Screen Orchestrator Guide

Version 2005, Rev. A

July 2005

Siebel Systems, Inc., 2207 Bridgepointe Parkway, San Mateo, CA 94404

Contents

1 What's New in This Release

What's New in Screen Orchestrator Guide, Version 2005, Rev. A

Table 1 lists changes in this version of the documentation to support release 2005 of the software.

Table 1. What's New in Screen Orchestrator Guide, Version 2005, Rev. A

Topic	Description
Customizing the Screen Orchestrator on page 22	New topic. Summarizes the ways in which you can customize the Screen Orchestrator.
The Screen Orchestrator Toolbar on page 26	New topic. Describes the tools available in the toolbar of the Screen Orchestrator.
Drawing Chart Notes on page 33	Updated to include information about the Edit Event Details and Edit Transition Details buttons.
Resizing the Statechart Window on page 42	New topic. Describes how to resize the statechart window.
Printing Statecharts on page 44	Updated to describe the statechart printing functionality in more detail.
Configuring WAR File Properties for Statecharts on page 50	Updated with a description of the Libraries Dir field in the Confirm War Properties screen.
Custom Controllers and Guard Conditions on page 60	Heading changed to reflect the fact that you can write custom guard condition classes as well as custom controller classes.
Adding a New Controller to the Screen Orchestrator on page 74	Updated to add com.eontec.statemachine.helpers.MultipleRequestController to the list of controllers.
Defining the Validation Rules on page 123	Updated to describe the validation rules fields in the Specify validation rule for input requirement screen.
About Timing Points on page 129	New topic. Describes the purpose of timing points in the state machine framework code.
Timing Points in the State Machine on page 129	New topic. Describes the use of timing points in the state machine and includes a corrected sample of how timing points are coded.

Additional Changes

The content of this guide was rewritten and restructured to implement Siebel style.

What's New in Screen Orchestrator Guide, Version 2005

Table 2 lists changes in this version of the documentation to support release 2005 of the software.

Table 2. What's New in Screen Orchestrator Guide, Version 2005

Topic	Description
Chapter 11 Forward Engineering	New Chapter. Describes the Forward Engineering functionality, which allows you to propagate design model changes to existing statecharts.

2 Screen Orchestrator Overview

This chapter provides an overview of the Screen Orchestrator as well as statechart and state machine concepts. It includes the following topics:

About the Screen Orchestrator

The Screen Orchestrator is a tool that allows you to design and implement an application using statechart principles. The Screen Orchestrator allows you to visually draw a statechart representation of your proposed application and to specify interactively the actual processes and state types that are used by the application when it runs. The tool allows you to deploy the visually-drawn statechart to a live HTTP server or application server where the application is run and controlled by a state machine. The state machine reads the deployed statechart and uses it to control the actual application.

Understanding statechart principles and notation is a prerequisite to using the Screen Orchestrator correctly. Therefore, it is extremely important that you read the following topics to get a basic understanding of statecharts, their notation, and the state machine.

About Statecharts

The Unified Modeling Language (UML) definition of a statechart diagram is as follows:

A statechart diagram represents the behavior of entities capable of dynamic behavior by specifying its response to the receipt of event instances. Typically, it is used for describing the behavior of class instances, but statecharts may also describe the behavior of other entities such as use-cases, actors, subsystems, operations, or methods.

The key concept in understanding why statecharts are used to represent user interfaces is the fact that statechart diagrams are capable of handling or modeling dynamic behavior through events. Users interact with a user interface dynamically through events. Statecharts are therefore ideally suited to describing how a user interacts with a user interface.

For example, consider the following user interaction with a login screen in a user interface:

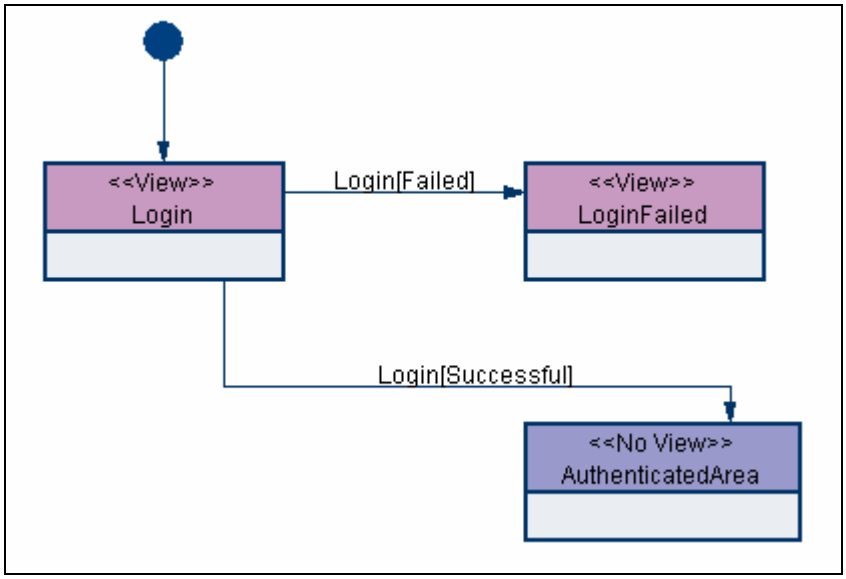
1. The user enters the username and password.
2. The user clicks a login button to activate the login request.

If the login is successful, the user is allowed into the rest of the system. If the login fails, the user is taken to the login-failed screen.

Clicking the login button is an event that must occur for the request to be processed. The event has two possible outcomes in this instance: it is either successful, or it fails. The statechart representation of this user interaction is shown in Figure 1 (the notation is explained in Statechart Notation Explained on page14).

The statechart in Figure 1 shows how a login user interface can be modeled in statechart notation. When the user interacts with a screen, an event is created. An event is created when a user clicks a button, or selects a radio button, or submits a form, or for whatever action you want the user to take. In terms of the Screen Orchestrator, states are often what the user sees on the computer screen (the view). In the login example, there are three states or screens, the Login screen itself, the LoginFailed screen, and the AuthenticatedArea, where one or more screens and hence states, can exist.

Figure 1. Login Statechart Diagram



In the Screen Orchestrator, you therefore use statecharts to capture the user's interaction with the user interface through the modeling of the events that describe the system. This book concentrates primarily on using statecharts for the design and implementation of user interfaces. However, you can also use statecharts for other purposes. For example, you can use a statechart to capture the flow of a process on the server-side of the application. You can use the statechart in long-lived multiple transitions to route a process from one state of the transition to the next.

About the State Machine

While a statechart is the representation of the modeled user interaction of a user interface, the state machine is a framework that uses that statechart to control the real user interface. The state machine framework is based on an open source project, the jstatemachine (www.jstatemachine.org). In Siebel Retail Finance, this framework has been extended to be aware of Retail Finance processes and is part of the MCA Services.

The state machine framework is loaded and runs on any HTTP server that supports Java servlets and Java Server Pages (JSPs). The state machine reads a statechart produced by the Screen Orchestrator and uses that statechart to control the real user interaction coming from the user interface. The key thing to remember here is that the Screen Orchestrator constructs the statecharts, while the state machine loads the statechart and uses it to control the actual user interface.

Why Statecharts and the State Machine Are Used

Any large system's user interface today is normally designed using a modern integrated development environment (IDE). While such tools are extremely powerful in building complex user interfaces, user interface software often has the following characteristics:

- The code can be difficult to understand and review thoroughly.
- The code can be difficult to test in a systematic and thorough way.
- The code can contain bugs even after extensive testing and bug fixing.
- The code can be difficult to enhance without introducing unwanted side effects.
- The quality of the code tends to deteriorate as enhancements are made to it.

Despite the obvious problems associated with user interface development, little effort has been made to improve the situation. However, the use of statecharts to specify the flow and control of the user interface is a major step in improving this situation. The user interface design can now be captured, understood, and interpreted by existing and new developers of the system. You now have the visual record of the flow of control of the system, and you can see the side effects of any change on the user interface.

The design of the user interface is too often left almost entirely to the developer and their understanding of the use cases. Greater design work needs to be done on the user interface so that the user interface can be more easily understood, developed, and maintained. Statecharts can play a key part in achieving this design work.

The state machine is an extension of using statecharts. If the user interface is described using a statechart, then why not use the actual statechart within the user interface application to maintain control of the actual system? Any changes in the statechart are then automatically reflected back in the actual application. The state machine does precisely that; it takes the actual statechart and controls the application directly with the statechart. The user interface developer is then free to concentrate on building and creating the views for the system.

Statecharts and the state machine enforce the Model-View-Controller (MVC) programming model.

- The Controller of the user interface is the statechart that was drawn, while the state machine is the run-time environment for that statechart.
- The Views are the views of the application seen by the user. The developer can create views for the application that contain view code only. In the application, views are implemented by Java Server Pages (JSP) or Swing panels.
- The Model is defined by the input parameters to the states, events, and transitions. Later topics describe what is meant by input parameters and the maintaining of the Model details used in the statecharts).

The state machine and the Screen Orchestrator are aware of Retail Finance processes, so they can invoke the processes, and more importantly interpret their responses so that you require little or no control code. This awareness leads to a user interface system that is highly controlled and whose side effects are easily understood and changed as the system grows and you make enhancements, thus avoiding the problems sometimes associated with user interface designs.

For more information about the state machine framework, see the MCA Services API documentation.

Statechart Notation Explained

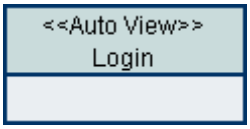
Statechart notation essentially consists of two representations: states and state transitions.

States are represented in the Screen Orchestrator by a rectangular box, while an arrow represents a state transition. Events, another important statechart element, are not pictorially represented, but they are identified in a statechart in the labels attached to state transitions.

States

The statechart that you define for the user interface is merely the representation of the possible states of the interface. A state, in the case of the state machine and the Screen Orchestrator, represents what the user sees on the computer screen. In terms of the Screen Orchestrator, a screen is termed a view. A standard state drawn on a statechart in the Screen Orchestrator is shown in Figure 2.

Figure 2. A Standard State



Every state has a title and subtitle. In Figure 2, <<*Auto View*>> is a title, and *Login* is a subtitle. The subtitle on each state indicates the name of the state or view. The title always indicates the type of state that the state represents. The Screen Orchestrator provides three basic state types:

- **An Auto View state.** This state indicates that a view is represented by the state but that no current view is available to be attached to the state. The state machine can generate an automatic view for this state dependent on the state input parameters and the events leaving the state.
- **A View state.** This state indicates that a view is represented by the state and that an actual view can be attached to the state. When the state machine runs the application, the attached view is displayed to the user.
- **A No View state.** This state is often used to indicate that the state is a parent state (although an AutoView and View state can also be parent states). Parent states are used to split the user interface into subsystems. The No View state is also used to represent server-side states because these states do not represent any particular view of the system.

Each state type available in the Screen Orchestrator is shown in Figure 3, Figure 4, and Figure 5, each of which shows one of these states.

The states are color-coded for easy identification.

Figure 3. An Auto View State

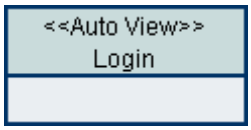


Figure 4. A View State

Figure 5. A No View State

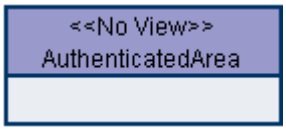
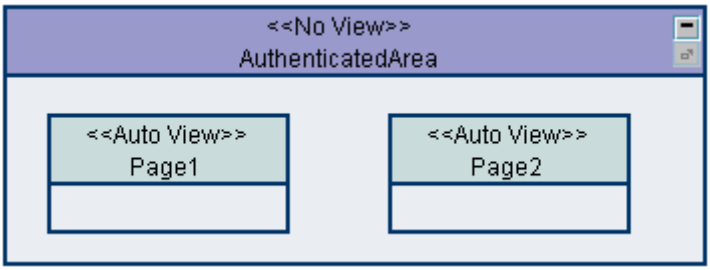


Figure 6. Parent and Child States



Parent and Child States

You can add a state to other states, so that the added state becomes a child of the enclosing state. Figure 6 shows the AuthenticatedArea state as a parent state of the child states Page1 and Page2.

States can inherit events and transitions from their parent states. That is, an event or transition available from the parent state is also available from its child states, with the exception of the case where a child state has an event of the same name.

Events

When the user interacts with the screen, an event is initiated. An event might be clicking a button, clicking a radio button, submitting a form, or any action you want the user to take. An event is identified in the state machine by its source and name. In the example shown in Figure 1, clicking the login button on the login screen is an event being initiated.

State Transitions

On a statechart, arrows represent the transitions between states. Each transition has a label with the following syntax (all three parts are optional):

Event [*Guard condition*]/*Action*

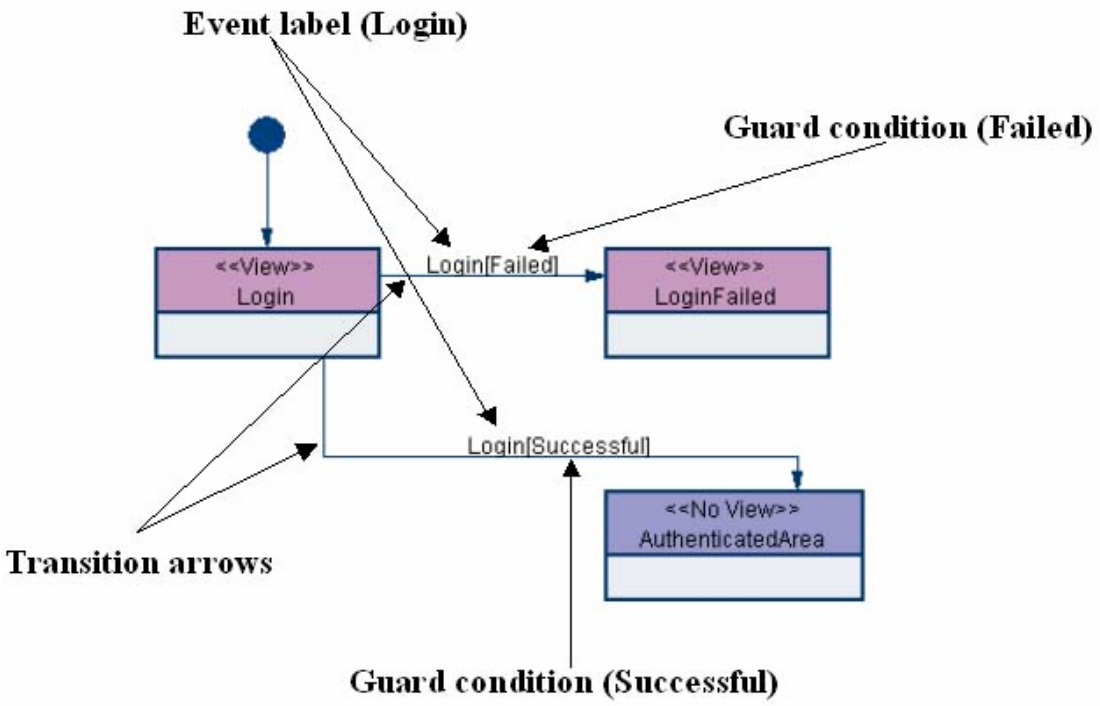
For example, in the transition Login[Successful]/StoreUserId shown in Figure 8:

- The event is Login.
- The guard condition is Successful.
- The action is StoreUserId.

The event is the user action that fires the transition.

Each transition can be guarded by a condition or set of mutually exclusive conditions that must return true for a particular transition to be followed. After the event occurs, the guard condition of each transition possible for that event is tested. One of the conditions returns true and the state machine follows that transition to the resultant state. The state machine then informs the user interface and the display is updated to show what is proper for that state. Figure 7 illustrates the login event and its transitions.

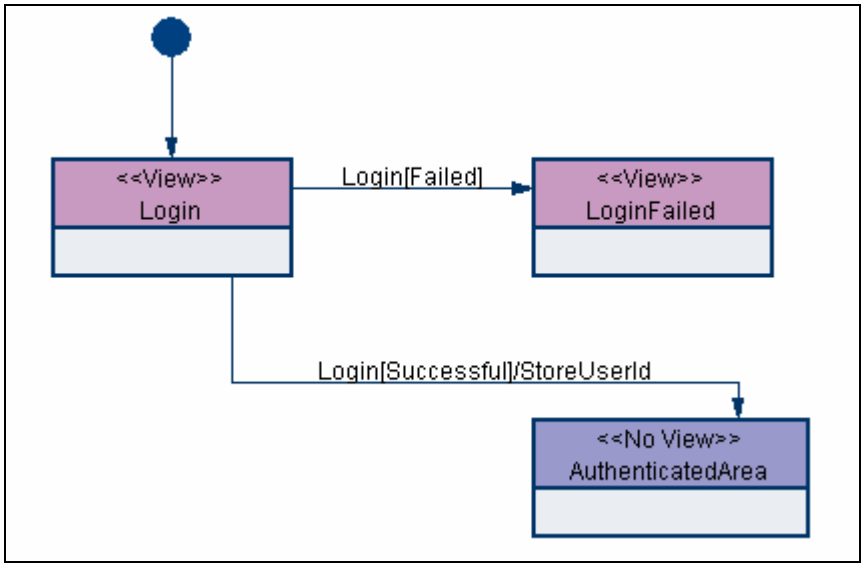
Figure 7. Identifying Events and Transitions on a Statechart



Actions (also termed *side effects*) are associated with transitions and are considered to be processes that can occur as a result of a transition. In the login example, you can extend the Login[Successful] transition to include an action, StoreUserId (as shown in Figure 8).

When the user clicks the login button, the Login event is fired. The login results are tested to see whether the login was successful. If the login was successful, the Login[Successful]/StoreUserId transition is followed. The action associated with this transition is to take the user's user ID and store it in the user's HTTP session.

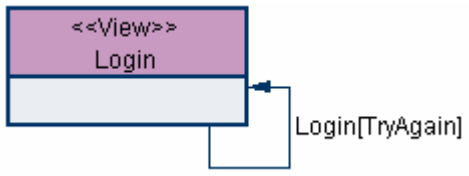
Figure 8. Extending the Login Example to Include a Transition Action



Self-Transition

A self-transition behaves exactly as a normal transition does except that its start state and end state are the same. Figure 9 illustrates how a self-transition is drawn in the Screen Orchestrator.

Figure 9. Self Transition



You use a self-transition when you want to send the user back to the currently displayed screen, if a certain guard condition is met. In the example in Figure 9, when the Login event is fired, and if the guard condition TryAgain is met, the Login screen is redisplayed by the state machine.

Pseudo-States

A number of pseudo-states are also available within statechart notation. Pseudo-states represent special types of state that indicate very specific types of behavior when included on the statechart. The pseudo-states are described in the following subtopics

Initial State

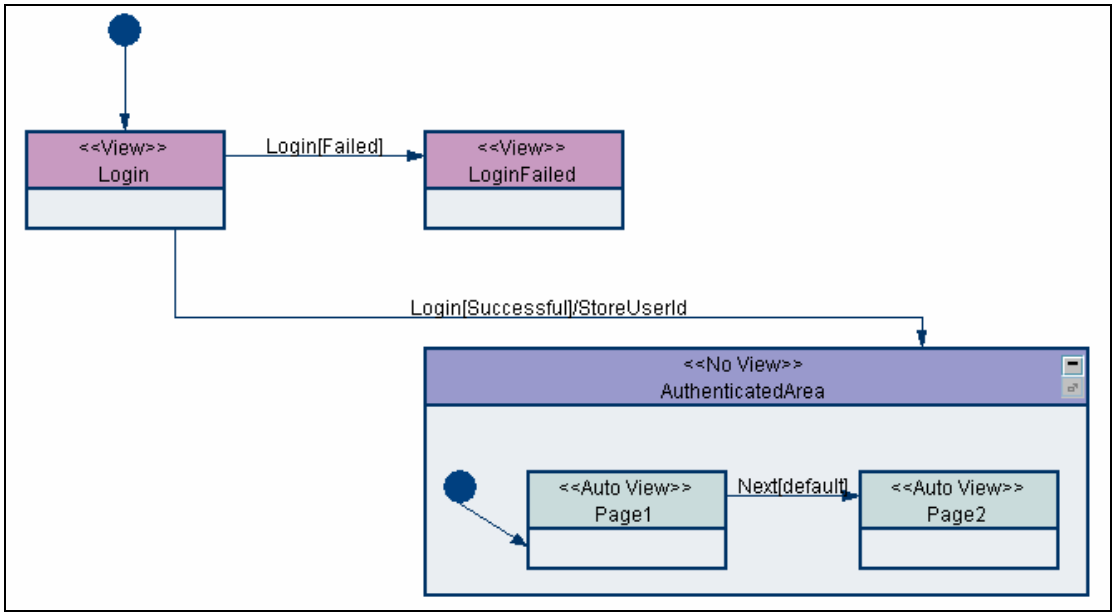
A solid circle as shown in Figure 10 represents an initial pseudo-state.

