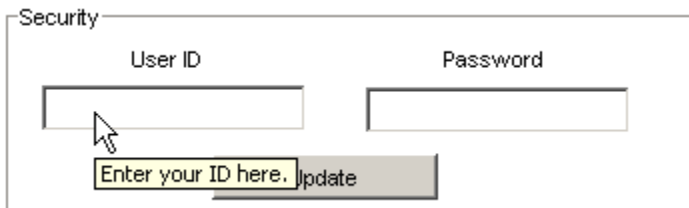
Font	SWISS8
Foreground Color	DEFAULT_COLOR
Form Fit	False
Group	True
Height	23
Help	
HpsID	EDIT_1
Immediate Return	False
Left	33
Link	IVP_SECURITY_USERID_C...
Mandatory	False
Password	False
Resize	False
Short Help	Enter your ID here.
Tab	True

This text appears as tooltip help in the Java client.



Once the application has been prepared and generated in Java, you can have tooltips display in the Java application:

### Tooltip example



## Status Field Help

You can also create status field text for a window in a C application. This displays brief help text for the window or the objects within the window, depending on which item has the focus during execution. When the application is executed, the status field displays the help text that is associated with the selected object. This text is drawn from the property called Short Help and is editable in the Properties window.

To define the status line help (for C applications), complete the following steps:

1. In the window of the C application, add or designate a dedicated and unique edit field to display the help text for the window's controls. This can be an edit field or a multiline edit field. Since there can be only one field to display help for any controls in the window, it also may make sense to add a text label explaining the function of the field.

### Adding dedicated help field



2. Right-click the window and select *Properties* . When selecting the window itself, make sure no objects are selected when you right-click.
3. In the Properties for the window, set the Status Field as the name of the edit field you have added. The Short Help text for any objects will be sent to this field.
4. Right-click any objects on the window and select *Properties* . Add whatever Short Help text that you want to display in the edit field. You can enter ASCII (unformatted) text up to 30,000 characters. See, for example, [See Short Help property editing](#).
5. Commit the changes before exiting Window Painter to save changes to the help text or to the short help text. For information about using Window Painter, refer to the *Development Tools Reference Guide* .

**Short Help property editing**

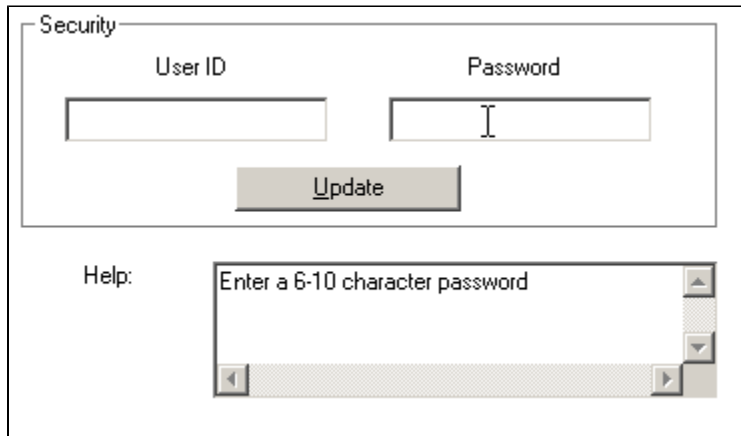
Font	SWISS8
Foreground Color	DEFAULT_COLOR
Form Fit	False
Group	True
Height	23
Help	
HpsID	EDIT_2
Immediate Return	False
Left	195
Link	IVP_SECURITY_PASSWORD_C...
Mandatory	False
Password	True
Resize	False
Short Help	Enter a 6-10 character password
Tab	True
Visible	True
Width	132

This text appears as status field help in the C client.



Once the application has been prepared and generated to C, the Short Help text displays in the field designated as the Status Field in the window when a control has focus:

**C client Short Help**



## F1 Help

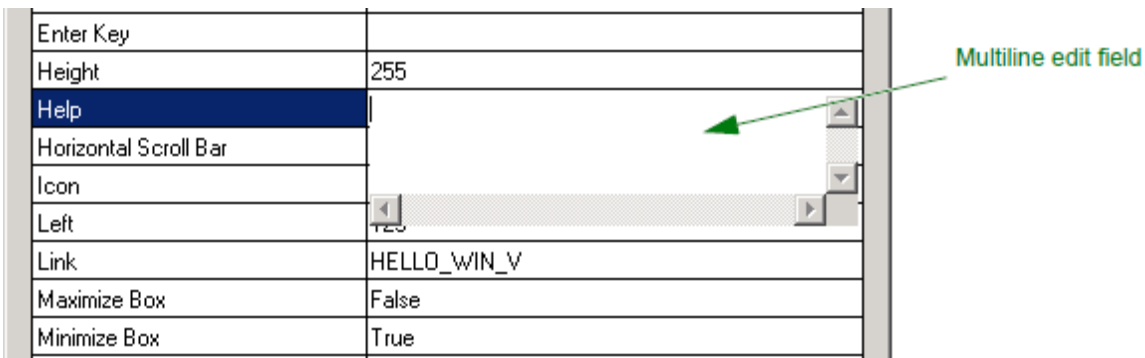
AppBuilder windows and controls provide a Help attribute which can be called via an F1 call in the C client. This Help text appears in a separate Help window. This help system is simple compared to external help applications. However, again, it is built-in and part of the AppBuilder system and repository.