Figure 10. An Initial Pseudo-State

An initial pseudo-state indicates the starting point or state for a statechart. Initial pseudo-states are also used in parent states to indicate where the starting point is within that parent state. Figure 11 shows the login example extended to include child states in the AuthenticatedArea parent state.

The statechart has two initial pseudo-states: one to indicate where the application starts, and the second to indicate which state is displayed first when the AuthenticatedArea state is entered (for example, Page1).

Figure 11. Using Initial Pseudo-States



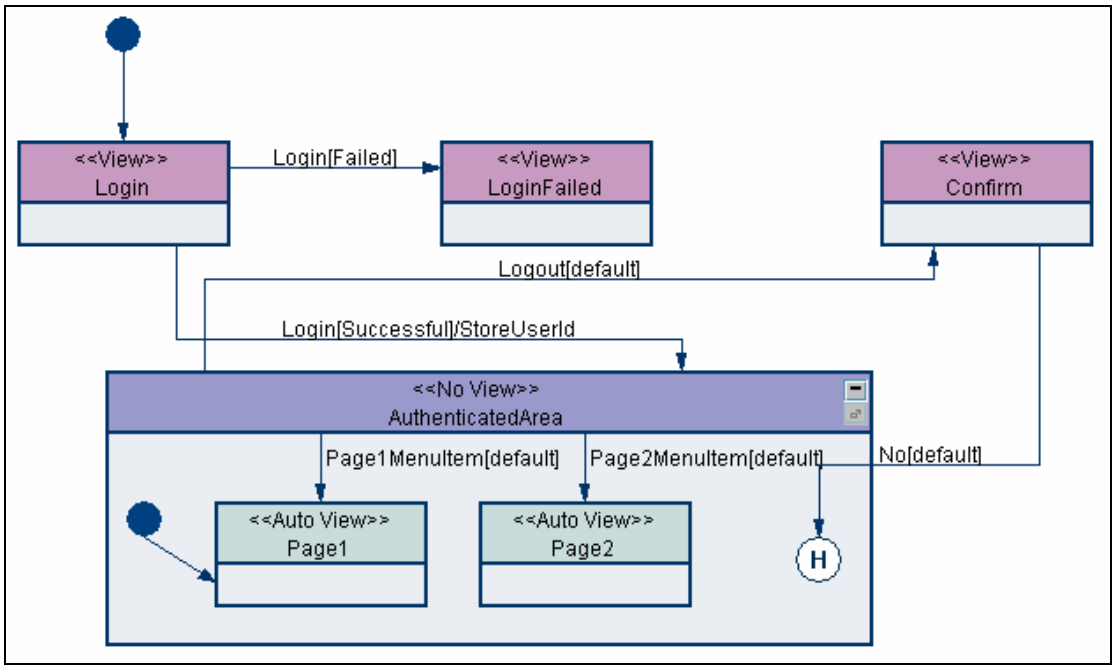
History State

A circle enclosing an H as shown in Figure 12 represents a history pseudo-state.

Figure 12. A History Pseudo-State

A history pseudo-state refers to children of a state that have recently been visited by the user. The history pseudo-state allows the user to return to the state that was the most recently visited immediate child of a parent state. Figure 13 shows a history pseudo-state in the AuthenticatedArea state.

Figure 13. The Login Example Extended to Use a History Pseudo-State



In this example, if the user cancels their logout event while on the Confirm state, the user is returned to the last visited state within the Authenticated area (that is, either Page1 or Page2, depending on which of the two screens were displayed before the Confirm state was displayed).

History-Star State

A circle enclosing a H* as shown in Figure 14 represents a history-star pseudo-state.

Figure 14. A History-star Pseudo-State

A history-star pseudo-state is very similar to the history pseudo-state. However, the history-star pseudo-state recursively returns the user to the immediate child of the parent state ending with the deepest visited state.

For example, if the statechart shown in Figure 13 is modified to use the history-star rather than the history pseudo-state, the state machine allows the user to return to the last displayed state and its deepest visited state. So if the user visits Page1 followed by a child state of Page1, the user is returned to the child state of Page1.

Final State

A solid circle enclosed by an outer circle as shown in Figure 15 represents a final pseudo-state.

Figure 15. A Final Pseudo-State

The final pseudo-state is used to indicate the final activity allowed on a statechart. It triggers a transition for leaving the application fired from the connected state to the final pseudo-state.

Exception State

A circle with an X across it as shown in Figure 16 represents an exception pseudo-state.

Figure 16. An Exception Pseudo-State

An exception pseudo-state is not part of the standard UML statechart notation; it was added to the state machine for exception handling. The state machine recognizes that exceptions can occur during event processing, and allows you to specify states within your user interface as exception states. When an exception is thrown, the user interface is placed properly in the appropriate exception state, with the user session still intact.

Effects of Pseudo-States on State Transitions

Pseudo-states indicate very specific behavior when used on a statechart. Similarly, pseudo-states have effects on state transitions that you should be aware of. For example, some pseudo-states can start a state transition but cannot end a state transition. Other pseudo-states cannot start a state transition but can end a state transition. Table 2 indicates the type of behaviors that are allowed by pseudo-states when drawing a state transition with them on a statechart.

Table 3. State Transition Behavior for Pseudo-States

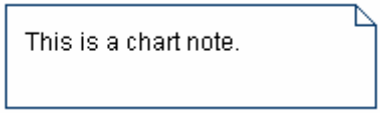
Pseudo-state	Start State in a Transition	End State in a Transition
Initial	Yes	No
History	No	Yes
History-star	No	Yes
Final	No	Yes
Exception	No	Yes

Chart Notes

You can add chart notes anywhere within a statechart to visually comment the statechart. The state machine makes no use of these notes. They are only used to explain the statechart to other users.

Figure 17 shows a chart note.

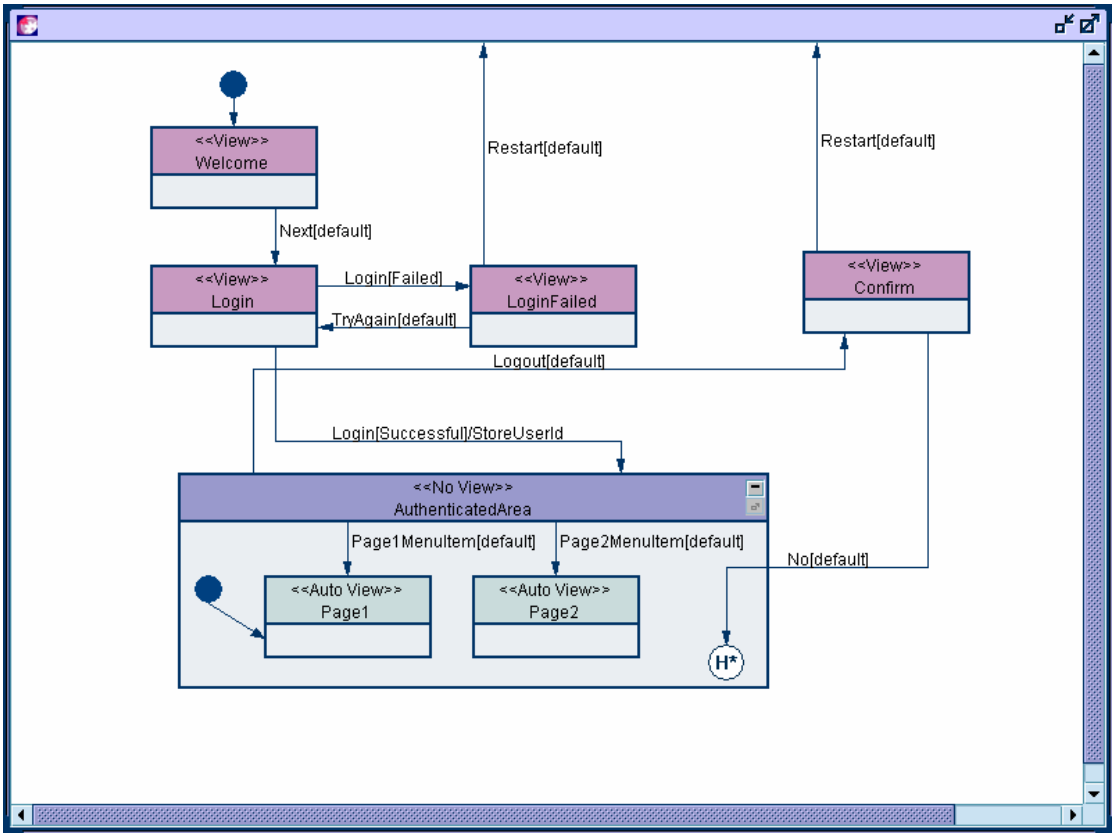
Figure 17. A Chart Note



A Simple Statechart Example

In the previous topics, a login example was used to demonstrate the statechart notation. In this topic that example is now extended to produce the statechart as shown in Figure 18. This topic explains what the statechart means and how to interpret it.

Figure 18. A Simple Statechart Example



The figure illustrates a simple statechart that represents a user interface that allows a user to log in to an authenticated area, move around that area, and then to restart the application by logging out of the application. Initially, when the user starts the application, the Welcome view is displayed as indicated by the statechart's initial pseudo-state. On the Welcome screen or view is a Next button. When the user clicks the Next button, the state machine displays the login view to the user. On the Login view, the user can enter their username and password. When the user clicks the Login button a Login event is fired. The state machine handles the result of the login and determines which guard condition is met.

- If the guard condition Failed is met, the state machine follows the Login[Failed] transition and displays the LoginFailed screen to the user. On the LoginFailed view there is a Restart button and a Try again button. If the user clicks the Try again button, the state machine displays the Login screen again. However, if the user clicks the Restart button, the state machine displays the Welcome screen.
- If guard condition Successful is met, the state machine follows the Login[Successful]/StoreUserId transition, and the state machine performs the StoreUserId action. The StoreUserId action requires the state machine to take the user ID returned from the login process and to store that user ID in the user's HTTP session. When the actions is completed, the state machine enters the AuthenticatedArea of the application and displays the Page1 view to the user as indicated by the initial pseudo-state in the AuthenticatedArea.

The user can move between Page1 and Page2 in the AuthenticatedArea by using menu or form buttons on the AuthenticatedArea screens. The AuthenticatedArea indicates that there is a Logout event. This event is inherited by the AuthenticatedArea's child states. This implies that both the Page1 and Page2 screens have a Logout button so that the user can fire the Logout event. The statechart does not require a Logout state transition arrow to be drawn from Page1 and Page2 to the Confirm state because Page1 and Page2 automatically inherit the event by being child states of the AuthenticatedArea state.

Customizing the Screen Orchestrator

The Screen Orchestrator is very flexible in that you can develop new views, controller classes, and guard condition classes and register them for use within the Screen Orchestrator. This customization is described in the following chapters:

- Chapter 6, Writing Controller Classes
- Chapter 7, Writing Guard Condition Classes
- Chapter 12, Writing a Swing Application, which contains a section about adding new view classes.

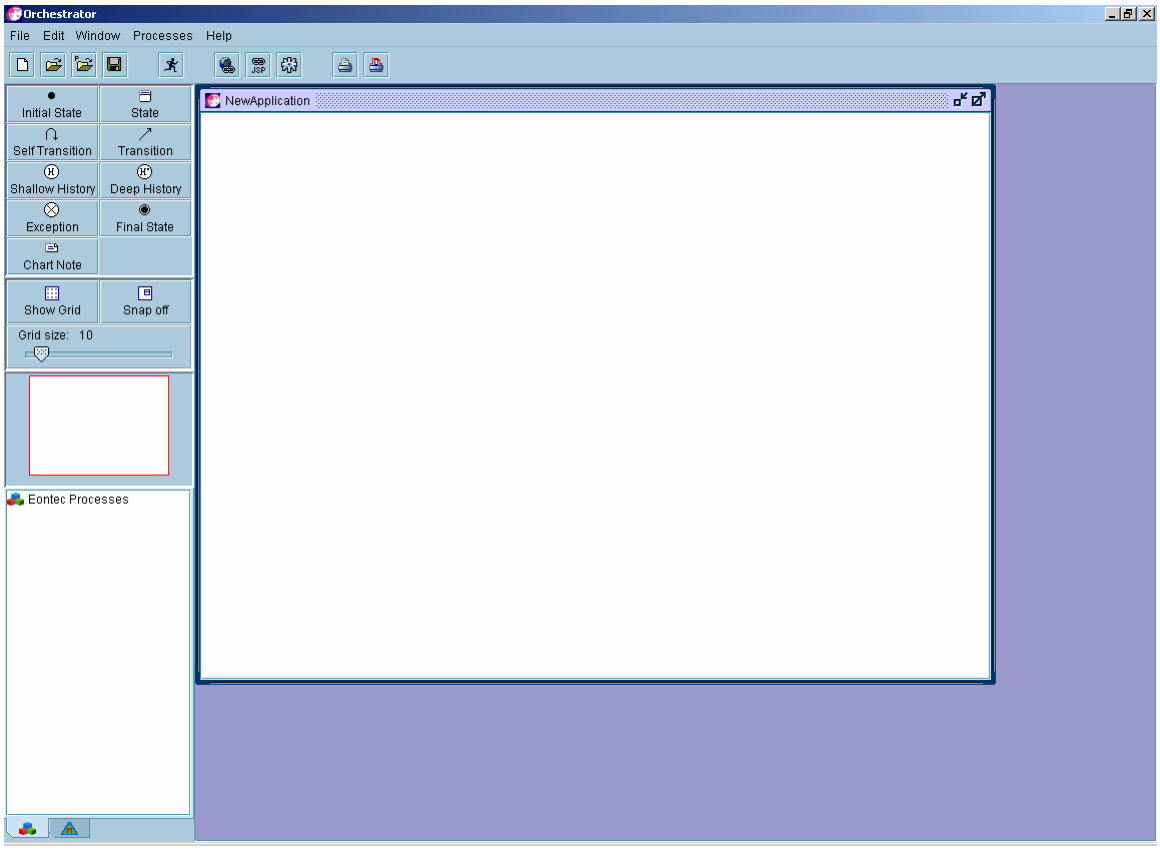
3 Basic Screen Orchestrator Drawing

This chapter introduces the basic drawing capabilities of the Screen Orchestrator. After you work though this chapter, you should be able to build a standard statechart with the Screen Orchestrator. This chapter includes the following topics:

The Main Screen Orchestrator Window

When the Screen Orchestrator starts, the screen in Figure 19 is displayed.

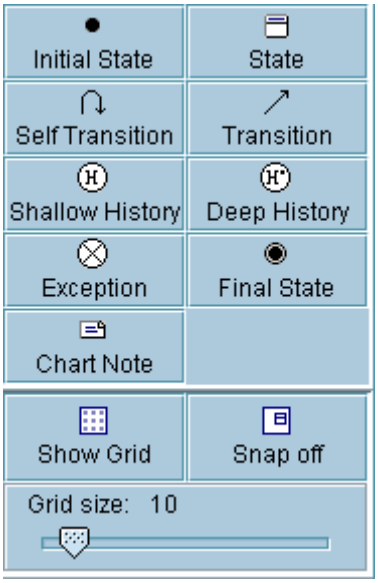
Figure 19. The Screen Orchestrator Window



As shown in Figure 19, The Screen Orchestrator provides a menu and toolbar to access its major functions. The main drawing components of the Screen Orchestrator, with which you draw statecharts, are displayed in a palette in the upper-left of the Screen Orchestrator (Figure 20). This component palette contains components for the states, transitions, and notes that you add to a statechart.

Below the component palette on the main window is a navigation window (Figure 21) that displays a miniature view of the statechart that you are currently drawing. You can use this window to navigate a

Figure 20. Component Palette Used in Creating Statecharts



statechart to a particular area within the currently visible statechart window.

Below the navigation window is a further panel, the Siebel Processes panel (see Figure 22). This panel displays the list of Siebel processes available to you when populating a statechart with process information.

Figure 21. The Navigation Window

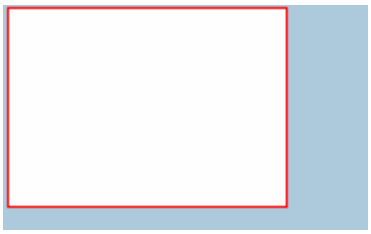
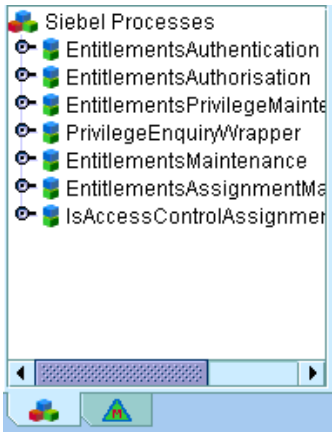


Figure 22. The Siebel Processes Panel



At the bottom of the Siebel Processes panel are two tabs that allow you to toggle between the Siebel Processes panel and a further panel, the Application Metrics panel (see Figure 23). This panel displays the number of states, view states, autoviews, events, complex controllers, transitions, and complex guard conditions in a statechart. You can use this information to determine the complexity of the statechart, to help with task estimation, and to track progress of the application, that is, the number of autoviews left to be coded to actual views.

Figure 23. The Application Metrics Panel

As mentioned previously, this topic concentrates on the basic drawing of statecharts using the Screen Orchestrator, later topics explain fully the use of the process panel.

The Screen Orchestrator provides a multiple document interface (MDI) desktop in which you can draw statecharts. The Screen Orchestrator allows you to create or edit a single statechart at a time. However, you can open larger parent states into smaller subchart windows within the desktop. Opening parent states as subcharts is explained in detail in Opening Subcharts on page 93.

The Screen Orchestrator Toolbar

The Screen Orchestrator contains a toolbar that provides icons for tools as described in Table 4.

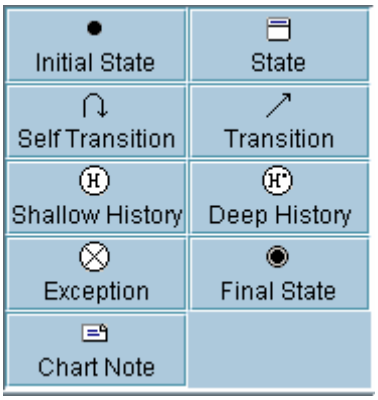
Table 4. The Screen Orchestrator Toolbar

Icon	Equivalent Menu Option	Description
	File > New	Starts a new statechart.
	File > Open	Opens an existing statechart. For more information, see Opening a Statechart on page 42.
	File > Open Part	Opens a part statechart. For more information, see Multiple User Support on page 97.
	File > Save XML	Saves the statechart. For more information, see Saving a Statechart on page 41.
	No equivalent	Opens the preview window for a statechart. For more information, see The Preview Capability on page 47.
	File > Save XML and Re-generate War	Deploys a Web Archive (WAR) file. For more information, see Deploying the WAR File on page 51.
	No equivalent	Tests whether Java Server Pages (JSP) can be compiled. For more information, see Testing Whether JSPs Can Be Compiled on page 127.
	No equivalent	Generates JSPs of panel files, or both, for a view. For more information, see Generating JSP and Swing Panel Files on page 127.
	File > Print All	Prints all of the currently selected statechart. For more information, see Printing Statecharts on page 44.
	File > Print to Scale	Prints the currently selected statechart to scale. For more information, see Printing Statecharts on page 44.

The Statechart Drawing Components

The statechart drawing components as shown in Figure 24 support the statechart notation as described in Statechart Notation Explained on page 11.

Figure 24. Statechart Notation Components Palette Used for Drawing Statecharts



Drawing States or Pseudo-States

You draw states or pseudo-states by dragging them from the component palette to the statechart window. When you create a state, you must enter state details. Pseudo-states do not require titles or require to be configured in any way. You can move a state or pseudo-state at any time by clicking on the state and dragging the state to any new location within the statechart window. Figure 26 illustrates the statechart window after a state titled Welcome is created.

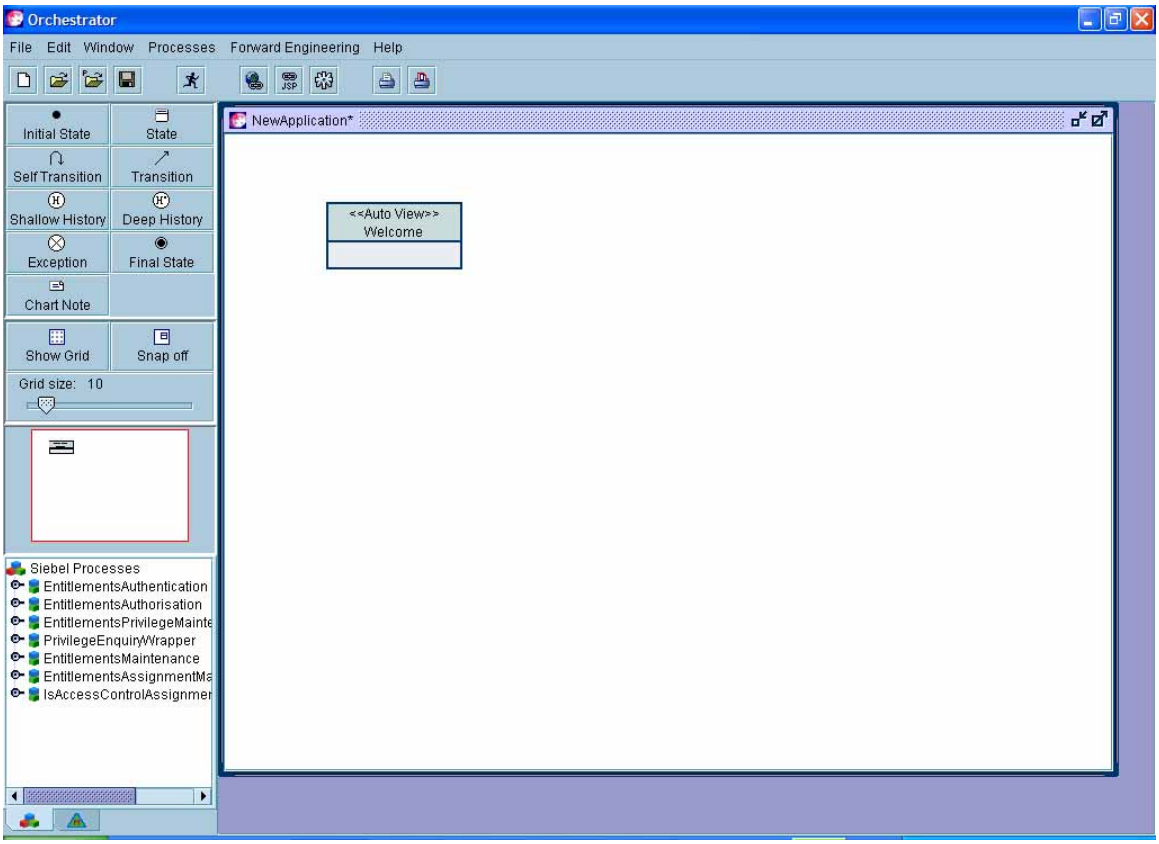
To create a state or a pseudo-state on the statechart window

1. Click on the required state or pseudo-state in the component palette and drag the state onto the statechart window.
2. Release the mouse button over an area in the statechart window where you want to drop the state or pseudo-state.

   A state or pseudo-state is created by the Screen Orchestrator on the dropped location in the statechart. For a state, the Enter State Details screen is displayed.
3. Enter the state's details; as a minimum type a title, and click OK.

   The state is displayed with the appropriate title and state type.

Figure 26. Welcome State Drawn



Drawing State Transitions

To connect two states together to represent a state transition, you click on the transition component (Figure 25) and then drag and drop the component onto the statechart directly over the header of the state or anywhere over a pseudo-state that is the starting state of a state transition.

Figure 25. The State Transition Component

To draw a state transition

1. Click on the transition component in the component palette.

   The Screen Orchestrator immediately changes the cursor to a cross symbol.

2. Drag and drop the component to the header of the start state of the transition.

3. Move the mouse to the end state or pseudo-state of the transition.

   As the mouse is moved about the statechart towards the end state of the transition, a temporary state transition line is drawn towards the end state.

4 When the mouse is over the header of the state or anywhere over a pseudo-state, click on the state.

The cursor is returned to the default cursor, and the state transition is drawn. The Transition Wizard is then displayed.

5 Enter the event name of the transition or select an existing event on the starting state of the transition if one already exists.

For example, in the login example used in the statechart concepts topic, if you were drawing a transition from the Welcome state to the Login state, you would type Next in the Event field.

6 Click Next.

The Enter Transition Details screen is displayed.

7 Enter an action description and guard condition for the state transition.

NOTE: In the state transition from the Welcome state to Login state no action or guard condition exists.

8 Click Finish.

The state transition is created. For the Login example, the state transition Next[default] shown in Figure 27 is created.

To draw the Login[Successful]/StoreUserId state transition, you enter the information shown in Figure 28 and Figure 29 in the Transition Wizard. This action creates the state transition shown in Figure 30.

Figure 27. The Next[default] State Transition

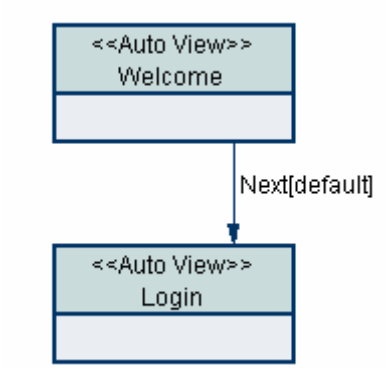
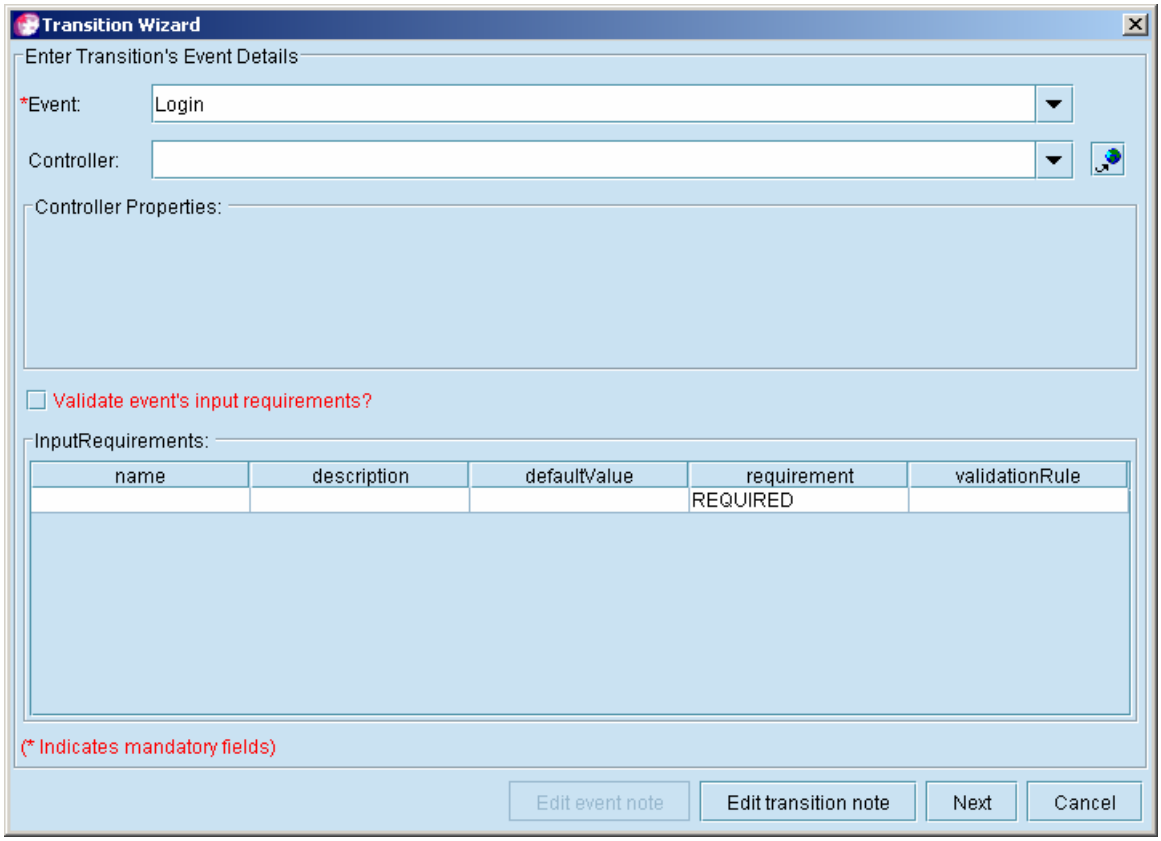


Figure 28. Event Details Screen with Login Event



This section only describes the basics of drawing state transitions; detailed information about coding events in the Screen Orchestrator and using the Transition Wizard is given in Chapter 5, Coding Events with Processes and Guard Conditions.

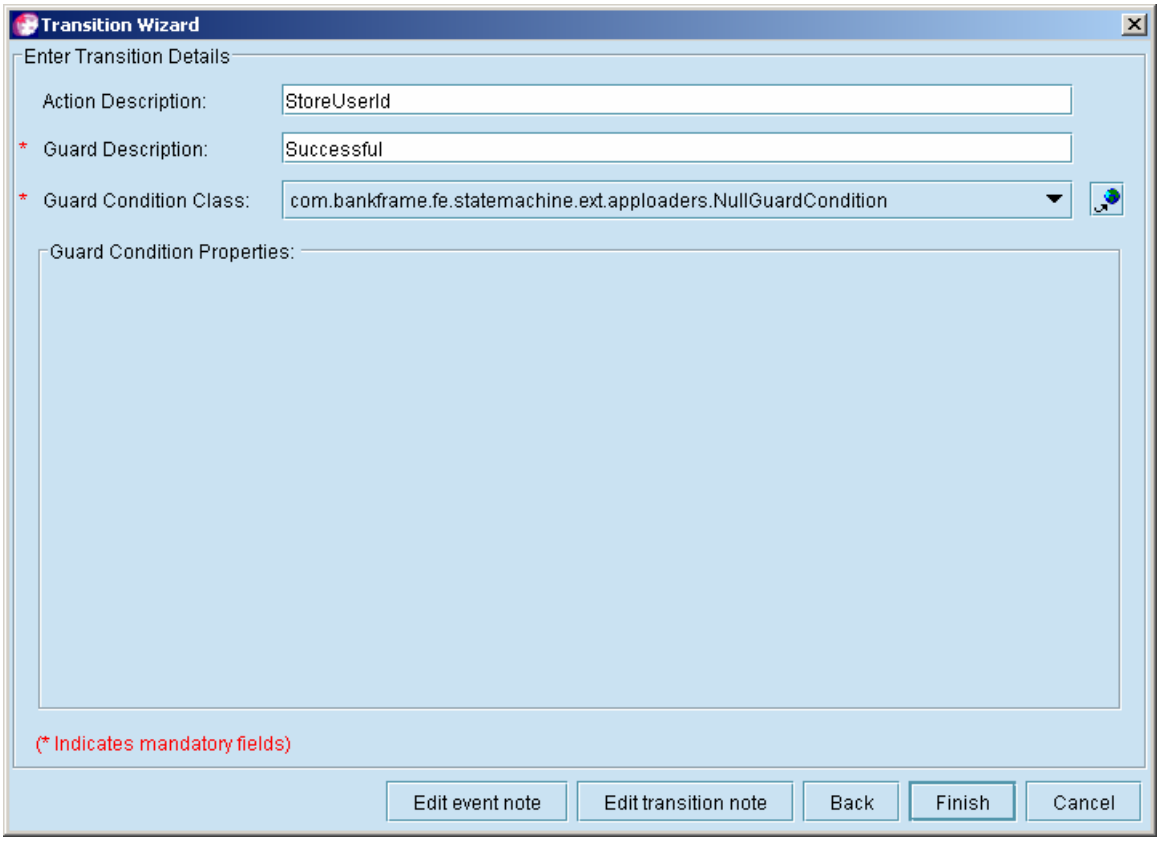


Figure 29. Transition Details Screen with StoreUserId Action and Successful Guard Condition

Figure 30. The Login[Successful]/StoreUserId State Transition

Drawing Self-Transitions

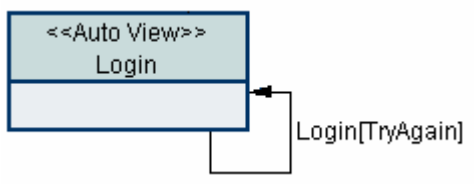
To draw a self-transition, you click on the self-transition component in the component palette (Figure 31) and drag the component onto a state header. You cannot add a self-transition to pseudo-state; a warning is displayed if you attempt to do so.

Figure 31. The Self-Transition Component

When you release the mouse button over a state, the Screen Orchestrator draws a self-transition around the state and displays the Transition Wizard.

A self-transition has exactly the same properties as a state transition, except that its start state and end state are the same. Figure 32 shows a self-transition drawn around a Login state, when the Login event's guard condition is TryAgain. If the login fails, the state machine can redisplay the Login state allowing the user to try logging in again.

Figure 32. The Login[TryAgain] Self-Transition

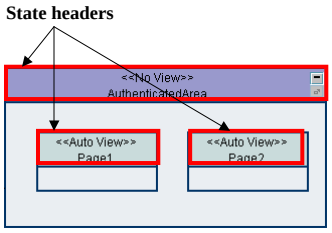


State Header

When you drop a transition or self-transition on a state you must drop the drawing component on the state's header. Similarly, when you specify the end state of a state transition, you must click on the state's header for the Screen Orchestrator to identify the state that is used for the end state of that transition. The state header is the top rectangular box of the state where the state's type and name (or title) is displayed. Figure 33 highlights the header area of a number of states.

The bottom rectangular box of a state is where you can add child states to the state. Adding child states to a parent state is discussed in Adding Child States on page 34.

Figure 33. The State Header Area



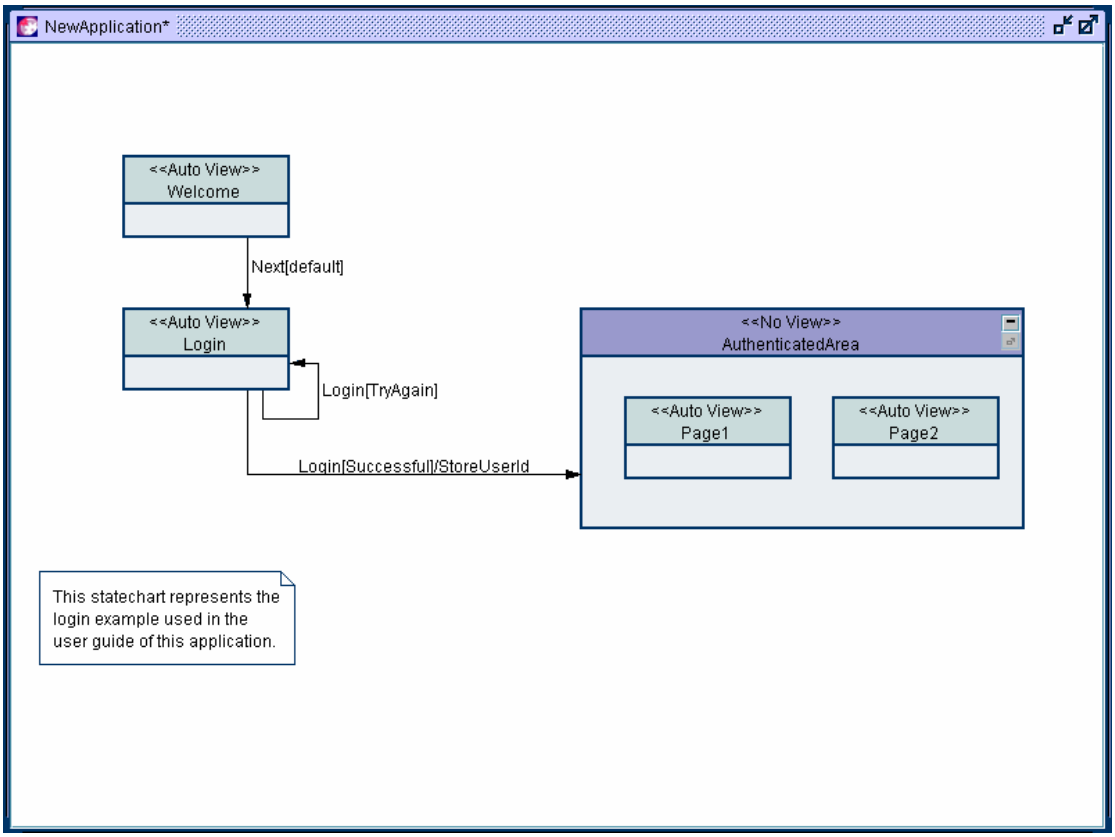
Drawing Chart Notes

You can add chart notes anywhere in a statechart. To draw a chart note, you click on the chart note component in the component palette (which is shown in Figure 34) and drag the component onto the statechart.

Figure 34. The Chart Note Component

A statechart can have as many notes as you want, and you can add them to states just as you can add any child state to any parent state.

Figure 35. A Statechart Showing a Chart Note Attached



You can also add notes to events and transitions by clicking respectively the Edit Event Details and Edit Transition Details button in the Transition Wizard (see Figure 28). These notes are added to the XML file that is produced when you save the statechart.

To draw a chart note

1. Click the chart note component in the Component Palette.
2. Drag the chart note to the required location on the statechart.
3. Click in the chart note and type text directly into the note.

   The note grows automatically in size as you type more text.

More Drawing State Details

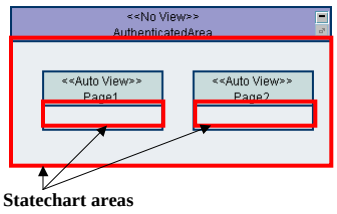
You can create states and pseudo-states and move them anywhere within other states. You can resize them and edit their details, for example, to change the name, or the state type. The following topics describe how to perform these actions.

Adding Child States

To add a state or pseudo-state as a child to a parent state, you simply drag and drop a state component from the component palette onto the parent state's statechart area.

Every state has a statechart area on which child states can be added as shown in Figure 36.

Figure 36. Statechart Areas Within a State



When you drop a state onto a parent state's statechart area, the parent state is resized to fit the child state and any other child states of the parent. If you then move the child state further around the parent state, the parent state continues to be resized if required.

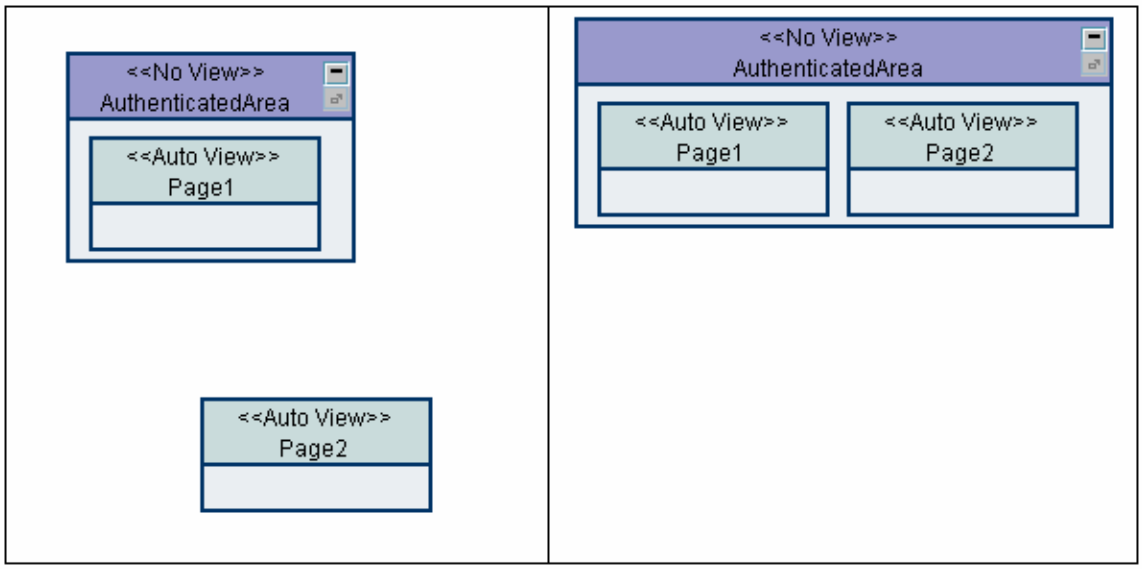
Moving States

After you create a state or pseudo-state, you can move it to any location in the statechart.

To move any state, simply click anywhere on the state's header or, if it is a pseudo-state, anywhere on the body of the pseudo-state, and drag and drop the state to its new location.

If you move a state inside a parent state, the parent state is resized automatically if required. If the state is outside a parent state but should be inside, dragging the state inside the parent state is allowed. The state immediately becomes a child state of the parent state. Similarly, the reverse is also true. If a state is initially located as a child state of some parent state, moving the child state anywhere outside the parent state immediately releases the state as a child state of the parent state.

Figure 37. Moving the Page2 State into the AuthenticatedArea State



Editing State Details

You can change the details of a state, such as the title, state type, state details, or input parameters at any time.

To edit state details

1. Right-click on the state header to display the state's pop-up menu.
2. Navigate to the Edit > Details > Enter State Details screen.
3. Edit the state details as required.

   For example, to change the AuthenticatedArea's state type from an AutoView state type to a No View or (NONE) state type, select NONE in the State Type list.
4. Click OK to save your changes.

Resizing States

The Screen Orchestrator automatically resizes states when adding child states or moving states within a parent state. However, you can also specify a state's actual size.

To resize a state

1. Right-click on the state header to display the state's pop-up menu.
2. Navigate to the Edit > Size > Enter State Size screen.

   The screen indicates the current width and height of the state and indicates the best-fit size for the state.

3 Type the required width and height in the Enter size: and click OK.

Deleting States

You can delete any state or pseudo-state. When you delete a parent state, all its child states, transitions, and child transitions are also deleted.