### C Client

Complete the following steps to enter or update AppBuilder's Help text for a window object (the window itself) or for any object in the window:

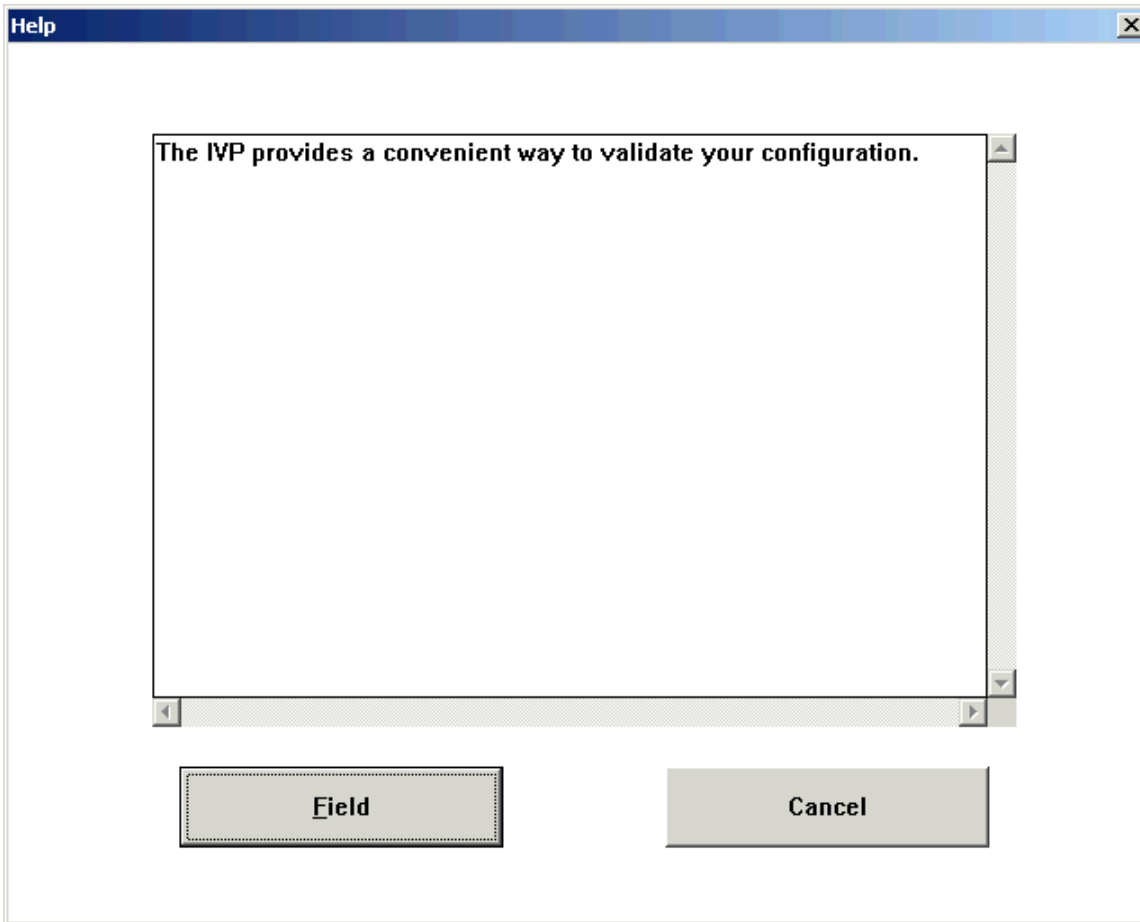
1. Open the window in Window Painter. If the help text is for the window itself, make sure no other objects are selected.
1. Right-click the window and select *Properties* . The window must have a view inserted or a warning shows in the Help field.
2. For an object in the window, right-click the object and select *Properties* .
3. In the property called *Help* , click in the field next to the name Help, and a multiline edit field is displayed. See [See AppBuilder help text](#) for an illustration of the field.
4. Type the help text for the window or window object in this field.
5. When done, press *Enter* .

### AppBuilder help text



This help text is stored with the application in the repository. The help file (.hlp) is not a Windows Winhelp file; it is a proprietary AppBuilder format. As a result, an AppBuilder .hlp file cannot be opened separately from the AppBuilder application. Once the application has been prepared and generated to C, pressing F1 calls the help text for the active object:

### F1 Help in C client



### ***F1 Help at Execution Time***

When the application is executed, each displayed Help pop-up window appears as it does in the Window Painter with two exceptions: The *Apply* push button does not exist, since end users cannot modify the help text, and the *Window help/Field help* push button appears in place of the *Apply* push button.