To delete a state

1 Right-click on the state header to display the state's pop-up menu.

2 Click Delete.

3 Click Yes to confirm the deletion.

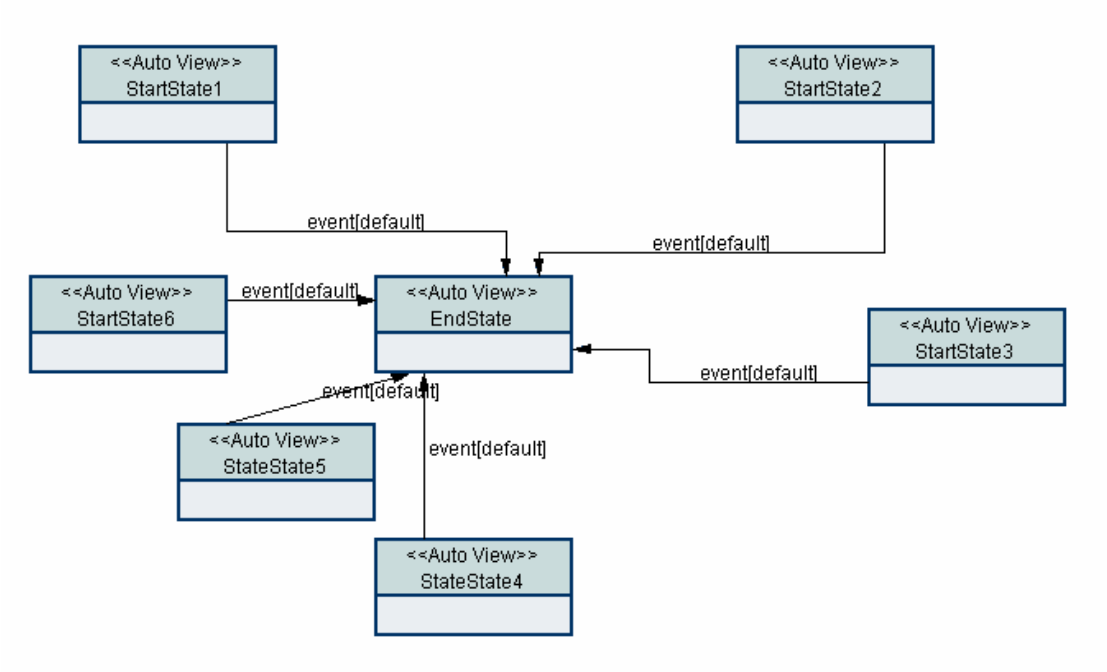
More Drawing Transition Details

You can draw transitions between any two states; however, you can only draw certain types of transitions between states and pseudo-states. For information about which transitions are supported by pseudo-states, see Table 3 on page 20.

About Transition Arrows

When you draw a transition between any two states (regardless of whether a state is a pseudo-state), the Screen Orchestrator determines how the transition arrow is drawn between them. You have no real control over how the transition arrow is drawn. However, moving the states that participate in the transition can alter how the transition arrow is drawn. For example, Figure 38 illustrates the types of transition arrows that are drawn given the relative locations of the states in the transitions.

Figure 38. Types of Transition Arrows Drawn by Screen Orchestrator



If you need to change a transition arrow and how it is drawn, you must adjust the location of one of the transition's states. Moving one of the states involved in the transition further away or closer or at a different angle to the other state forces the transition arrow to be drawn differently.

Drawing Transitions to the Master State

The desktop window that displays the statechart is itself a state and as such you can draw transitions to it. Within the Screen Orchestrator this state is known as the master or application state. State transitions are often drawn to the master state to indicate that the application must be restarted. In the login example described in Chapter 2, a state transition arrow is drawn from the LoginFailed state and the Confirm state to the master state to indicate that the application must be restarted as a result of these transitions.

To draw a transaction to the master state

1. Click on the transition component in the component palette.
2. Drag the component to the start state of the transition.
3. Move the mouse to anywhere in the statechart area apart from where a parent or child state already exists, and click on the statechart.
4. Enter the required details for the transition, and click OK.

Figure 39 indicates the position of the end transition mouse-click for drawing the Restart transition arrow on the Confirm state.

Figure 40 shows the completed Restart transition for the Confirm state.

Figure 39. Creating a Transition from a State to the Master State

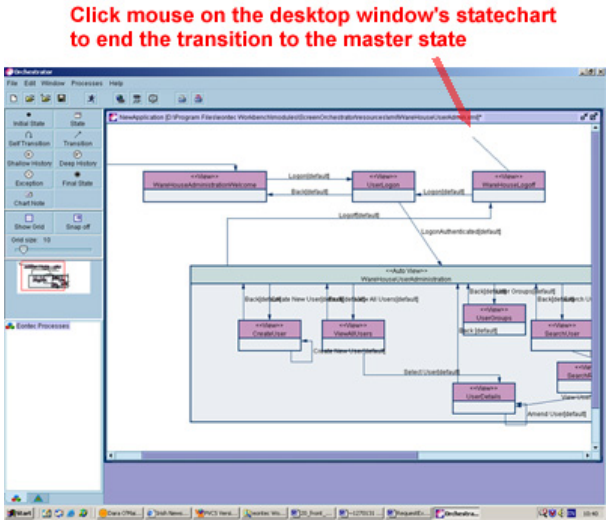
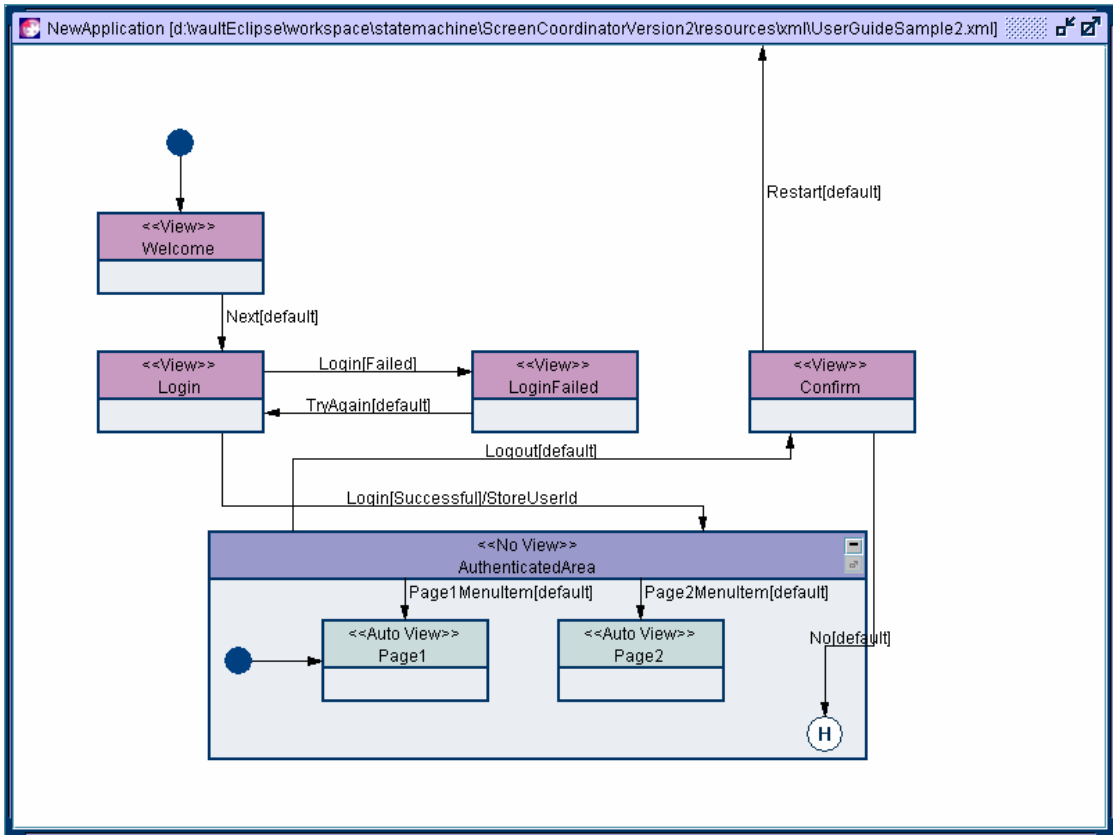


Figure 40. The Restart Transition from the Confirm State to the Master State



Drawing Transitions to and from Parent and Child States

You can draw transitions between parent and child states just as you draw a normal state transition. Dragging the transition component onto the parent state's header starts the transition and clicking the mouse on the child state's header ends the drawing of the transition as normal. Figure 40 shows two such transitions. The AuthenticatedArea has parent to child transition arrows to the Page1 and Page2 states.

Drawing Transitions to and from Unrelated Child States

You can draw transitions to and from states that are unrelated child states. This capability allows you to draw a transition arrow between any two states anywhere on the statechart, regardless of whether they are children of the same parent state, not children, or just child states of the statechart itself. The only limitation on drawing a transition arrow is when one of the states is a pseudo-state. For information about which transitions are supported by pseudo-states, see Table 3 on page 20.

Editing Transition Details

You can edit the details for any transition either directly or through the start state for the transition.

To edit transition details directly

1. Right-click on the transition to display the transition's pop-up menu.
2. Click the Edit *label* option.
3. Edit the transition's details as required.
4. Click Finish to save your changes.

To edit transition details from the start state

1. Right-click on the state header of the start state of the transition to display the state's pop-up menu.
2. Navigate to the Transitions > Edit option and click the transition's label option, for example, Login[Failed].
3. Edit the transition's details as required.
4. Click Finish to save your changes.

Deleting Transitions

You can delete a transition either directly or through the start state for the transition.

To delete a transition directly

1. Right-click on the transition to display the transition's pop-up menu.
2. Click the Delete *label* option.
3. Click Yes to confirm the deletion.

To delete transition details from the start state

1. Right-click on the state header of the start state of the transition to display the state's pop-up menu.
2. Navigate to the Transitions > Delete option and click the transition's label option, for example, Login[Failed].
3. Click Yes to confirm the deletion.

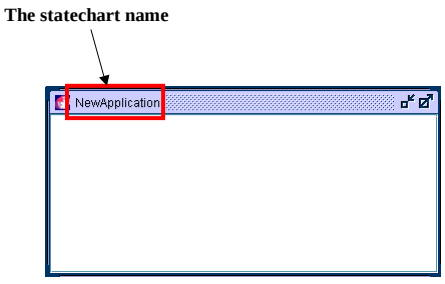
More Drawing Statechart Details

Apart from drawing states and transitions, you can edit details of the statechart itself, as described in the following topics.

The Statechart Name

Each statechart represents an application. The name of the statechart is also the application name and the name of the master state of the statechart. The name of the statechart is displayed in the title bar for the desktop window of the statechart, as shown in Figure 41.

Figure 41. Statechart with Name Highlighted



Renaming the Statechart

You can rename statecharts at any time.

To rename a statechart

1. Right-click on the statechart in an area where no state is located.
2. Click Edit Application Title from the pop-up menu.
3. Type the new title for the application and statechart.
4. Click Save.

   The title bar of the statechart window changes to reflect the new application title.

Saving a Statechart

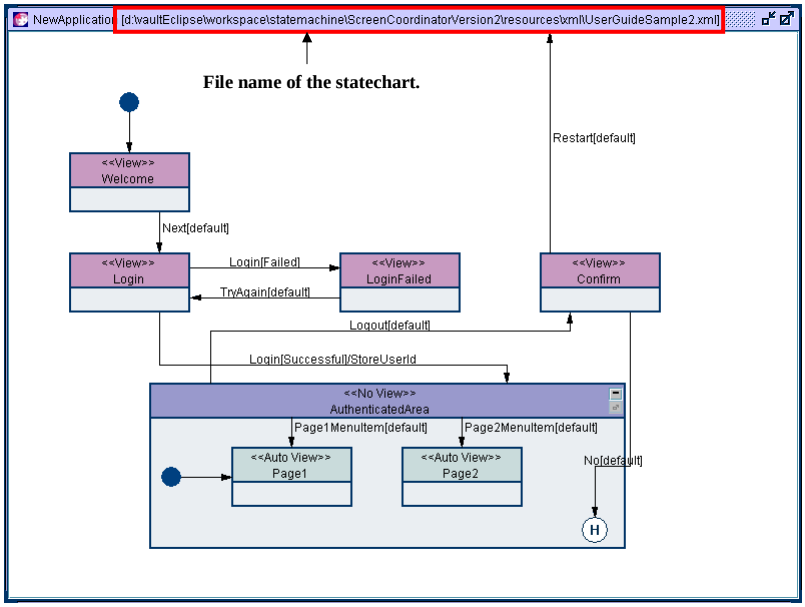
By default the statechart is saved with a filename that is the same as the statechart's application title. You can give a statechart any name, but it is recommended that you save it with the same name as the application.

When you save the statechart, the title bar of the statechart window is updated to include the filename of the statechart, see Figure 42.

To save a statechart

- Select File > Save XML to save a statechart to a file with the same filename as the application. If the statechart was not saved before, you are prompted for a filename for the statechart.
- Select File > Save XML As and enter a new filename, to save a statechart to a file with a different name from the application.

Figure 42. Statechart Window with Filename of the Statechart Highlighted



As you add items to a statechart such as states, pseudo-states, and transitions, move any state in the statechart, or edit any states or transition details, the Screen Orchestrator indicates that a file save is required by adding an asterisk to the statechart's window title bar.

Renaming a Saved Statechart

If you started a statechart and built it up over some period of time, you have already named the statechart and saved it to a file. Over time you might then want to rename the statechart to a more appropriate name.

When you rename a statechart, the Screen Orchestrator also automatically renames the file associated with the statechart to match.

Opening a Statechart

You open a statechart by:

- Clicking the Open File button on the toolbar
- Selecting File > Open from the menu bar
- By selecting the file from the recently opened file list on the File menu, if the statechart was opened recently

If you already have a statechart opened, you are asked to save it (if required) before opening the new statechart.

Resizing the Statechart Window

You can resize the window in which the statechart is displayed.

To resize a statechart

1. Right-click on the statechart to display the statechart's pop-up menu.
2. Navigate to the Edit Application Size > Enter State Size screen.

   The screen indicates the current width and height of the statechart and indicates the minimum size for the statechart.
3. Type the required width and height in the Enter size: text boxes, and click OK.

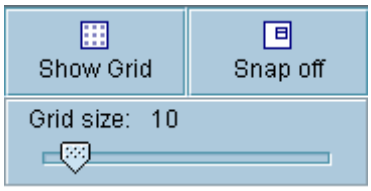
Miscellaneous Drawing Features

The following subtopics describe the remaining important features required for basic drawing with the Screen Orchestrator. More advanced drawing features are covered in the Advanced Drawing chapter.

Using the Grid and Snap To Features

The Screen Orchestrator's component palette provides grid and snap to features as shown in Figure 43.

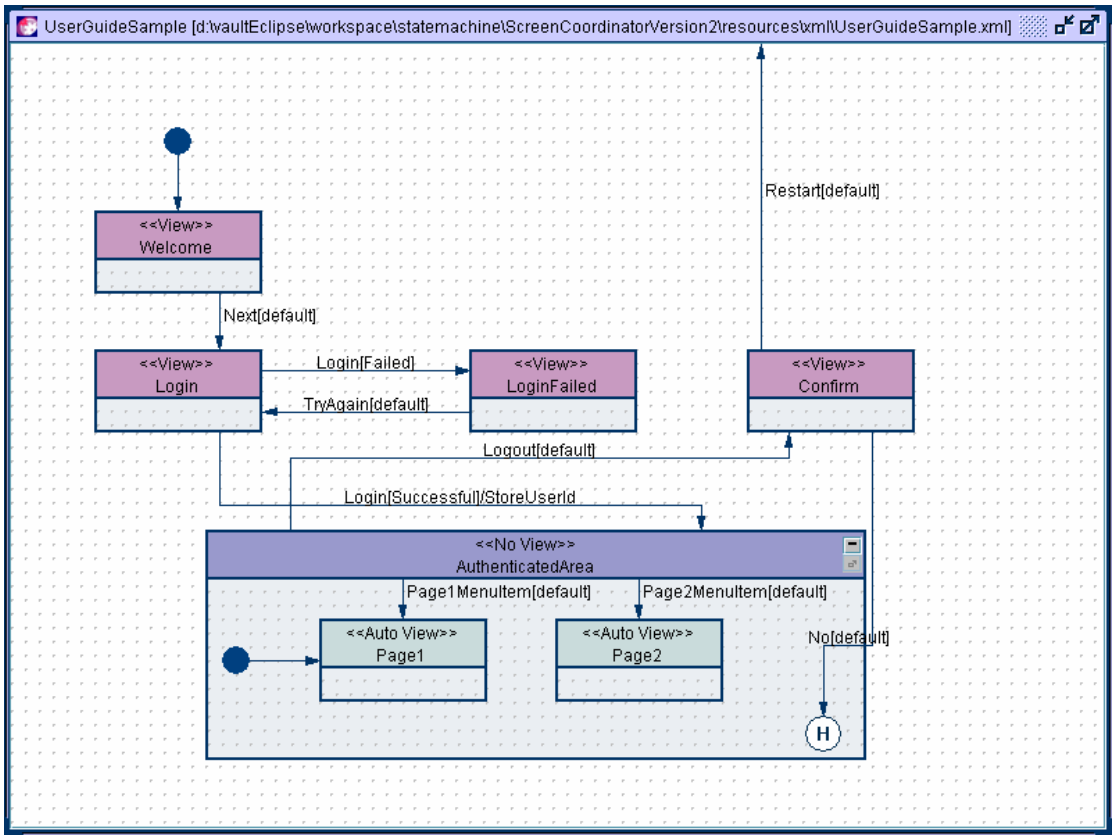
Figure 43. The Show Grid and Snap Off Buttons



By default the grid feature is off, while the snap to feature is on, when you start the Screen Orchestrator.

The grid feature allows you to use a visible grid to position and align states drawn on the statechart. To switch the grid on, click the Show Grid button. The statechart window then displays a grid as shown in Figure 44.

Figure 44. Statechart Window with Grid Feature On



The Show Grid button is toggled with a Hide Grid button, so that you can hide the grid when required.

NOTE: One side effect of switching on the grid is that specifying transitions is slightly slower.

You can adjust the grid size by moving the grid size slider left or right. Moving the slider left decreases the grid size down to a minimum of 5 pixels per grid square. Moving the slider to the right allows the grid size to be increased to a maximum of 50 pixels per grid square. The default grid size is 10 pixels per grid square.

The snap to tool allows you to locate states on actual grid points regardless of whether the grid is visible. If you create or move a state while the snap to tool is on, the state is automatically located to the nearest grid point.

Clicking the Snap Off button switches the snap to feature off. When you create or move states they are located exactly where they are dropped. Clicking the Snap Off button toggles the button back to a Snap On button.

Using the Navigation Panel

As a statechart gets larger and moves outside the size of the statechart window, that window becomes scrollable. To help navigate around the window, the Screen Orchestrator includes a navigation panel as shown in Figure 45.

Figure 45. The Navigation Panel for a Scrollable Statechart

The navigation panel displays a miniature representation of the statechart. The red box in the panel indicates the currently visible area in the statechart's desktop window. If you click inside the red box, the cursor changes to a hand and you can drag the box to a new location within the navigation panel. As you drag, the statechart window is moved and located to correspond with the area visible in the navigation panel.

Printing Statecharts

You can print a statechart by clicking a Print button on the toolbar or by selecting an option from the File menu.

To print a statechart

- Select Print All, to print the currently selected statechart to scale, and to print all pages, if it is larger than a single page.
- Select Print to Scale, to display the Print Dialog screen, then print the statechart to a single page or multiple pages.

- Click Single page to print the statechart on one page. If the statechart is larger than a single page, the statechart is scaled to fit on a single page.
- Click Multiple pages to print the statechart on multiple pages.

Exporting a Statechart as a GIF File

You can export a statechart as a GIF file by selecting File > Export as a GIF file from the menu bar.

You can then import the GIF file into design documents or send it electronically, which allows users without a Screen Orchestrator installation to view and analyze the statechart.

4 The Preview Capability and Web Deployment

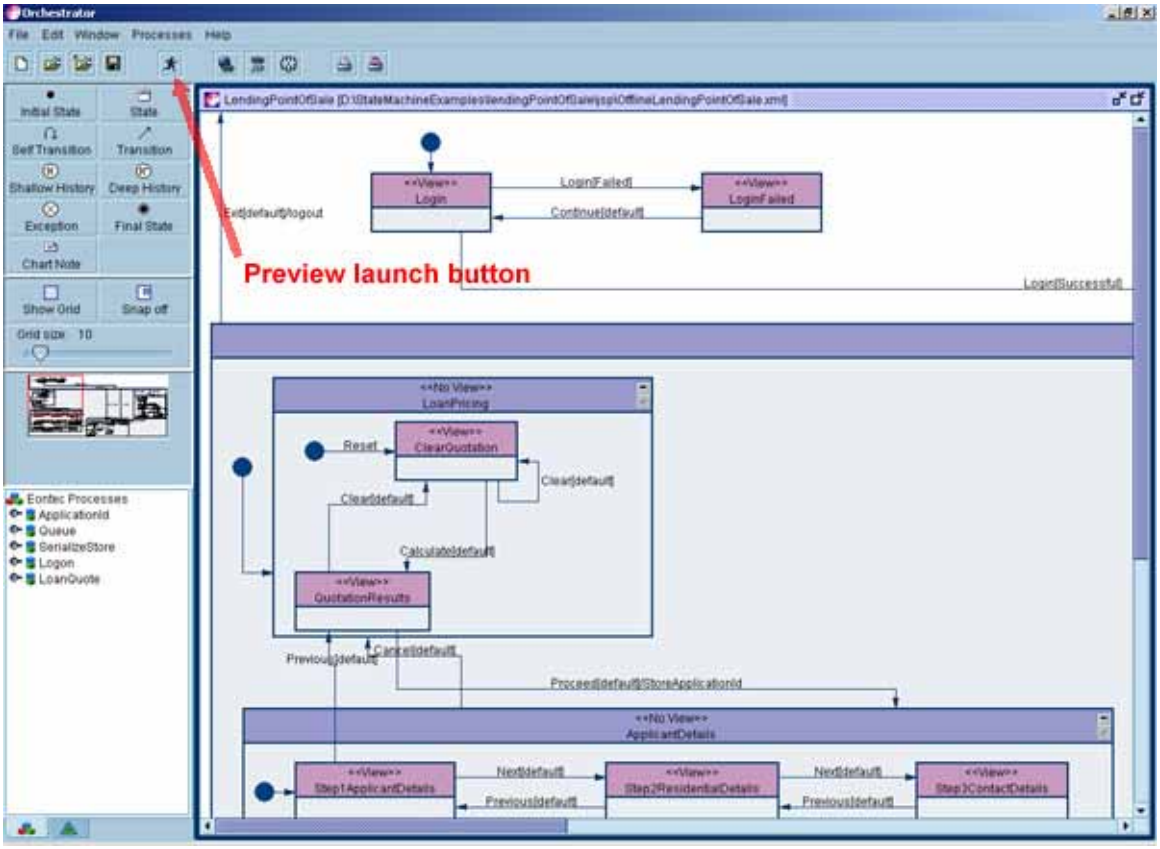
The application that you model in the Screen Orchestrator is typically Web-based, therefore you must package the application into a Web Archive (WAR) file and deploy it on a suitable Web server. The Screen Orchestrator provides functions for creating and deploying a WAR file for Web-based applications; it also provides a preview window for testing and verifying the statechart design before deployment.

This chapter includes the following topics:

The Preview Capability

As you develop a statechart you can preview it by clicking the Preview button, see Figure 46.

Figure 46. Highlighted Preview Button



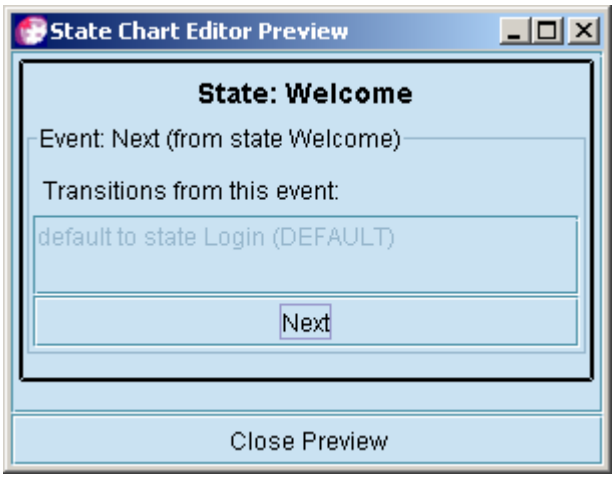
The preview window displays the initial state and allows you to fire events to the next state in the drawn statechart. The preview window does not use the specified view state type but instead uses a Swing-based autoview to display the state. If the state's events have the correct controllers and guard condition classes, you can use the preview to follow the actual transitions that occur in the real application.

The Screen Orchestrator also allows you to use a special controller with the preview window. If you use the AutoViewController for events when drawing the statechart, the preview window allows you to actually select the transition that you want to follow. This capability is extremely useful for verifying all the statechart's transitions including those that might represent rare behavior that is difficult to duplicate.

The figures in this topic use the Login example with AutoViewController, and illustrate the way in which you can follow the transitions with the preview feature.

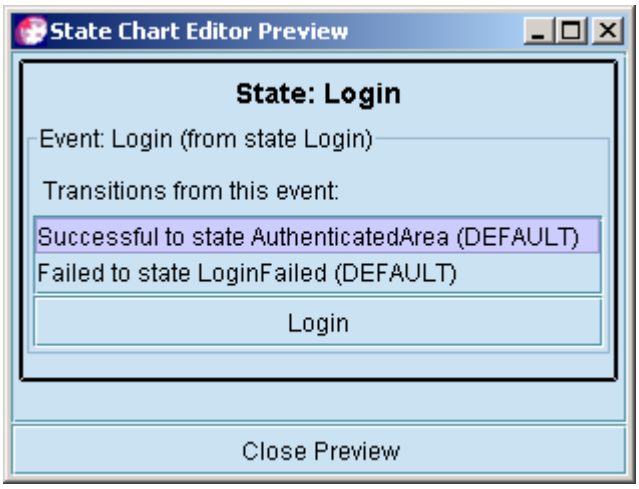
The initial state of the Login example application is the Welcome state. You fire the Next event by clicking the Next button in the preview, see Figure 47.

Figure 47. Statechart Preview: Welcome State



The Next event takes you to the Login state (Figure 48). This state's Login event has two possible transitions. The event is either successful, or it fails. The AutoView Controller is used for this event, which allows you to select the Successful transition to be followed when you click the Login button.

Figure 48. Statechart Preview: Login State



As shown in Figure 48, the Login was successful, and so the initial state of the AuthenticatedArea is the Page1 state (Figure 49.) You now click the Logout button to fire the Logout event, and proceed to the Confirm state (Figure 50).

Figure 49. Statechart Preview: Page1 State

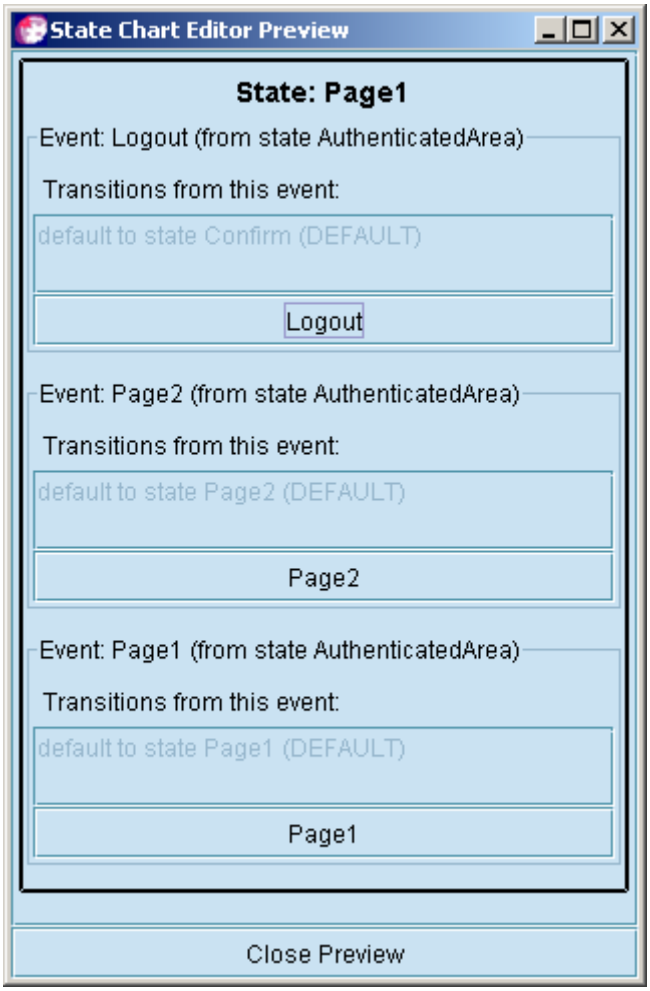


Figure 50. Statechart Preview: Confirm State

Web Deployment Capability

You can use the Screen Orchestrator to produce a WAR file for deployment of Web-based applications; the following subtopics describe how to access and use this feature.

Supported Web Servers

You can deploy the state machine framework on any suitable HTTP server that supports Java servlets and Java Sever Pages (JSP). The state machine framework and the WAR files produced by the Screen Orchestrator have been tested on the following application servers:

- JBoss 3.0.*x* and higher
- JBoss 3.0.*x* with Tomcat and higher
- BEA WebLogic 6.1 and higher
- IBM WebSphere 5.*x* and higher (you might need to update the jdom.jar installed with this application server for the state machine to work correctly)

Configuring WAR File Properties for Statecharts

When you create a new statechart, default WAR file properties are automatically assigned. However, you can configure the properties for each statechart.

To configure the WAR file properties

1 In the Screen Orchestrator navigate to File > War Properties.

The Confirm War Properties screen is displayed. There are two tabs, one showing the default set of properties and one that allows you to specify a set of properties.

2 If you want to use the default set of properties, click Save Default Properties.

3 If you want to change any properties, complete the details on the Chart Properties tab.

The fields are described in the following table.

Field	Comment
Destination Dir	Type the directory where the application puts the packaged WAR file, that is, the webapps folder of the Web server. For example, for JBoss, specify the \jboss-3.0.x\server\default\deploy directory. When the WAR file is generated, it is transferred to this location for deployment. This property is probably the property that you change most often.
Temp Dir	Type the directory where the WAR file is generated and packaged before it is copied to the Destination Dir value.
Libraries Dir	Type the directory where the JAR files that are built into the WAR file are located.
Classes Dir	Type the directory for any additional controllers or guard condition classes in the application that are to be included in the application's WAR file.
HTML Directory	Type the directory for any HTML or JSP files that the WAR file needs. Any additional directories and files such as images and style sheets must also be located in this directory.

4 Click OK.

NOTE: Chart properties are not saved until you save the statechart.

Deploying the WAR File

When you are happy with the properties for the WAR file, you can deploy the application by selecting File > Save XML and Re-generate War.

This action creates the WAR file and places it the directory specified in the Destination Dir property in the WAR file properties. Some application servers have the ability to hot deploy the WAR file, for those that do not have this facility, you must restart the application server.

Running the Application

Generally, the URL to run the application is of the following format:

http://*serverName*:*portNumber*/*ApplicationName*/StateMachine

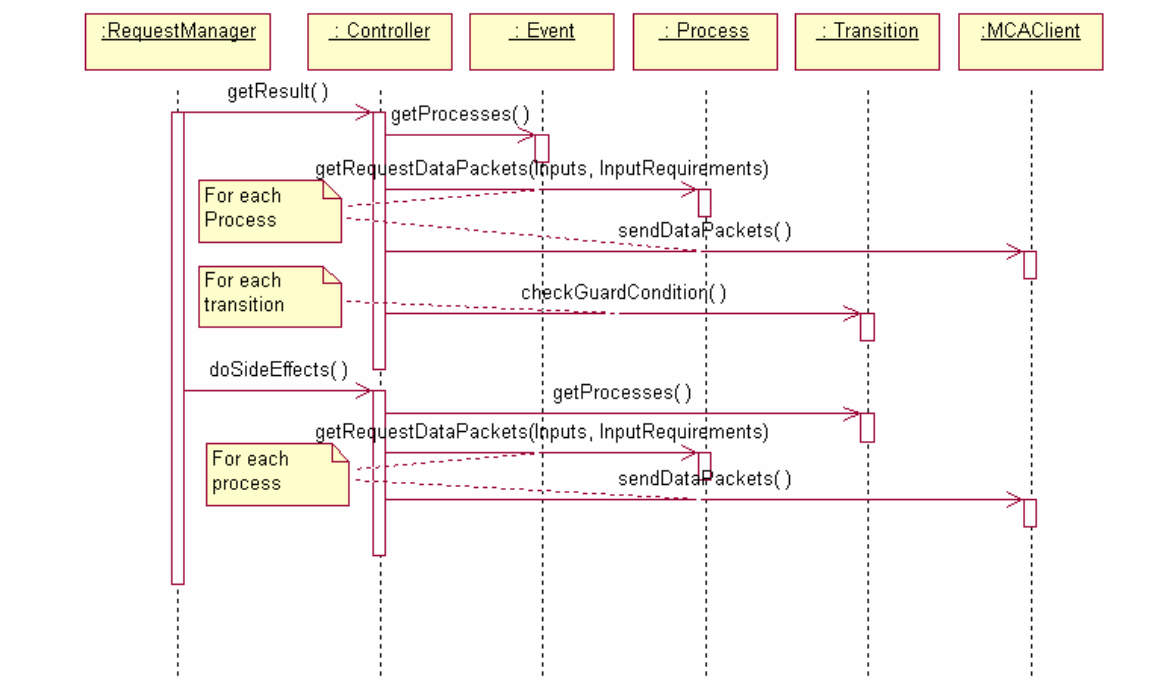
For example, if a file EbankingExample.xml is deployed on an application server installed locally, and on a port number of 8080, the URL is: http://localhost:8080/EBankingExample/StateMachine.

5 Defining Events with Processes and Guard Conditions

This chapter describes how to define complex events in the Screen Orchestrator. It describes the sequence of the state machine when handling an event, and how to add processes and useful guard conditions to that sequence to control a real user interface.

This chapter includes the following topics:

Figure 51. Sequence Diagram for Handling an Event



Handling an Event

When the state machine receives an event, it follows the sequence shown in Figure 51.

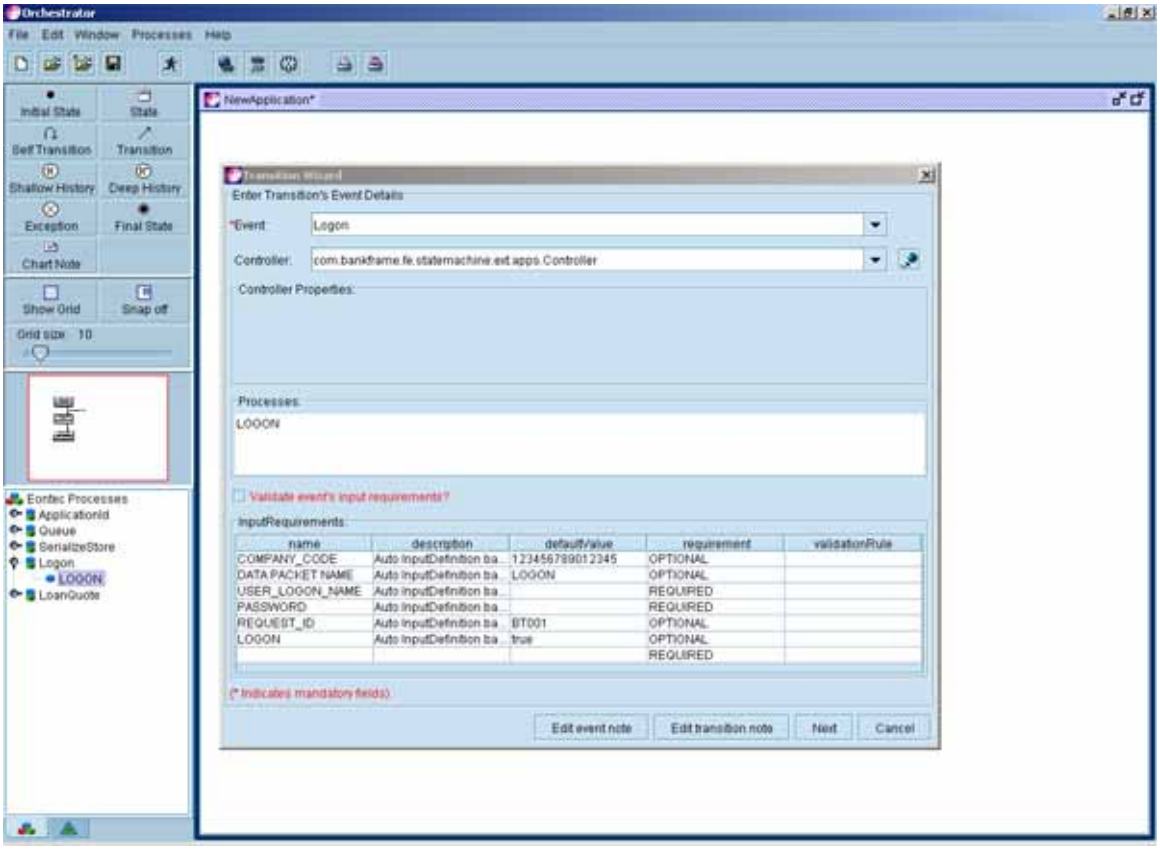
1. When the event is received, the state machine calls the Controller's getResult method.
2. The controller calls all the processes associated with the event.
3. The controller calls the checkGuardCondition method for each of the transitions. One of the guard conditions evaluates to TRUE. The getResult method returns that transition.
4. The state machine calls the doSideEffects method.
5. The controller calls all the processes associated with the transition.

NOTE: This is the sequence followed by the com.bankframe.fe.statemachine.ext.apps.Controller class. Other controller classes can behave differently.

Associating Processes with Events and Transitions

You associate processes with events and transitions by dragging the process from the process tree to the Transition Wizard (Figure 52).

Figure 52. Page 1 of the Transition Wizard

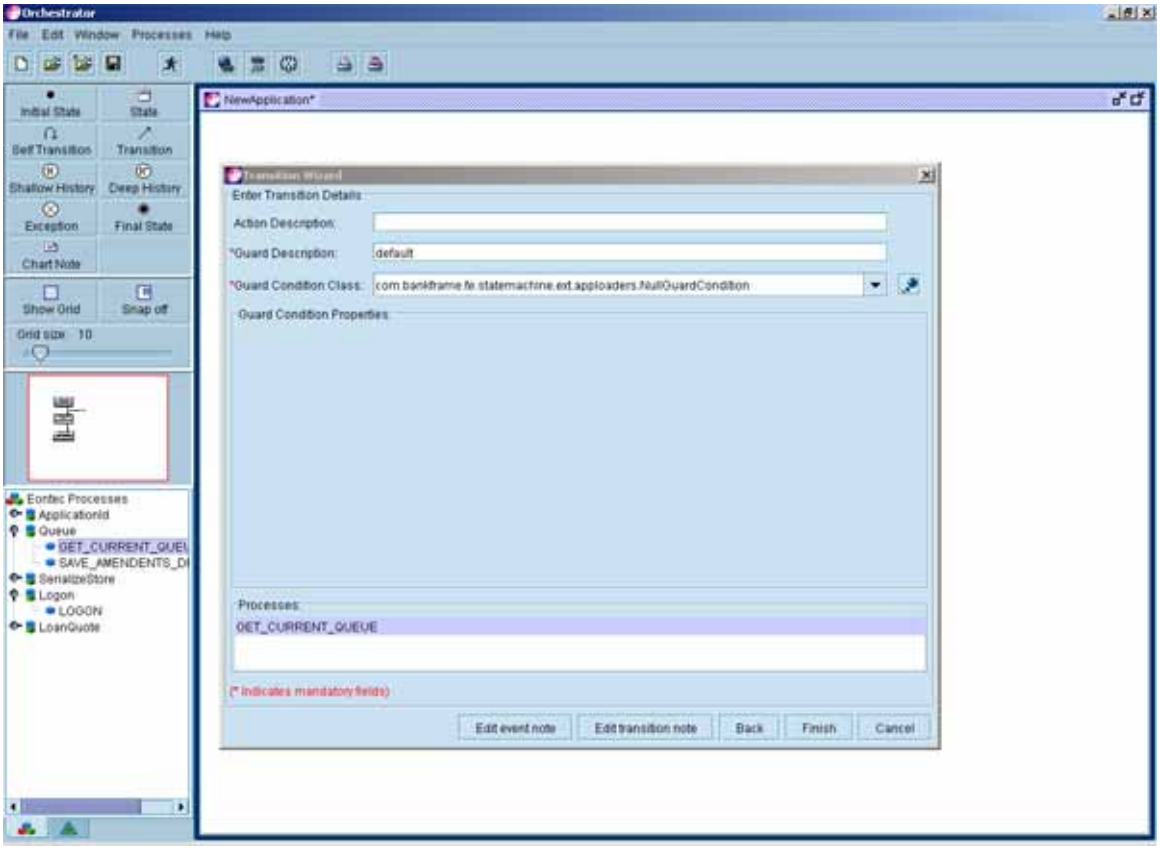


See Chapter 9 for details about loading and editing processes on the process tree.

The Transition Wizard has two pages. To associate a process with an event, drag the process onto the InputRequirements table in the first page of the wizard, as shown in Figure 52. To associate a process with a transition, drag the process onto the second page of the wizard, as shown in Figure 53.

Processes on the event (the first page) are called when the event is received from the user before any guard conditions are tested, as described in the Handling an Event topic. Processes on the transition are called after the transition's guard condition is tested, and only if the guard condition evaluates to true.