Execution behavior of Windows help is determined by the environment and the native help system.

AppBuilder defaults to field level help when the user presses the *F1* key. Field help is when an object within the window (rather than the window itself) has focus. The first help pop-up window displayed is always field help.

Window help is when the window itself has focus. After the user has displayed the specific field help for an object, they can view the window object help by selecting the *Window help* push button, or they can get window help by closing the field help pop-up window. Close the window by selecting the *Cancel* push button. To return to the field-level help when the window object help is displayed, select the *Field help* push button. There is only a single push button on each help window, but its name and function changes depending upon whether window help or field help is currently being displayed.

If you do not provide help text for any object in a window or the window itself, when the user presses *F1*, the system displays the message "No Help Available for the Window." If you do not provide help text for a particular object, when the user presses *F1*, the system displays the message "No Help Available for the Object."

## **AppBuilder External**

You can create and build many different kinds of online help or documentation files outside of AppBuilder and then link them to an AppBuilder application. As noted above, the disadvantage to this system is that the content is not stored or linked and updated with the AppBuilder application. An advantage is that there are more options for building and displaying the help system for the application. In general, options include:

- [See Java Client](#)

- [See Windows \(C\) Client](#)
- [See Other Forms of Help](#)

## Java Client

The Java Client only supports F1 Help accessed via the JavaHelp API. For Java applications, you develop the help set files, help map files, and other Java files outside of AppBuilder and point to them in the Help property of the window or window object. Specify the default HelpSet filename for all Java client applications in the appbuilder.ini file. To use the default HelpSet, remove the semicolon at the start of the line, and replace the filename with actual file name. Calling a Java client rule's SetHelpFile() ObjectSpeak method overrides the HELP\_SET\_NAME entry for the rule and all its subrules in the hierarchy.

```
;HELP_SET_NAME=filename.ext
```

When developing distributed applications, AppBuilder provides built-in hooks to JavaHelp that allow you to easily add context-sensitive help to your applications. The following instructions explain how to add context-sensitive help to your applications:

1. Create a HelpSet for the application. The HelpSet file should contain information about the HelpMap, HelpIndex, Table of Contents, and HelpSearch. Refer to the JavaHelp documentation from Sun Microsystems for steps to create a HelpSet.
2. Create the HelpMap file. A HelpMap is an XML file that specifies the mapping between help identifiers (or helpIDs) and HTML files. The name of the file (for example, MAP.jhm) must be the same as that specified in the mapref location. When creating a HelpMap file for AppBuilder applications, you must specify the help IDs as follows:

For windows, the help ID must be the long name of the window.

For user interface objects placed on the window, the ID must be the window long name, followed by a dot, followed by the system identifier (HPSID) of the object.

For each ID you must specify an HTML file contains the help description for the control.

3. Create the HelpIndex file for the application.
4. Create the Table of Contents file.
5. In the appbuilder.ini file (located by default in the <AppBuilder>\JAVART directory) set the HELP\_SET\_NAME to the name of the HelpSet file created in Step 1.



The appbuilder.ini file must be distributed with your Java application in order for help to be accessible to your users. |

A system component call, SET\_HELP\_FILE\_NAME, is required to associate the external JavaHelp helpset with the application. Once this has been done, the F1 key should be active.

For more information on creating JavaHelp, see the documentation for JavaHelp located on the Sun Web site ( <http://java.sun.com> ).

### [Example: JavaHelp Files](#)

#### *Example 1: HelpSet File*

Here is a sample HelpSet file, named HPSApp.hs:

```

<?xml version='1.0' encoding='ISO-8859-1' ?>
<helpset version="1.0">
  <!-- title -->
  <title>Hps NC Client - Help</title>
  <!-- maps -->
  <maps>
    <homeID>MyWindowLongName</homeID>
    <mapref location="Map.jhm"/> <!-- that's the map file name -->
  </maps>
  <!-- views -->
  <view>
    <name>TOC</name>
    <label>Table Of Contents</label>
    <type>javax.help.TOCView</type>
    <data>ncHelpTOC.xml</data>
  </view>
  <view>
    <name>Index</name>
    <label>Index</label>
    <type>javax.help.IndexView</type>
    <data>ncHelpIndex.xml</data>
  </view>
  <view>
    <name>Search</name>
    <label>Search</label>
    <type>javax.help.SearchView</type>
    <data engine = "com.sun.java.help.search.DefaultSearchEngine">
      JavaHelpSearch
    </data>
  </view>
</helpset>