Figure 53. Page 2 of the Transition Wizard



Setting the Input Requirements

When you add a process to an event or transition, the parameters to that process become input requirements to the event. (The transition does not have independent input requirements, so all parameters to all processes on an event and all its transitions are considered input requirements to the event.)

Each input requirement (set in the table on the first page of the Transition Wizard) has the following attributes:

- **name.** The parameter name. In most cases, the parameter name is a DataPacket key, as used by a process that the event references.
- **description.** The optional parameter description.
- **defaultValue.** The optional default value.
- **requirement.** The requirement type, which must take one of the following values:
  - **REQUIRED.** Indicates that the user must supply the parameter value.
  - **OPTIONAL.** Indicates that the user can supply the parameter value.
  - **CONSTANT.** Indicates that the parameter value is always the supplied default value.
  - **CODED.** Indicates that customized code supplies the parameter. This option is used very rarely).
  - **PROCESS.** Indicates that the process definition supplies the parameter value. This option is used for parameters such as the DATA PACKET NAME and REQUEST_ID.
- **validationRule.** The optional validation rule for the input requirement. See Chapter 13 for more information.

NOTE: You can add input requirements that are not required by any process. For example, if you are using an InputBasedGuardCondition, you can add the parameter being tested by that guard condition to the input requirements.

Deleting Input Requirements

When events or states no longer use certain input requirements, you can remove them from the input requirements table.

To delete input requirements

- To delete a single input requirement, right-click on the appropriate entry in the table, and select Delete Selected Input Requirement.
- To delete all the input requirements, right-click on any entry in the table and select Delete All Input Requirements.

How the Request DataPackets are Built

To be sure the correct data are sent to the processes, it is important to understand how the Controller and Process objects build up the DataPackets that are sent through the MCA Services client.

The requirement type specified in the input requirements for each parameter determines the value that is used, according to these rules:

- **REQUIRED.** The value is taken from the request received from the user. If the value is not in the request, a null value is used.

- **OPTIONAL.** The value is taken from the request received from the user if possible. If the request does not contain a parameter of the correct name, the value is the default value for this input requirement.
- **CONSTANT**. The value is always the default value for this input requirement.
- **CODED.** The value must be provided by custom-written Java code in the controller.
- **PROCESS.** The value is taken from the DataPacket definition in the process.

The DataPackets are built to contain all the keys specified in the DataPacket definitions in the process.

Defining Guard Conditions

After you define processes and ensure that they receive the correct data from the user interface, the next step is to define the guard conditions on the event's transitions.

The basic rule you have to remember is: for every event, no matter what inputs are provided from the user or received from the processes, exactly one of the guard conditions on the event's transitions must be true. All other guard conditions must be false.

There is one exception to this rule: if an event has only one transition, that transition is always followed no matter what the guard condition.

There are various types of guard condition available by default in the Screen Orchestrator; the following sections describe these guard conditions.

NullGuardCondition

This is a guard condition that always returns an undefined value, neither true nor false. You use this guard condition only if the event has only one transition, or if you are using a controller that does not test guard conditions (such as the AutoViewController or SimpleController).

FixedValueGuardCondition

This is a guard condition that always returns either true or false as set in the Transition Wizard. You can use this guard condition during testing of an application to force it along a particular route to an area you need to test.

InputBasedGuardCondition

The InputBasedGuardCondition tests some value received from the user or in the user session.

You can set the properties described in Table 5.

Table 5. Guard Condition Properties for InputBasedGuardCondition

Field	Description
Input Must	Select whether or not the input must contain or must not contain the specified value.

Field	Description
Case Sensitive	Select whether or not the test should be case-sensitive.
Input Source	Select the input source: ■ ANY. Any input source. ■ Request ■ User Session ■ Visit For more information about these sources, see The Inputs Object on page 71.
Parameter Name	Type the name of the input.
Parameter Value	Type the value to test for.

ResultBasedGuardCondition

The ResultBasedGuardCondition tests the result from a process associated with the event.

You can set the properties described in Table 6.

Table 6. Guard Condition Properties for ResultBasedGuardCondition

Field	Description
Result Must	Select whether or not the result must contain or must not contain the specified value.
Case Sensitive	Select whether or not the test should be case-sensitive.
Process Name	The name of the process that was called.
Response DataPacket Name	The name of the DataPacket in the result DataPackets that should be tested.
DataPacket Key	The key part of the key/value pair within the DataPacket to be tested.
Value	The value part of the key/value pair within the DataPacket to be tested.

NOTE: The DataPacket name and key are always considered case sensitive, and only the value is tested for case sensitivity.

TimeoutGuardCondition

The TimeoutGuardCondition tests whether a timeout has resulted from a process associated with the event. You can set the properties described in Table 7.

Table 7. Guard Condition Properties for TimeoutGuardCondition

Field	Description
Timeout Parameter	Type the timeout parameter to test the process against; for example, TIMEOUT_STARTED would start counting the timeout from the time the process started.
Timeout Threshold	Type the number of milliseconds that would constitute a timeout. For example, type 150000 for a timeout of 2.5 minutes.

EmptyResponseGuardCondition

The EmptyResponseGuardCondition tests whether the response from a process associated with the event is an empty response.

The only field to complete is the Process Name field, in which you type the name of the process that was called.

Other Controller Classes

The preceding sections assume that you use the com.bankframe.fe.statemachine.ext.apps.Controller class as the event controller. There are other controller classes that you can use in different circumstances, and these are described in the following sections.

The SimpleController

The SimpleController class, com.bankframe.fe.statemachine.base.apps.SimpleController, is used to handle all trivial events. The SimpleController class can handle all events with just one transition and no associated processes. The SimpleController handles these trivial events faster than the main Controller class.

The AutoViewController

The AutoViewController class, com.bankframe.fe.statemachine.base.apps.AutoViewController, is used in conjunction with the AutoView, XSLTAutoView, or preview features to allow you to choose which transition to follow based on selecting from a list of available transitions.

Additional Controllers

The following additional controllers are available in the com.eontec.statemachine.helpers package:

- **ChannelClientController.** This controller provides a mechanism for specifying what channel client is used when executing processes. By default, it is set to use HttpClient, and in this mode it behaves in exactly the same way as the main Controller class.
- **DataCollectorController.** This controller is a subclass of the ChannelClientController, but it adds very special behavior in handled DataPackets. DataCollectorController can build multiple DataPackets from the input request values and can append these DataPackets to any request that is executed by any process specified by the event or transition.
- **MultipleRequestController.** This controller is used to handle more complex multiple DataPacket requests.
- **ClearUserSessionController.** This controller is a subclass of the ChannelClientController and is used to clear values in the Inputs object.
- **AddToUserSessionController.** This controller is a subclass of the ChannelClientController and is used to add values to the Inputs user session.

For more information about how to use these classes, see the MCA Services API documentation.

Custom Controllers and Guard Conditions

There may be some times when the standard processes, controllers, and guard conditions are not enough to meet the requirements of the user interface. In that case, you can write controller classes and guard conditions with custom code to meet the requirements. See Chapter 6, Writing Controller Classes and Chapter 7, Writing Guard Condition Classes for information about how to write custom code.

Adding Common Fields to Every Request

The state machine can add common items to every request sent to a Retail Finance server. For example, an application that uses Entitlements requires the following values to be contained within every request:

```
ENTITLEMENTS_CHANNEL_ID

ENTITLEMENTS_ACTOR_ID

ENTITLEMENTS_ACCESS_PROVIDER_ID

ENTITLEMENTS_ACCOUNT_NUMBER

ENTITLEMENTS_BRANCH_CODE
```

You configure the state machine to add these items to every request by setting values in the BankframeResource.properties file as follows:

```
‘# Common Request Items
```

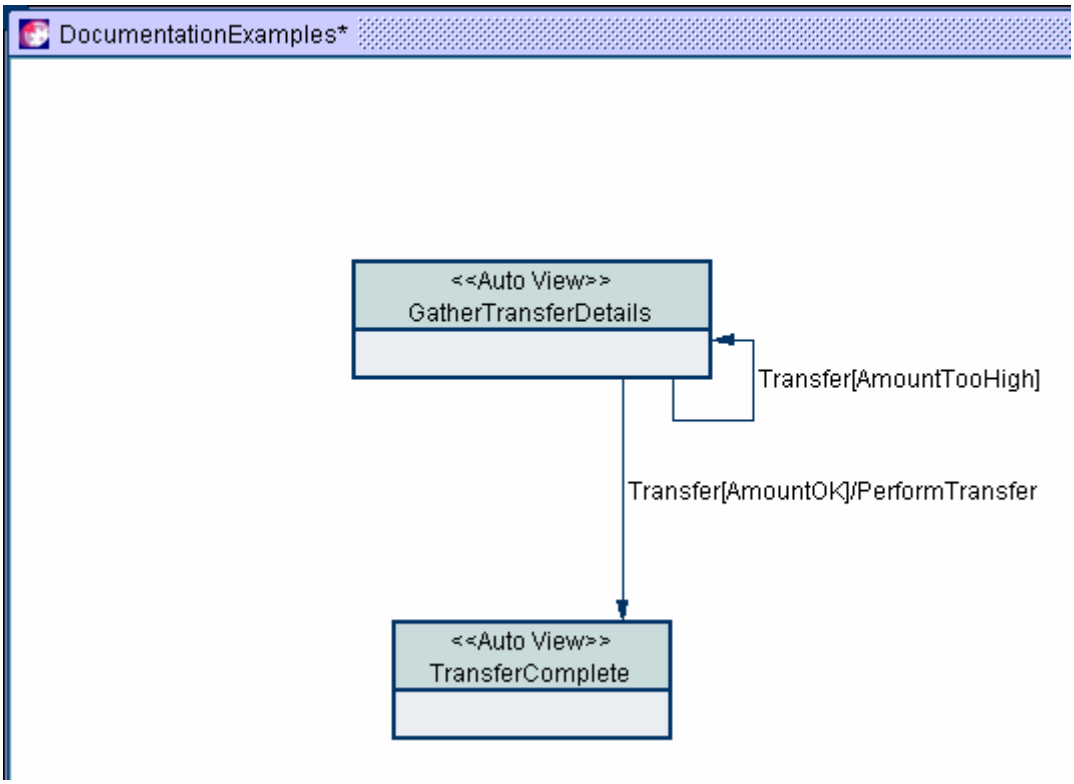
```
#########################

common.request.items.enable=true

common.request.items.fields=ENTITLEMENTS_CHANNEL_ID;
ENTITLEMENTS_ACTOR_ID;ENTITLEMENTS_ACCESS_PROVIDER_ID;
ENTITLEMENTS_ACCOUNT_NUMBER;ENTITLEMENTS_BRANCH_CODE'
```

These items are not required to be in the process definition, but you must add their key names to the common.request.items.fields value in the BankframeResource.properties file and then set the common.request.items.enable value to true. The values for these fields must be available in the Inputs object used by the state machine to hold user data. If the value for these fields is not in the Inputs object already, they are put into the request as blank values.

Figure 54. Worked Example



A Worked Example of Coding an Event

This section contains a worked example of how you might code an event with a process call, a result-based guard condition, and an action on one of the transitions. The resulting transition would appear as shown in Figure 54.

In the example, the user interface is intended to allow a user to request a funds transfer between two of their accounts. Users are restricted in the amount they can transfer in any given day, so the system

must check whether the amount chosen is above the limit. If it is not above the limit, the transfer must be performed and the user given a transaction record number.

To support this, two tier-1 methods are defined. The first method determines whether the amount specified is within the allowed range (CHECK_DAILY_LIMIT_FOR_ACCOUNT process), the second performs the transfer and returns the record number (MAKE_TRANSFER process).

You must specify two transitions:

1. Transfer[AmountTooHigh]. This transition is followed if the process does not return a DataPacket called AMOUNT_OK.
1. Transfer[AmountOK]/Perform Transfer. This transition is followed if the process returns a DataPacket called AMOUNT_OK. The transition also has an associated action, as it must actually complete the transfer requested by the user.

The two transitions make sure that no matter what is returned from the CHECK_DAILY_LIMIT_FOR_ACCOUNT process, one of the two guard conditions is true.

To specify the Transfer[AmountTooHigh] transition

1. Drag a self transition from the component palette onto the GatherTransferDetails state.
2. In the Event field type Transfer.
3. In the Controller field select com.bankframe.fe.statemachine.ext.apps.Controller.
4. Drag the CHECK_DAILY_LIMIT_FOR_ACCOUNT process onto the event.

   The InputRequirements box is filled with values taken from the process definition.
5. Set the requirement type for each of these inputs, as shown in the following table.

Name	Comment
AMOUNT	Select REQUIRED as this is supplied by the user.
BRANCH_CODE	Select REQUIRED as this is supplied by the user.
COMPANY_CODE	Select CONSTANT, as the company code does not change from the value supplied.
DATA_PACKET_NAME	Select PROCESS as this is particular to the process.
REQUEST_ID	Select PROCESS as this is particular to the process.
SOURCE_ACCOUNT_NUMBER	Select REQUIRED as this is supplied by the user.

6. Click Next.
7. In the Guard Description field, type something meaningful, such as AmountTooHigh.

8 In the Guard Condition Class field, select ResultBasedGuardCondition and complete the guard condition properties as described in the following table.

Field	Description
Result Must	Select: not contain the following.
Case Sensitive	Select false.
Process Name	Type CHECK_DAILY_LIMIT_FOR_ACCOUNT.
Result DataPacket Name	Type AMOUNT_OK.
DataPacket Key	Type DATA PACKET NAME.
Value	Type TRANSFER_OK. The data packet name will be tested, so the key is DATA PACKET NAME, and the value is the data packet name to test for.

9 Click Finish.

To specify the Transfer[AmountOK] / PerformTransfer transition

1 Drag a self transition from the component palette onto the GatherTransferDetails state, then click on the TransferComplete state.

2 In the Event field, select Transfer.

The controller, process and input requirements data are filled as for the Transfer[AmountTooHigh] transition.

3 Click Next.

4 In the Guard Description field, type something meaningful, such as AmountOK.

5 In the Guard Condition Class field, select ResultBasedGuardCondition and complete the guard condition properties, as shown in the following table.

Field	Description
Result Must	Select: contain the following. The guard condition on this transition is the opposite of the Transfer[AmountTooHigh] transition.
Case Sensitive	Select false.
Process Name	Type CHECK_DAILY_LIMIT_FOR_ACCOUNT.
Result DataPacket Name	Type AMOUNT_OK.
Datapacket Key	Type DATA PACKET NAME.

Field	Description
Value	Type TRANSFER_OK. The data packet name will be tested, so the key is DATA PACKET NAME, and the value is the data packet name to test for.

6 Enter an action description such as PerformTransfer, and drag the MAKE_TRANSFER process onto the Transition Wizard.

7 Click the Back button.

The InputRequirements table now has extra entries for parameters required by the MAKE_TRANSFER process:

- The destination account number VO (FINANCIAL_TRANSACTION_DESTINATION_ACCOUNT_VO_IMPL)
- Some common system attributes VO FINANCIAL_TRANSACTION_COMMON_ATTRIBUTES_VO_IMPL)
- The source account VO (FINANCIAL_TRANSACTION_SOURCE_ACCOUNT_VO_IMPL)

8 For each of the extra parameters, type REQUIRED in the Requirement field (the user must supply all of them).

9 Click Next, and then click Finish.

In this worked example you have:

- Created an event with two transitions
- Set guard conditions so that the correct transition is followed
- Added a process to be called before the guard conditions are checked and another process to be called if one of the transitions is followed.

Blocking Events from States

To block events from states other than the current state, the most important thing is that the Exception State is used in conjunction with setting the block.states key to true in the ScreenOrchestrator\resources\BankframeResources.properties file.

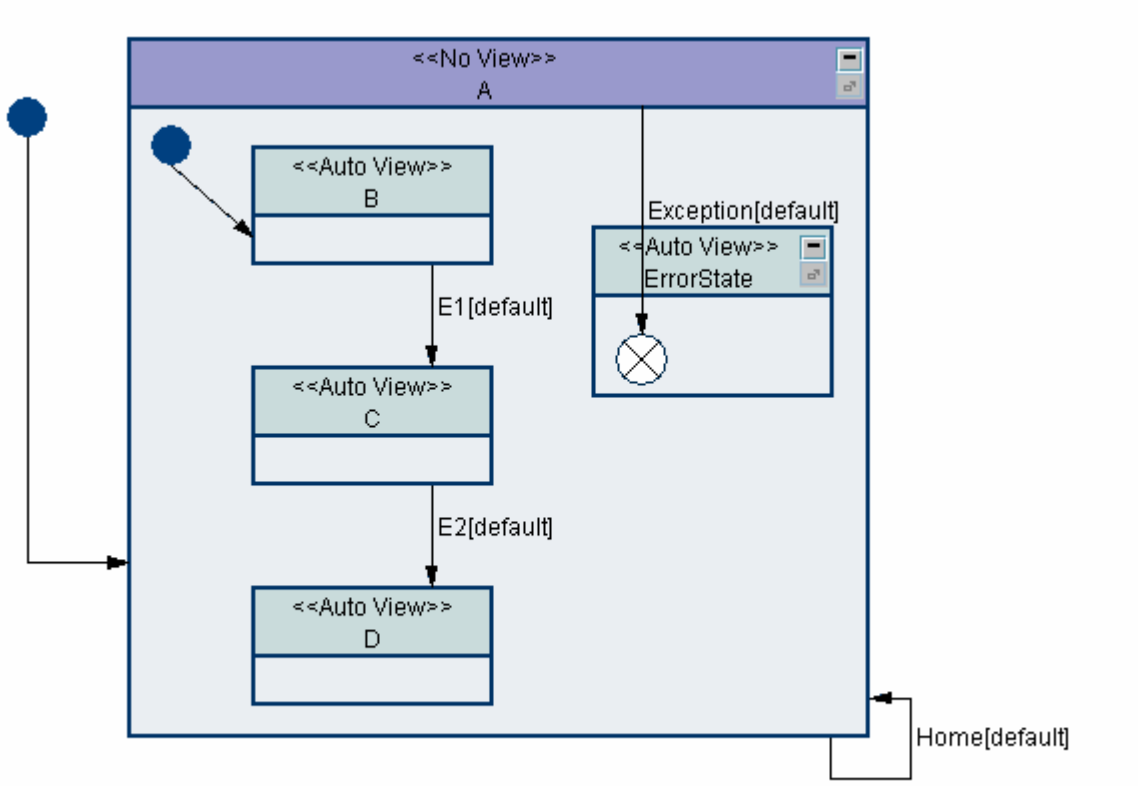
```
block.states=true
```

In the statechart shown in Figure 55, the states B, C, and D all inherit the Exception event. If an event is to be drawn to an exception state it must be called Exception (otherwise it does not work). When the C state is displayed, and the Back button is clicked on the browser, the B state is displayed. If the E1 event is then clicked, the state machine throws an exception, which forces the application to follow the Exception event and hence displays the ErrorState view. From the ErrorState the user can reenter the application through the Home event. The ErrorState can be anything you want, as long as the ExceptionState is a child of a viewable state.

The one scenario that you cannot block is when the user is on a state and clicks the Back button and then the Forward button on the browser. In this scenario the user can fire the event. The browser is back displaying the correct state and therefore the event is allowed.

The ScreenOrchestrator\resources\BankframeResources.properties file must set the block.states to true, but when building the WAR file, double check that the WAR file properties settings can find the relevant properties file. Also make sure that the directory specified in the Libraries Dir WAR property contains the current mca.jar file. For more information, see Configuring WAR File Properties for Statecharts on page 50.

Figure 55. Blocking Event from States Example



6 Writing Controller Classes

This chapter describes how to write Controller classes. It describes the responsibilities of such classes, the application programming interfaces (API) to be used, and some of the classes that are provided by default. The chapter also includes a section on adding your controller classes to the Screen Orchestrator.

This chapter contains the following topics:

The Responsibilities of a Controller

Controller classes are the implementation of the control logic of the user interface. Controllers are the event handlers, with every event having a Controller configuration of its own.

Controllers have two basic responsibilities:

- **To choose one of the event's transitions to follow.** When the Controller receives an event, it must look at the data supplied by the user and perhaps request information from the model. Based on this it must choose one of the event's transitions to follow. A Controller can choose a different transition each time the event is received, based on the business logic, the data supplied by the user, and the state of the model.
- **To perform any actions required by that transition.** Having chosen a transition, the Controller must perform any actions that are required by that transition.

As an example, using the worked example of the previous chapter (see Figure 54), an application might allow a user to make a funds transfer between two accounts if the amount transferred in any single day is less than some limit. In this case, the Controller first tests that the amount is under the limit and choose one of the two transitions based on that. If the controller chooses the transition to the TransferComplete state, it then performs the actions of completing the transfer.

The IController Interface

All Controller classes must conform to the com.bankframe.fe.statemachine.base.apps.IController interface. This interface defines two methods reflecting the two primary responsibilities of the Controller classes:

- **`IStateTransition getResult(RequestContext requestContext, IEvent event) throws StateMachineUserException`**

  This method represents the Controller's responsibility to choose between the transitions on the event. The RequestContext object contains all the information about the current user and request, while the IEvent object is the event that is to be processed. Typically this method is implemented as a series of calls to the model or tests on the input parameters, followed by a call to event.getTransition(transitionName) to get the IStateTransition object to return.