```

#### Example 2: HelpMap File

The following is an example of a HelpMap file that specifies which HTML files contain help information for a window and an edit field and push button located on the window.

```

<?xml version='1.0' encoding='ISO-8859-1' ?>
<map version="1.0">
  <mapID target="MyWindowLongName" url="Mainwindow.html" />
  <mapID target="MyWindowLongName.Edit1" url="edit1.html" />
  <mapID target="MyWindowLongName.ExitBtn" url="exitb.html" />
</map>

```

## Windows (C) Client

You can create Windows (Winhelp) help by compiling a help topic file that contains the help text and a help project file that contains the necessary information to convert the topic file. The help topic file is a rich text format (.RTF) file. To create the help topic file, you can use any editor or word processor that can edit and write .RTF files. Footnotes structure the .RTF file so that it can be compiled into a help file. The rich text format file(s) can then be compiled using the Microsoft Help Compiler.

To make the help context-sensitive, the topic identifier (HPSID) of the object links the object to the compiled help file. In the help topic (.RTF) file, the system identifier is indicated as a keyword footnote. Because the system identifier is displayed to end users when they search the help file, use a meaningful system identifier. You must also make the system identifier and corresponding keyword unique to prevent the application from calling the wrong help topic. See the documentation on the Microsoft Help Compiler for details on the help topic (.RTF) file structure.

The required help project (.HPJ) file is invoked when the Help Compiler runs, and it contains the information that the compiler needs to convert the files into a binary help resource file. The Help Compiler creates the binary help resource file from the topic files listed in the help project file. The compiled help has the same file name as the project file but with an .HLP extension. Refer to the documentation about the Microsoft Help Compiler for details about compiling help files.

The compiled help (.HLP) file is then linked to the application through the use of two AppBuilder system components or through the use of the command line parameter `-h help_file_name` in the file master.exe. Refer to the appendix in the *Deploying Applications Guide* for a summary of the valid run-time parameters.

### System Components for C Client Online Help

The AppBuilder environment supports three system components to link to an external help system:

**SET\_HELPFILE\_NAME** – Use this component to specify the .HLP file to which calls are subsequently made with the SHOW\_HELP\_TOPIC component.

**SHOW\_HELP\_TOPIC** – Use this component to display the help panel for a specific object in a window, referenced by its system identifier (HPSID). The HELP\_KEYWORD field used in the SHOW\_HELP\_TOPIC component call is usually a text string consisting of the system identifier (HPSID) of the Window Painter object—for example, a push button—about which a help panel is to be displayed.

**HPS\_SET\_HELP\_TOPIC** – Use this component to set a topic for an entire window rather than a field within the window. This effectively makes the help less context sensitive. In doing so, each time *F1* is pressed, the help panel for the entire window is displayed, taking the focus off a specific field.

For more details about these system components, refer to the *System Components Reference Guide*.

## Other Forms of Help

Depending upon your application environment and development tools, other forms of online help may also be available and worth considering. Microsoft HTML Help (both 1.x and 2.x), WebHelp, Oracle Help for Java, and XML are all possible options. They all must be developed and stored externally and then linked to your AppBuilder application.

There are numerous independent help authoring tools (HATs) available, most of which provide the ability to generate multiple forms of online documentation and user assistance products.

## Enabling Application Help

You can specify only one type of help system for an application. Use the appropriate variable and path name in the NC section of the appbuilder.ini file (for Java applications) or the AE Runtime section of the hps.ini file (for C applications) on the workstation.



This variable must also be set correctly on all the production workstations on which the generated application runs.

## Data Type Support

### Data Type Support

The AppBuilder DEC and PIC data types provide ways to represent numbers. Data type support includes the following:

- [DEC Value Support](#)
- [PIC Value Support](#)

The AppBuilder DATE data type provides a mechanism for storing and managing four-digit year dates within an AppBuilder development application. The DATE variable has a length of 4 bytes. The value of the DATE variable is the number of days since the date of origin (January, 1, 0000). The DATE data type also provides accurate support for leap years. DATE data type support addresses these issues:

- [Short-form Date Support](#)
- [Repository Migration Date Support](#)

The AppBuilder CHAR, VARCHAR, MIXED, and DBCS data types provide character data type support. [See Character Data Support](#) discusses considerations for using these fields in C user components.

### DEC Value Support

DEC variable is declared as *DEC(length,scale)*, where *length* is the length of the data item and *scale* is the number of decimal places to the right of the decimal point. Data is represented from right to left (low-order bytes to high-order bytes). The maximum size for a DEC field on all platforms is 31.

- The fractional portion (if any) as a series of digits (0-9) for the length of the fractional portion
- The absolute value of the integer portion or zero
- A minus sign (if appropriate) immediately to the left of the integer portion
- Blanks left-filled for the remainder of the field

The following topics are discussed in this section:

- [Decimal Point Position](#)
- [Valid Characters](#)
- [Format of DEC Value](#)

### Decimal Point Position

The position of the decimal point in the DEC value is statically defined, so the length and scale are not stored together with a decimal value. Length and scale are passed to the code generator runtime functions when necessary.

## Valid Characters

A DEC field is represented as a C character array (not necessarily null-terminated, since its length is always known). The size of the array is the length of the DEC field plus one. Valid characters in this array are minus sign (-), digits (0, ..., 9) and space ( ).

## Format of DEC Value

All decimal values regardless of their origin should have the following unambiguous normalized representation.

### DEC Value Format

```
<DEC value> ::= ' '*<IntegerPart><DecimalPart>
<IntegerPart> ::= {'-'|' '}{'0'|<NonZeroDigit><Digit>*}
<DecimalPart> ::= <Digit>*
<NonZeroDigit> ::= '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
<Digit> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

and where:

- number of bytes in <IntegerPart> is equal to (length - scale)
- number of bytes in <DecimalPart> is equal to scale



The first significant digit of <IntegerPart> is the first non-zero digit in it, or '0' in its right-most position, if it is zero. The '0' must be present, if <IntegerPart> is zero.

Representation of the DEC must satisfy the following conditions:

- <IntegerPart> must be padded with blanks until the first significant digit
- <DecimalPart> must be padded with zeroes until its end

### Example: DEC Value Code Sample

```
" -0050" represents -0.05
" 1300" represents 1.3
" 0004" represents 0.004
" 130000" represents 130.0
" 3000" represents 3.0
" 0000" represents 0
" 13 " is incorrect, because trailing spaces in the integer part are not allowed
"+ 1300" is incorrect, '+' is not allowed
" 130" is incorrect, integer part must have at least one digit
" 00130" is incorrect, leading zeros in the integer part are not allowed
" 013 " is incorrect, trailing blanks in the decimal part are not allowed
```

## PIC Value Support

PIC variable is declared using a storage pattern string or a storage picture format. This is described in the *Rules Language Reference Guide*.

The size of a picture field is determined by the Field picture-storage property. The picture property for a Field object in the repository is 30 characters long (Field picture-storage and Field picture-display). However, in the Rules Language, one should be able to use a longer local PIC field for data manipulations within the rule code. The maximum size for a PIC field on all platforms is 31.



You cannot create a table containing a PIC field. PIC field is a display-orientated data type, not a database field type, so it is not supported in table.

The following topics are discussed in this section:



- [See Valid Characters](#)
- [See Format of PIC Value](#)

## Valid Characters

PIC value is represented by C character array (not necessarily null terminated). Valid characters in this array are digits '0',..., '9', sign plus '+', and sign minus '-'.

## Format of PIC Value

The internal representation of PIC value must be formatted according to its storage picture in the following way:

- For each '9' (digit position) in the storage pattern, there must be a digit in the internal representation.
- For 'S' (sign position) in the storage pattern, there must be a sign in the internal representation. If no sign is declared, no byte is reserved for it.
- For 'V' (decimal point position) in the storage pattern, no byte is reserved in the internal representation.

Length of PIC field is defined as the number of '9's in the storage picture, and scale is defined as the number of '9's after the 'V' sign. Scale is 0 if 'V' is not present in the storage picture.



In Java and OpenCOBOL, it is now possible to declare picture with a trailing sign. This is a different data sub-type, and it is possible to redefine its internal character representation using a custom data converter. For more details, see *Rules Language Guide, PIC with trailing sign* subchapter.

## PIC Value Format

```
<Signed PIC value> ::= {'-'|'+'}<IntegerPart><DecimalPart>
<Unsigned PIC value> ::= <IntegerPart><DecimalPart>
<IntegerPart> ::= <Digit>*
<DecimalPart> ::= <Digit>*
<Digit> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

where:

- number of bytes in <IntegerPart> is equal to (length - scale)
- number of bytes in <DecimalPart> is equal to scale



For runtime functions to correctly interpret PIC value, it must be clear whether the value is signed/unsigned (that is, if its storage picture contains S or not), its length (the number of 9s and Ss in storage pattern, that is, the size of the C array allocated for the value) and scale.

## Example: PIC Value Code Sample

Consider PIC **s999v999** of 123456

```
"-000050" represents -0.05
"+001300" represents 1.3
"+000004" represents 0.004
"+130000" represents 130.0
"+003000" represents 3.0
"+000000" represents 0
"1234567" is incorrect
" 123456" is incorrect; sign must be present
"+ 1100" is incorrect; blanks are not allowed
```

Consider PIC **999v999** of 123456

```
"000050" represents 0.05
"001300" represents 1.3
"000004" represents 0.004
"130000" represents 130.0
"003000" represents 3.0
"000000" represents 0
"+23456" is incorrect; sign is not allowed
" 1100" is incorrect; blanks are not allowed
```

## Short-form Date Support

AppBuilder provides the option of a short-form DATE data type for use by AppBuilder developers. The short-form of the DATE data type provides a method for adding the century portion to short dates. The short-form DATE data type is compensated for in the following cases:

- **DATE Input Field** is used when the developer wants the end user to input the short-form of the date into an edit field in a window.
- **DATE Data Type** is used if character fields exist without the century information. The character fields can (without century information) be converted into DATE data types in the Rules Language. These character fields without century information are under application control and can be stored in databases or even generated fields.