- **`void doSideEffects(RequestContext requestContext, IStateTransition transition) throws StateMachineUserException`**

  This method represents the Controller's responsibility to perform any actions required by the transition. The `requestContext` object is the same object as passed to the `getResult` method, while the transition object is the one returned by the `getResult` method. Typically, this method is implemented as a series of if/then/else if/ blocks testing the name of the transition. Each block contains the code to perform the required actions.

For more information about these methods, see the MCA Services API documentation.

Assuming appropriate processes were deployed, the event might be handled by the following code for the IStateTransition and doSideEffects methods:

```
      public IStateTransition getResult(RequestContext requestContext, IEvent event)
throws StateMachineUserException {

            try {

                  ChannelClient client = ChannelClientFactory.getChannelClient();

                  DataPacket requestData = new
DataPacket("CHECK_DAILY_LIMIT_FOR_ACCOUNT");

                  requestData.put(DataPacket.REQUEST_ID, TRANSFERS_REQUEST_ID);

                  requestData.put("COMPANY_CODE",
requestContext.getRequest().getParameterValues("COMPANY_CODE")[0]);

                  requestData.put("BRANCH_CODE",
requestContext.getRequest().getParameterValues("BRANCH_CODE")[0]);

                  requestData.put("SOURCE_ACCOUNT_NUMBER",
requestContext.getRequest().getParameterValues("SOURCE_ACCOUNT_NUMBER")[0]);

                  requestData.put("AMOUNT",
requestContext.getRequest().getParameterValues("AMOUNT")[0]);

                  Vector requestVector = new Vector(1);

                  requestVector.add(requestData);

                  Vector responseData = client.send(requestVector);

                  if
(((DataPacket)responseData.firstElement()).getName().equals("AMOUNT_OK")) {

                        return event.getTransition("AmountOK");

                  } else {
```

```
                        return event.getTransition("AmountTooHigh");
                    }
                } catch (ProcessingErrorException ex) {
                    throw new StateMachineUserException(ex);
                }
        }

        public void doSideEffects(RequestContext requestContext, IStateTransition
transition) throws StateMachineUserException {
                if (transition.getName().equals("AmountOK")) {
                    try {
                        ChannelClient client =
ChannelClientFactory.getChannelClient();
                        DataPacket requestData = new DataPacket("TRANSFER_FUNDS");
                        requestData.put(DataPacket.REQUEST_ID,
TRANSFERS_REQUEST_ID);
                        requestData.put("COMPANY_CODE",
requestContext.getRequest().getParameterValues("COMPANY_CODE")[0]);
                        requestData.put("BRANCH_CODE",
requestContext.getRequest().getParameterValues("BRANCH_CODE")[0]);
                        requestData.put("SOURCE_ACCOUNT_NUMBER",
requestContext.getRequest().getParameterValues("SOURCE_ACCOUNT_NUMBER")[0]);
                        requestData.put("DEST_ACCOUNT_NUMBER",
requestContext.getRequest().getParameterValues("DEST_ACCOUNT_NUMBER")[0]);
                        requestData.put("AMOUNT",
requestContext.getRequest().getParameterValues("AMOUNT")[0]);
                        Vector requestVector = new Vector(1);
                        requestVector.add(requestData);
                        Vector responseData = client.send(requestVector);
                    } catch (ProcessingErrorException ex) {
                        throw new StateMachineUserException(ex);
                    }
                }
        }
```

The SimpleController Class

The com.bankframe.fe.statemachine.base.apps.SimpleController class is an implementation of IController that is intended to control events with only one transition and no actions. In many applications, particularly Web applications, there are many events that are simply navigation events, that is, they take the user from one screen to another and do nothing else. The SimpleController can handle all of these events.

The Main Controller Class

The most commonly used Controller class is the com.bankframe.fe.statemachine.ext.apps.Controller class. This class is a complex and complete implementation of the IController interface that can use process data, input requirements, and guard condition data entered in the Screen Orchestrator to carry out all the responsibilities of a Controller.

This class follows the steps:

1. The `getResult` method calls all of the processes defined for the event in the correct order, using data from the user input, input requirements, and process definitions as appropriate to build up the `DataPackets` to be sent through MCA Services. Results from the process calls are added to the user session.
2. For each transition on the event, the Controller calls the `checkGuardCondition` method. Depending on the configuration of the transition, this could check the user inputs, the results from the processes, or other data, to decide whether the transition must be followed.
3. One of the transition `checkGuardCondition` methods should return `IGuardCondition.TRUE`. This transition is returned from the Controller's `getResult` method.
4. The `doSideEffects` method calls all of the processes defined for the transition in the correct order, using data from the user input, input requirements, and process definitions as appropriate to build up the `DataPackets` to be sent through MCA Services. Again, results from the process calls are added to the user session.

The steps in this process should be sufficient to handle the majority of all events that are included in an application user interface.

For those events that the steps cannot handle, you can extend the Controller class in various ways to add extra functions.

The Modified Controller Contract

The contract defined by the IController interface is a very general contract that you can use in any environment. The Controller class in the com.bankframe.fe.statemachine.ext.apps package provides a different definition of the getResult and doSideEffects methods geared more specifically to the Automated Methodology:

- IStateTransition getResult(IEvent event, Inputs inputs, RequestContext requestContext) throws StateMachineUserException, ProcessingErrorException
- void doSideEffects(IEvent event, IStateTransition transition, Inputs inputs, RequestContext requestContext) throws StateMachineUserException, ProcessingErrorException

For more information, see the MCA Services API documentation.

The IEvent, IStateTransition, and Inputs objects passed into these methods are all customized.

The IEvent and IStateTransition objects have a getProcesses method that returns an Iterator over all the processes associated with the event or transition. The IEvent object has a getInputRequirements method that makes available all the requirements and default values entered in the Screen Orchestrator. IStateTransition includes a checkGuardCondition method, to test whether the transition's guard condition has been met.

The Inputs Object

The Inputs object provides a single view of all the data provided by the user or recorded previously in the current user's session. It combines three different data sources:

- The Request, which contains the data entered by the user in the user interface before firing the current event.
- The Visit, which is generally empty, but can contain data placed there by another Controller or View. The Visit is intended to store data that might be needed by a View. The visit is stored by the state machine so that it can be reloaded if there is a return to the same result state through a History or History-star pseudo-state.
- The User Session, which can store data about the user that might be required anywhere in the application. This can include details like the user's name, active role, actor ID, and so on.

The Inputs class provides the following methods for getting and setting parameter values:

- **`Enumeration getParameterNames()`**

  This method provides an Enumeration over all the names of all the parameters in the three data sources.

- **`Object getparameter(String parameterName)`**

  This method provides the value of the named parameter; it looks first in the request, then the visit, and finally the user session.

- **`Object getparameter(String parameterName, int inputSource)`**

  This method provides the value of the named parameter in the specified input source. The input source must be one of `INPUT_SOURCE_ANY`, `INPUT_SOURCE_REQUEST`, `INPUT_SOURCE_VISIT` or `INPUT_SOURCE_USER_SESSION`.

- **`void setParameter(String parameterName, Object parameterValue)`**

  This method sets a parameter value in the request.

- **`void setParameter(String parameterName, Object parameterValue, int inputSource)`**

  This method sets a parameter in the specified input source.

Extending the Controller Class

In addition to the steps described, the Controller class calls a number of empty methods at different times during the processing. You can override these empty methods to add extra functionality.

The full sequence of method calls is:

- The framework calls getResult.
- The getResult method calls getResultPreProcess. Override getResultPreProcess if you need to manipulate the user inputs or perform any other tasks before the Controller does anything.
- For each process in the event:
  - The getResult method calls modifyProcess. Override modifyProcess to change the automatically-produced request DataPackets. The modifyProcess method can be called many times by getResult and by doSideEffects, so make sure you are modifying the correct process call!
  - The getResult method calls executeProcess.
- The getResult method calls chooseTransition. Override chooseTransition if you want to choose the transition yourself, instead of using the transition.checkGuardCondition methods. You must override chooseTransition if any of the transition guard conditions might return IGuardCondition.UNDEFINED.
- The getResult method calls getResultPostProcess. Override getResultPostProcess if you need to extract certain pieces of information from the process responses, or if you want to change the default behavior of adding the response data to the user session.
- The getResultPostProcess method calls addResultsToUserSession.
- The getResult method returns the chosen transition to the framework.
- The framework calls doSideEffects.
- The doSideEffects method calls doSideEffectsPreProcess. Override doSideEffectsPreProcess if there is anything you need to do before the actions are performed.
- For each process in the transition:
  - The doSideEffects method calls modifyProcess. This is the same modifyProcess method that is called by getResult, so be careful when overriding modifyProcess to modify only the processes that you need to modify.
  - The doSideEffects method calls executeProcess.
- The doSideEffects method calls doSideEffectsPostProcess. This is your last chance to change the behavior of the controller.
- The doSideEffectsPostProcess method calls addResultsToUserSession.

The sequence diagrams illustrated in Figure 56 and Figure 57 show the methods and calling sequence when a com.bankframe.fe.statemachine.ext.apps.Controller's getResult and doSideEffect methods are invoked.

Figure 56. The com.bankframe.fe.statemachine.ext.apps.Controller Sequence Diagram

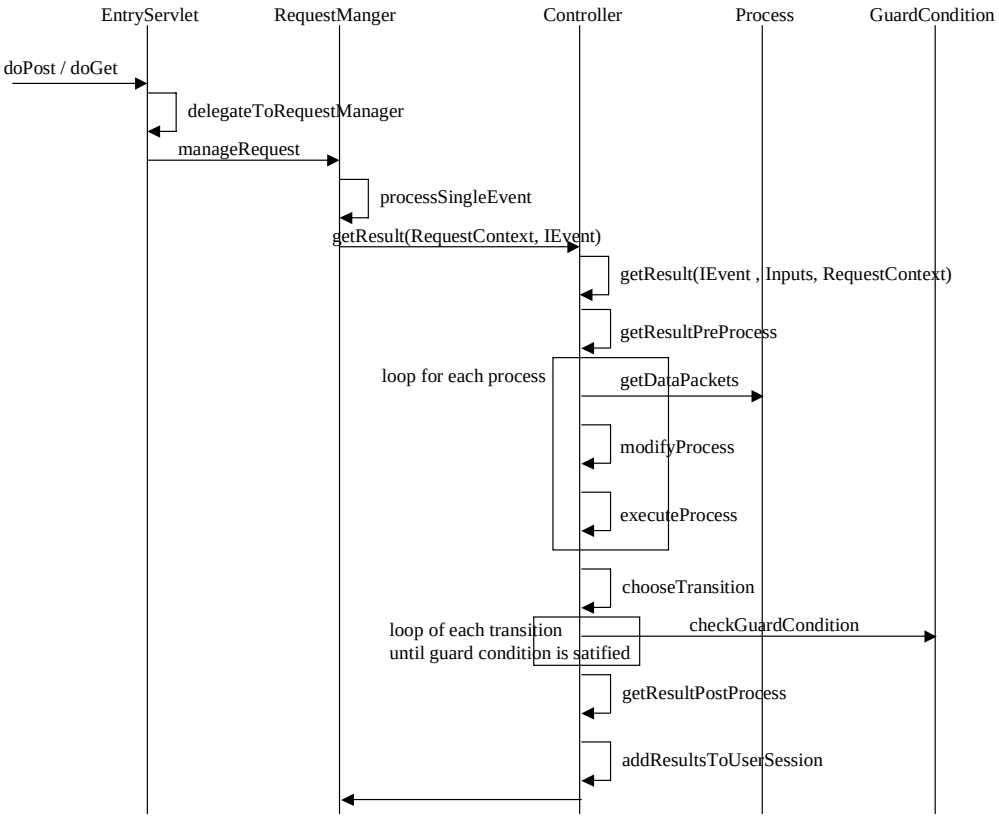
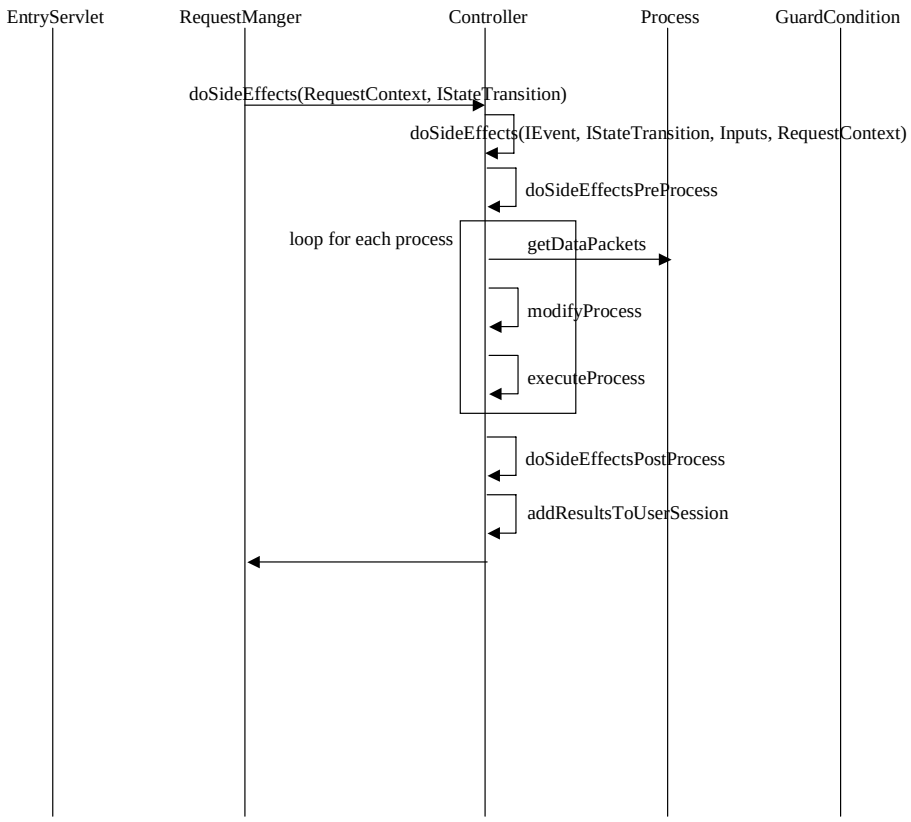


Figure 57. The com.bankframe.fe.statemachine.ext.apps.Controller Sequence Diagram (Continued)



For more information about all of these methods, including the method signatures, see the MCA Services API documentation.

Adding a New Controller to the Screen Orchestrator

When you have created a new controller class, you can use the class by typing the name into the Controller field in the Transition Wizard.

The controller classes that you can select in the Controller field are listed in the statechart.properties file, which contains the following controllers by default:

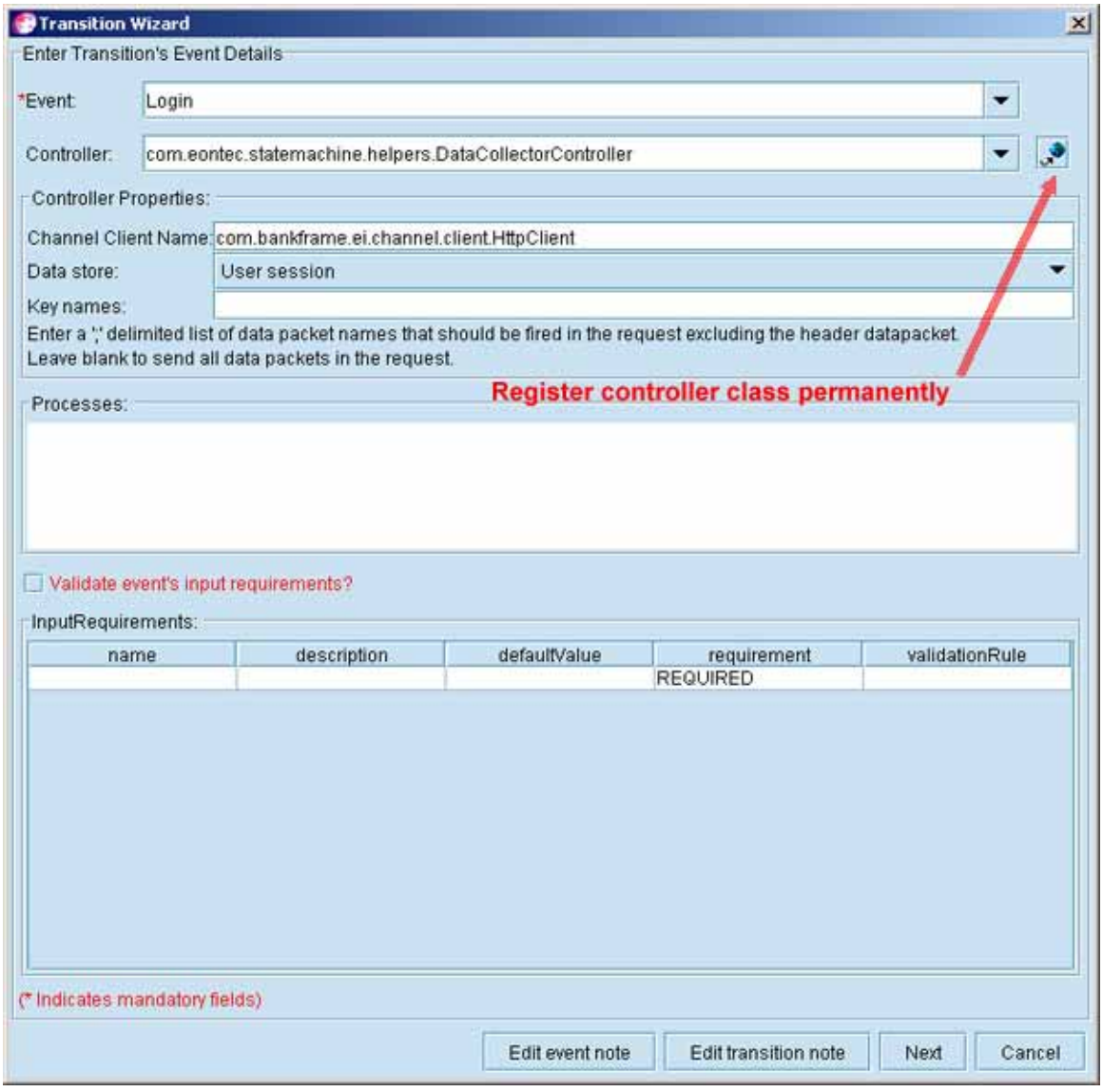
- com.bankframe.fe.statemachine.base.apps.SimpleController
- com.bankframe.fe.statemachine.base.apps.AutoViewController
- com.bankframe.fe.statemachine.ext.apps.Controller

- com.eontec.statemachine.helpers.ChannelClientController
- com.eontec.statemachine.helpers.DataCollectorController
- com.eontec.statemachine.helpers.ClearUserSessionController
- com.eontec.statemachine.helpers.AddToUserSessionController
- com.eontec.statemachine.helpers.MultipleRequestController

For more information about these controller classes, see Other Controller Classes on page 59.

To add a new controller, type the class name in the Controller field, and click the Register button to the right of the Controller field (Figure 58). This action adds the new class to the statechart.properties file. You can then select your new controller class from the list in the Controller field.

Figure 58. Transition Wizard with Register Controller Button Highlighted



Creating a Customizer for the Controller

The Controller Properties box on the Transition Wizard is managed by loading bean customizer classes for the controller class selected. This allows you to completely control how your controller looks in the Screen Orchestrator. The controllerProperties contain an entry for each attribute exposed by the bean.

If you create a customizer for your controller class, you must add it to the classpath setting in the *orchestrator-install-dir*\run.bat file, and restart the Screen Orchestrator; the Transition Wizard then loads your customizer.

For information about creating a customizer, see the JavaBeans API documentation.

7 Writing Guard Condition Classes

This chapter describes how to write guard condition classes. It describes the responsibilities of such classes, the application programming interfaces (API) to be used, and some of the classes that are provided by default. The chapter also includes a section on adding your guard condition classes to the Screen Orchestrator.

This chapter includes the following topics:

The Responsibility of a Guard Condition

A guard condition class has a very simple responsibility, namely to decide whether a transition should be followed in any given case.

The IGuardCondition Interface

All GuardCondition classes must implement the com.bankframe.fe.statemachine.ext.apploaders.IGuardCondition interface. This interface defines two methods for you to implement:

- **int checkGuardCondition(Inputs inputs, Vector processExecutionRecords, RequestContext requestContext, IStateTransition stateTransition)**

  This method must return either IGuardCondition.TRUE or IGuardCondition.FALSE. If it returns TRUE, the transition is followed, if it returns FALSE, the transition is not followed.

  The Inputs, RequestContext and IStateTransition objects supplied are the same as described for the Controller class. The processExecutionRecords vector contains com.bankframe.fe.statemachine.ext.apps.ProcessExecutionRecord objects, listing the details of all the processes executed by the Controller before calling checkGuardCondition.