### DATE Input Field

In a window, the AppBuilder DATE input field requires the long form of the date. As a convenience to the end users, they can use the AppBuilder applications to type a short form of the date into the DATE input field. AppBuilder runtime support adds the century portion of the year to the date and displays the four-digit date in the input field.

Thus, the end users can verify the century being used in the long form of the date after the application has been built. The DATE input field should always be defined large enough to display the long form of the date so that the end users can easily see the added century portion of the year. The system uses a sliding window algorithm to determine the century to be added to the short date so that you can identify the number of years in the future to define the dividing line between the past and the future.

For example, if the current year is **2001** and you define the "number of years in the future" to be **30**, then:

All years entered between **02** and **31** are interpreted as future dates, and are listed as **2002** through **2031**.

All years entered between **32** and **00** are interpreted as past dates, and are listed as **1932** through **2000**.

The year **01** is interpreted as **2001**. The current year (01) is always interpreted as being in the current century (that is, **2001** in this case).

The sliding window limits you to a date range of 100 years. In the example above, it is not possible to represent years earlier than 1931 or later than 2031 using short-form dates.

#### Setting Years in the Future

Workstation-based AppBuilder applications support the sliding window for workstation applications. Use the workstation parameter `YEARS_IN_THE_FUTURE` in the [AE Runtime] section of the hps.ini file, to define the number of years in the future. The default is **30**. For Java applications, you must set the value in the [NC] section of the appbuilder.ini file.

#### Support in 3270 Converse-Based Applications

The system uses the sliding window to determine the century portion of short form dates for 3270 Converse-based applications. The sliding window for the host is fixed at **30** years into the future. To confirm that you have the sliding window support in the AppBuilder system, verify that the module **HPEUE11** has a date and time stamp of **12/07/95 14.27** or greater.

#### Recommendation on Input Field Length

Although the user can enter the short form of the date into the DATE input field, if the field is not large enough to display the full long form of the date, then the user might have to scroll the DATE input field to see the added century portion of the year. If the end users require visual feedback of entered dates, make sure that the date input field is long enough to display the full long date format.

### DATE Data Type

The DATE data type provides support to convert a two-digit year field into an AppBuilder long-form DATE. For two-digit year field conversion to a long form of the date, the century determination between the host and the workstation are performed with different algorithms. The century determination operates as follows:

For **workstation-based applications**, use **19** for the century in converting two-digit years to the long form of the DATE.

For **mainframe-based applications**, use a fixed century determination point of **29**. A two-digit entry of **30** maps into a four-digit year of **1930**, and a two-digit entry of **29** maps into a four-digit year of **2029**. To confirm that you have the sliding window support in AppBuilder system, verify that the module **HPEUE11** has a date and time stamp of **12/07/95 14.27** or greater.

#### Recommendation for the Four-Digit Year

Since the conversion of the two-digit year to the four-digit year are not readily visible to the end user and the results of the conversion might not be observed until a later time, your applications should manage both the century portion of the year and the two-digit year component to ensure a complete and accurate four-digit year.

## Repository Migration Date Support

In the AppBuilder development environment, the repositories maintain only the last two YY digits of the date in their audit fields (for example, object create date or object last modify date). Because most comparisons are based on the equality test, as opposed to greater than or less than test, this is not a concern; however, date support becomes an issue to consider when performing:

- [Enterprise Repository Migration](#)
- [Workgroup Repository Migration](#)

### Enterprise Repository Migration

For the Enterprise (mainframe) Repository, migrations are managed using the Migration Selective Export tool. When the repository migration spans a century, the migration must be handled in two processes.

For example, using the Migration Selective Export date format of **YY/MM/DD**, the date range of 99/10/01 - 00/03/31 (October 1, 1999 through March 31, 2000) would be replaced with these two ranges:

- 99/10/01 – 99/12/31 (October 1, 1999 through December 31, 1999)
- 00/01/01 – 00/03/31 (January 1, 2000 through March 31, 2000)

These two range statements need to be linked with **AND** in the **Op** field of the first date range statement. The Migration Selective Export evaluates the object last modify date.

For more information, refer to the *Enterprise Migration Guide*.

### Workgroup Repository Migration

For the Workgroup Repository, the migrations are managed similarly to those of the Enterprise Repository. In Windows, the migration function of the Migration Import of the Repository Administration tool is used.

For example, to migrate all objects between 99/10/01 - 00/03/31 (October 1, 1999 through March 31, 2000), the migration is handled in two processes:

The first migration would specify **After** 99/10/01 (October 1, 1999 through December 31, 1999).

The second migration would specify **Before** 00/03/31 (January 1, 2000 through March 31, 2000).

The migration process evaluates the object date maintained field.



If you have developed unique repository utility applications using the Freeway APIs, you should provide support for handling the two digit year field, as the repositories only store the last two YY digits of the year.

For more information, refer to the *Repository Administration Guide for Workgroup and Personal Repositories*.

## Character Data Support

The AppBuilder character fields, CHAR, VARCHAR, MIXED and DBCS, are not null-terminated as string fields normally are in C. CHAR, VARCHAR, MIXED, and DBCS fields are always padded with spaces to their specified maximum sizes for DBCS fields, the padding character is a DBCS space.

If, in a C user component, you need to convert an AppBuilder character field to a C-style string, use `memcpy()` to copy the field to a new area (1-byte longer) before adding a null-terminator and optionally trimming any trailing spaces. *Do not* simply try modifying the data in the field in the passed view structure. For example:

### Example: Using `memcpy()`

```
char *str;
int i;
str = malloc( INPUT_VIEW_NAME->V_CHAR_FIELD + 1 );
memcpy( str, INPUT_VIEW_NAME->V_CHAR_FIELD, sizeof(INPUT_VIEW_NAME->V_CHAR_FIELD ) );
str[sizeof( INPUT_VIEW_NAME->V_CHAR_FIELD )] = '\0';
for ( i = strlen( str ) - 1; i >= 0 && str[i] == ' '; str[i] = '\0', i-- );
```

## Events Supported in C

This chapter lists the events used in event-driven processing of rules in C. For information about events in Java, refer to the *ObjectSpeak Reference Guide*.

Event-driven processing in C uses the following events. **NA** indicates that the given field is not used for an event and hence is undefined (or not applicable). They are listed alphabetically.

- [Asynch Event](#)
- [Check Box Click](#)
- [Child Process Terminated](#)
- [Close Window with System Menu](#)
- [Detached Rule Reactivated](#)
- [Double-Click or Tab Out of Field](#)
- [Double-Click or Tab Out of List Box Field](#)
- [Hot Spot Click](#)
- [Menu Selection](#)
- [Press Enter Key](#)
- [Push Button Click](#)
- [Radio Button Click](#)
- [Scroll Off Bottom of List Box](#)
- [Scroll Off Top of List Box](#)
- [Scroll Out of Range in List Box](#)
- [Time Out](#)
- [Undefined Event](#)

### Asynch Event

<b>Event Description:</b>	<b>Asynch event</b>
EVENT_TYPE:	ASYNC_EVENT
EVENT_NAME:	Name of target rule
EVENT_SOURCE:	NA
EVENT_QUALIFIER:	NA
EVENT_VIEW:	System ID of target rule input view
EVENT_PARAM:	NA

### Check Box Click

<b>Event Description:</b>	<b>Check box click (if immediate return)</b>
EVENT_TYPE:	INTERFACE_EVENT
EVENT_NAME:	HPS_IMMEDIATE_RETURN
EVENT_SOURCE:	Check box system identifier (HPSID)
EVENT_QUALIFIER:	NA
EVENT_VIEW:	NA
EVENT_PARAM:	NA

### Child Process Terminated

<b>Event Description:</b>	<b>Child process terminated</b>
EVENT_TYPE:	SYSTEM_EVENT
EVENT_NAME:	HPS_CHILD_END
EVENT_SOURCE:	Name of the rule
EVENT_QUALIFIER:	Instance name of rule (if applicable)
EVENT_VIEW:	NA
EVENT_PARAM:	NA

## Close Window with System Menu

<b>Event Description:</b>	<b>Close a window with the system menu (if the close return code is set)</b>
EVENT_TYPE:	INTERFACE_EVENT
EVENT_NAME:	HPS_WIN_CLOSE
EVENT_SOURCE:	Close return code
EVENT_QUALIFIER:	NA
EVENT_VIEW:	NA
EVENT_PARAM:	NA

## Detached Rule Reactivated

!Event Description:	<b>Detached rule reactivated</b>
EVENT_TYPE:	SYSTEM_EVENT
EVENT_NAME:	HPS_DETACH
EVENT_SOURCE:	NA
EVENT_QUALIFIER:	NA
EVENT_VIEW:	Name of view
EVENT_PARAM:	NA

## Double-Click or Tab Out of Field

<b>Event Description:</b>	<b>Double-click or tab out of any field after data entry (if immediate return)</b>
EVENT_TYPE:	INTERFACE_EVENT
EVENT_NAME:	HPS_IMMEDIATE_RETURN
EVENT_SOURCE:	Field system identifier (HPSID)
EVENT_QUALIFIER:	Column system identifier (HPSID) for a multicolumn list box only; otherwise, NA
EVENT_VIEW:	NA
EVENT_PARAM:	NA

## Double-Click or Tab Out of List Box Field

<b>Event Description:</b>	<b>Double-click or tab out of any list box field after data entry (if immediate return)</b>
EVENT_TYPE:	INTERFACE_EVENT
EVENT_NAME:	HPS_IMMEDIATE_RETURN
EVENT_SOURCE:	List box system identifier (HPSID)
EVENT_QUALIFIER:	NA
EVENT_VIEW:	NA
EVENT_PARAM:	Occurrence number of scroll box position

## Hot Spot Click

<b>Event Description:</b>	<b>Hot spot click</b>
EVENT_TYPE:	INTERFACE_EVENT

EVENT_NAME:	HPS_HS_CLICK
EVENT_SOURCE:	Hot spot system identifier (HPSID)
EVENT_QUALIFIER:	NA
EVENT_VIEW:	NA
EVENT_PARAM:	NA

## Menu Selection

<b>Event Description:</b>	<b>Menu selection</b>
EVENT_TYPE:	INTERFACE_EVENT
EVENT_NAME:	HPS_MENU_SELECT
EVENT_SOURCE:	Menu choice system identifier (HPSID)
EVENT_QUALIFIER:	NA
EVENT_VIEW:	NA
EVENT_PARAM:	NA

## Press Enter Key

<b>Event Description:</b>	<b>Press Enter key</b>
EVENT_TYPE:	INTERFACE_EVENT
EVENT_NAME:	HPS_PB_CLICK No Enter key: Press <b>Enter</b> with focus on push button. HPS_PB_CLICK Push button: Press <b>Enter</b> with focus anywhere on screen. HPS_WIN_ENTER Non-Push button: Press <b>Enter</b> with focus anywhere on screen.
EVENT_SOURCE:	Enter return code or push button system identifier (HPSID)
EVENT_QUALIFIER:	NA
EVENT_VIEW:	NA
EVENT_PARAM:	NA

## Push Button Click

<b>Event Description:</b>	<b>Push button click</b>
EVENT_TYPE:	INTERFACE_EVENT
EVENT_NAME:	HPS_PB_CLICK
EVENT_SOURCE:	Push button system identifier (HPSID)
EVENT_QUALIFIER:	NA
EVENT_VIEW:	NA
EVENT_PARAM:	NA

## Radio Button Click

<b>Event Description:</b>	<b>Radio button click (if immediate return)</b>
EVENT_TYPE:	INTERFACE_EVENT
EVENT_NAME:	HPS_IMMEDIATE_RETURN
EVENT_SOURCE:	Radio button system identifier (HPSID)

EVENT_QUALIFIER:	NA
EVENT_VIEW:	NA
EVENT_PARAM:	NA

### Scroll Off Bottom of List Box

<b>Event Description:</b>	<b>Scroll off the bottom of a list box</b>
EVENT_TYPE:	INTERFACE_EVENT
EVENT_NAME:	HPS_LB_BOTTOM
EVENT_SOURCE:	List box system identifier (HPSID)
EVENT_QUALIFIER:	Column system identifier (HPSID) for a multicolumn list box only; otherwise, NA
EVENT_VIEW:	NA
EVENT_PARAM:	Occurrence number of bottom

### Scroll Off Top of List Box

<b>Event Description:</b>	<b>Scroll off the top of a list box</b>
EVENT_TYPE:	INTERFACE_EVENT
EVENT_NAME:	HPS_LB_TOP
EVENT_SOURCE:	List box system identifier (HPSID)
EVENT_QUALIFIER:	Column system identifier (HPSID) for a multicolumn list box only; otherwise, NA
EVENT_VIEW:	NA
EVENT_PARAM:	Occurrence number of top

### Scroll Out of Range in List Box

<b>Event Description:</b>	<b>Scroll out of range in a list box</b>
EVENT_TYPE:	INTERFACE_EVENT
EVENT_NAME:	HPS_LB_OUTRANGE
EVENT_SOURCE:	List box system identifier (HPSID)
EVENT_QUALIFIER:	Column system identifier (HPSID) for a multicolumn list box only; otherwise, NA
EVENT_VIEW:	NA
EVENT_PARAM:	Occurrence number of scroll box position

### Time Out

<b>Event Description:</b>	<b>Time out (set by the system component SET_WINDOW_TIMEOUT)</b>
EVENT_TYPE:	INTERFACE_EVENT
EVENT_NAME:	HPS_WIN_TIMEOUT
EVENT_SOURCE:	Name of the window
EVENT_QUALIFIER:	NA
EVENT_VIEW:	NA
EVENT_PARAM:	NA

## Undefined Event

<b>Event Description:</b>	<b>Undefined event</b>
EVENT_TYPE:	SYSTEM_EVENT
EVENT_NAME:	HPS_SYS_UNDEFINED
EVENT_SOURCE:	NA
EVENT_QUALIFIER:	NA
EVENT_VIEW:	NA
EVENT_PARAM:	NA