- **void setGuardConditionProperties(Properties guardConditionProperties)**

  This method is called before checkGuardCondition. The guardConditionProperties object contains any information provided by the designer.

Adding a New Guard Condition to the Screen Orchestrator

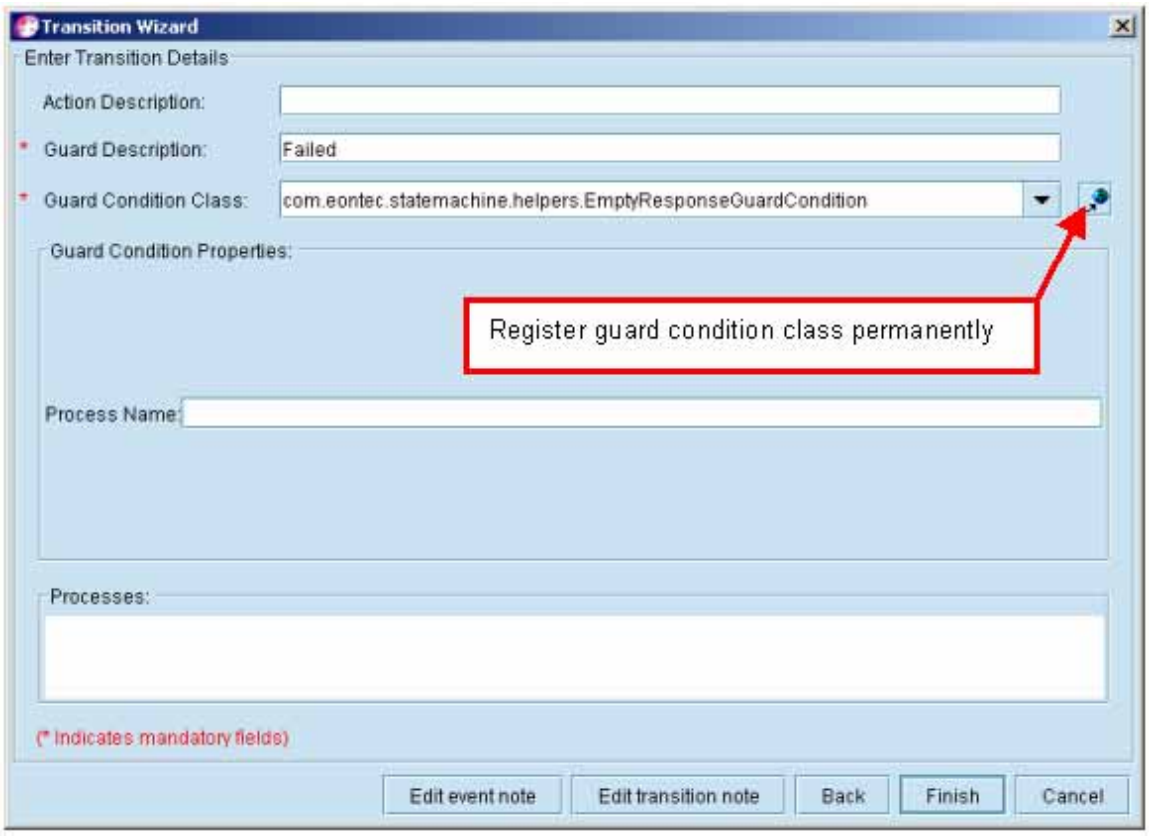
When you have created a new guard condition class, you can use the class by typing the name into the Guard Condition Class field in the Transition Wizard, and entering the guard condition properties.

The guard condition classes that you can select in the Guard Condition Class field are listed in the statechart.properties file, which contains the following guard conditions by default:

- com.bankframe.fe.statemachine.ext.apploaders.bean.ResultBasedGuardCondition
- com.bankframe.fe.statemachine.ext.apploaders.bean.InputBasedGuardCondition
- com.bankframe.fe.statemachine.ext.apploaders.bean.FixedValueGuardCondition
- com.eontec.statemachine.helpers.TimeoutGuardCondition
- com.eontec.statemachine.helpers.EmptyResponseGuardCondition

To add a new guard condition, type the guard condition class name in the Guard Condition Class field, and click the Register button to the right of the Guard Condition Class field (see Figure 59). This action adds the new class to the statechart.properties file. You can then select your new guard condition class from the list in the Guard Condition Class field.

Figure 59. Transition Wizard with Register Guard Condition Button Highlighted



Creating a Customizer for the Guard Condition

The Guard Condition Properties box in the Transitions Wizard is managed by loading bean customizer classes for the guard condition class selected. This allows you to completely control how your guard

condition looks in the Screen Orchestrator. The guardConditionProperties contains an entry for each attribute exposed by the bean.

If you create a customizer for your guard condition class, you must add it to the classpath setting in the *orchestrator-install-dir*\run.bat file, and restart the Screen Orchestrator; the Transition Wizard then loads your customizer.

For information about creating a customizer, see the JavaBeans API documentation.

8 Writing JSPs

This chapter describes the responsibilities of a Java Server Page (JSP) in the Screen Orchestrator framework, and the beans and tags available to help build JSPs. It includes the following topics:

Responsibilities of a JSP

In the Screen Orchestrator, views can be implemented as JSPs or as Swing panels. In the Screen Orchestrator framework, a JSP has two very simple responsibilities:

- **Displaying data to the user.** The JSP is required to format and display the data that the user expects to see, including all the formatting, framing, branding, and general prettiness that is required in the user interface.
- **Giving the user the opportunity to fire events.** For every state the user interface is in, there are events that the user can fire. The JSP must provide buttons, links, or similar widgets for the user, to allow events to be fired.

Within these two simple responsibilities, how you code the JSP is very flexible.

CAUTION: You must not include any tests in the JSP that result in loading new pages, forwarding, or redirecting the JSP. If there is ever a circumstance where you want to redirect or forward to another JSP based on some test in the JSP, you must change the statechart design so that the test is handled in a Controller class.

Getting Data into the JSP

The JSP is required to display data to the user. This data is made available to the JSP through the set of beans described in the following sections.

Inputs Bean

The Inputs bean contains all the data included in the request from the user, the current user session, and the current state visit.

Load the Inputs bean with the tag:

```
<jsp:useBean id="Inputs" scope="request"
class="com.bankframe.fe.statemachine.ext.apps.Inputs" />
```

The `scope="request"` attribute means that the newly created object is created and bound to the request object.

It is also useful to import the Inputs class to reference static members:

```
<%@ page import="com.bankframe.fe.statemachine.ext.apps.Inputs" %>
```

The Inputs bean provides the following methods for getting and setting parameter values:

- **`Enumeration getParameterNames()`**

  This method provides an Enumeration over all the names of all the parameters in the three data sources.

- **`Object getParameter(String parameterName)`**

  This method provides the value of the named parameter. The method looks first in the request, then the visit, and finally the user session.

- **`Object getParameter (String parameterName, int inputSource)`**

  This method provides the value of the named parameter in the specified input source. The input source must be one of `INPUT_SOURCE_ANY`, `INPUT_SOURCE_REQUEST`, `INPUT_SOURCE_VISIT` or `INPUT_SOURCE_USER_SESSION`.

- **`void setParameter(String parameterName, Object parameterValue)`**

  This method sets a parameter value in the request

- **`void setParameter (String parameterName, Object parameterValue, int inputSource)`**

  This method sets a parameter in the specified input source.

For more information about these methods, see the MCA Services API documentation.

ProcessExecutionRecords Bean

The ProcessExecutionRecords bean is a Vector of ProcessExecutionRecord objects, containing all of the processes executed while handling the current event. The responses from these processes, available as Vectors of DataPackets, contain all the data retrieved from the server by the Controller or View classes.

Load the ProcessExecutionRecords bean with the tag:

```
<jsp:useBean id="ProcessExecutionRecords" scope="request" class="java.util.Vector" />
```

State Bean

The State bean is the state that is to be displayed by the JSP. It is possible to use the same JSP to display different states, and the State bean gives you the current stateId or title.

Load the State bean with the tag:

```
<jsp:useBean id="State" scope="request"
class="com.bankframe.fe.statemachine.ext.apploaders.IState" />
```

View Bean

The View bean is the View class that is including the JSP.

Load the View bean with the tag:

```
<jsp:useBean id="View" scope="request"
class="com.bankframe.fe.statemachine.ext.connectors.servlet.JSPView" />
```

RequestContext Bean

The RequestContext bean contains other miscellaneous objects, including the state machine configuration, the application loader, user session, user session manager, and logger. You will probably not need this bean in most cases.

Load the RequestContext bean with the tag:

```
<jsp:useBean id="RequestContext" scope="request"
class="com.bankframe.fe.statemachine.base.RequestContext" />
```

Firing an Event from a JSP

The JSP does not need to supply the user with buttons or links to fire events. There are two distinct mechanisms that you can use to fire these events, as described in the following sections.

CAUTION: Do not mix the two mechanisms for firing events. If you start using one of these two approaches, keep using that approach. Any attempt to mix them causes events to fail.

Using the .jsm URL Extension

The statemachine servlet is configured to respond to all requests that end with .jsm. The servlet expects the stateId and event name to be supplied in the URL in the form *stateID.event name*.jsm.

You can use this URL format on both simple links and forms, using the following code:

```
<a href="<jsp:getProperty name="State" property="id" />.event.jsm"> event </a>
```

```
<form action="<jsp:getProperty name="State" property="id" />.event.jsm">
```

…

```
</form>
```

Replace ‘event; in the code samples with the correct event name.

Using the StateMachine URL

The statemachine servlet is also configured to respond to requests with the URL /StateMachine (relative to the Web application root). You can retrieve the absolute URL from the View bean.

You must supply two parameters called statemachineEventName and statemachineStateName with this URL. Use the following code as a guide:

```
<a href="<jsp:getProperty name="View" property="requestURL"
/>?statemachineStateName=<jsp:getProperty name="State" property="id"
/>&statemachineEventName=event">event</a>
```

```
<form action="<jsp:getProperty name="View" property="requestURL" />">
```

```
<input type="hidden" name="statemachineStateName" value="<jsp:getProperty name="State"
property="id" />">

<input type="hidden" name="statemachineEventName" value="event">

…

</form>
```

Replace ‘event’ in the code samples with the correct event name.

9 Integrating Processes in the Screen Orchestrator

This chapter describes the integration of processes within the Screen Orchestrator, which allows you to hook financial processes to the front-end components that the Screen Orchestrator generates.

This chapter includes the following topics:

About Integrating Processes

In terms of the Screen Orchestrator, a *process* is a unit of work performed within a deployed session bean. There can therefore be many processes within one session. In this sense, the Screen Orchestrator regards a session as an encapsulation of one or many financial processes. The manner in which the methods are called is controlled by the value of the DATA PACKET NAME key being passed in.

The integration of processes generally consists of two steps:

1. Importing processes from an external source, either manually or from an Automated Methodology model.
2. Associating these processes with events, or transitions, or both.

Importing Processes from an Automated Methodology Model

When you import processes into the Screen Orchestrator from an Automated Methodology model, the session name, process name, process signature, and process return type are automatically converted to the key/value pairings required for Request and Response DataPackets. Functional parameter objects, nonfunctional parameter objects, banking objects, and primary key classes used in the signature of a process are also automatically converted to the expected Request and Response DataPacket format. If a parameter is not defined as a result of the import process, it might be that the class type is not defined correctly in the Automated Methodology model.

To import processes from an Automated Methodology model

1. Right-click on the root node of the Siebel Processes Panel and select Import new sessions from model.
2. Click the button to the right of the text field and choose an XML file representing an Automated Methodology model.

   If you want, you can use the sample XML file, RetailAccount.xml, shipped with the application in the default XML folder.

   The process tree is populated with process information.

Manually Entering Process Information

You can add the process information manually.

To import processes manually

1. Right-click on the root node of the Siebel Processes Panel and select Add new session manually.
2. Type a process name, and click Next.

   The Set details screen is displayed.
3. Enter all relevant information for the Request DataPacket and the Response DataPacket.

   You can add or remove fields using the + and - buttons.

   You must supply a DataPacket name value with each process. This is used to name the node on the tree.
4. Depending on whether or not the response is a vector type or single DataPacket type, select the Vector check box.
5. There can be many processes in a session; if you want to add another process, click Next and enter the details for that process.
6. When you have entered the information for all processes, click Finish.

   The process information is added to the process tree.

Editing Processes

You can edit the details for a process at any time.

To edit the details of a process

1. Right-click on any process node.
2. Select Edit Process.
3. Edit the process details, and then click OK.

Deleting Processes from the Siebel Processes Panel

You can delete individual processes from the Siebel Processes panel, or you can delete all of the processes.

To delete an individual process

1. Right-click on the appropriate node in the Siebel Processes panel.
2. Select Delete Process.
3. Click Yes to confirm that you want to delete the process.

To delete all of the processes from the process list

1. Right-click on any node in the Siebel Processes panel.
2. Select Delete all Sessions.
3. Click Yes to confirm that you want to remove all the process definitions.

Assigning Processes to the Statechart

You can assign processes to a state or to a transition.

For more information on how processes are managed by the state machine framework after they are assigned, see Chapter 5, Designing Events with Processes and Guard Conditions.

Assigning Processes to a State

You assign a process to a state by dragging it to the state on the statechart.

To assign a process to a state

1. Drag the process from the process tree to the header of a state on the chart.
2. The Enter State Details screen is displayed, with the details of the process.
3. If you want to add further processes to this state, drag and drop them on to the Processes field or the Input Requirements field in the Enter State Details field.
4. Click OK to save the process details.

Adding Processes to a State Transition

You can also add processes to the transition details (as opposed to the transition event details). This means that the process is invoked as an action on the transition.

To add processes to a state transition

1. Make a new transition between two states.

   The Enter Transition Event Details screen is displayed.

2. Type the event name in the Event field.

3. In the Controller list, select a controller other than SimpleController or AutoViewController, which are used only for simple state navigation and do not invoke processes.

4. To add processes to this transition, drag and drop them from the process tree to the Processes field in the Enter Transition's Event Details screen.

   The processes are added to the transition's event details.

5. To add a process to the transition details (as opposed to the transition event details), click Next and drag and drop the processes on to the Processes field.

10 Advanced Drawing

This chapter describes the advanced drawing capabilities of the Screen Orchestrator. It includes the following topics:

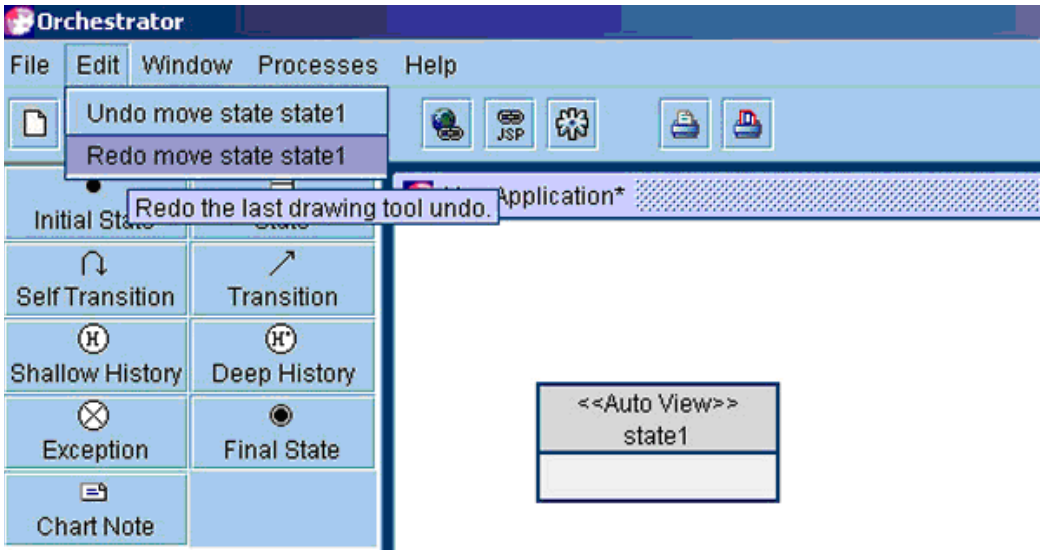
Undoing and Redoing Drawing Instructions

The Screen Orchestrator allows you to undo and redo drawing instructions. If you create, edit, or delete a state or transition, or move a state, you can undo that instruction. You can undo the last five drawing instructions.

You can also redo instructions that have been undone, so that the original instruction is done as originally specified.

To undo or redo instructions, you select the Edit > Undo or Edit > Redo menu options respectively in the Screen Orchestrator. The textual description of the Undo and Redo menu items change as you create, and undo, and redo instructions, as shown in Figure 60.

Figure 60. The Undo and Redo Menu Options



The textual description shows you the instruction that is undone or redone when you select the Undo or Redo menu option; in Figure 60, the instruction is “move state state1.” The Redo menu option is only available after you have selected Undo.

Copying, Cutting, and Pasting

In the Screen Orchestrator, you can cut and paste states to different parts of a statechart and cut and paste states between different statecharts. When you cut and paste a state, all its child states and transitions, apart from any transitions either entering or leaving the state, are also cut and pasted.

To cut and paste a state

1. Right-click on the state, and select Edit > Copy or Edit > Cut from the pop-up menu.
2. Right-click on the area to which you want to paste the state.
3. Select Paste to paste the state.

To cut and paste from one statechart to another

1. Open the statechart that you want to copy or cut from.
2. Right-click on the state, and select Edit > Copy or Edit Cut from the pop-up menu.
3. Open the statechart that you want to paste into.
4. Right-click on the area to which you want to paste the state.
5. Select Paste to paste the state.

Example of Cut and Paste Operation

This section illustrates the result of cutting and pasting a parent state and its child states to another parent state in a statechart. Figure 61 shows the statechart for a sample application.

Figure 62 shows the statechart after the Search state in the OpenAccount parent state is copied and pasted to the DrawDown parent state.

Figure 63 shows the statechart after the Search state in the OpenAccount parent state is cut and pasted to the DrawDown parent state.

Note that the Retrieve[default] transition is *not* copied as this transition leaves the Search state and hence is ignored when the copying is done. Also note that the initial state transition coming into the Search state is not copied. These transitions are removed in the case of cutting and pasting from the OpenAccount state.

Figure 61. The OpenAccount Search State

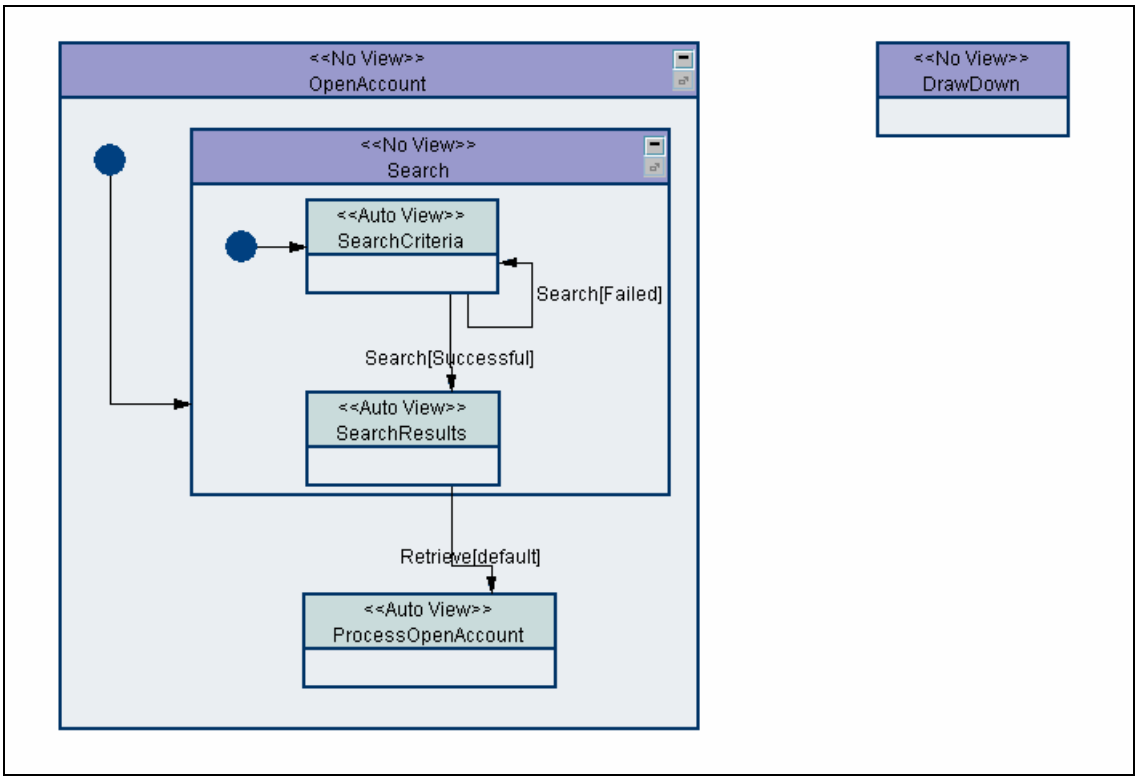


Figure 62. The Search State Copied and Pasted to the DrawDown State

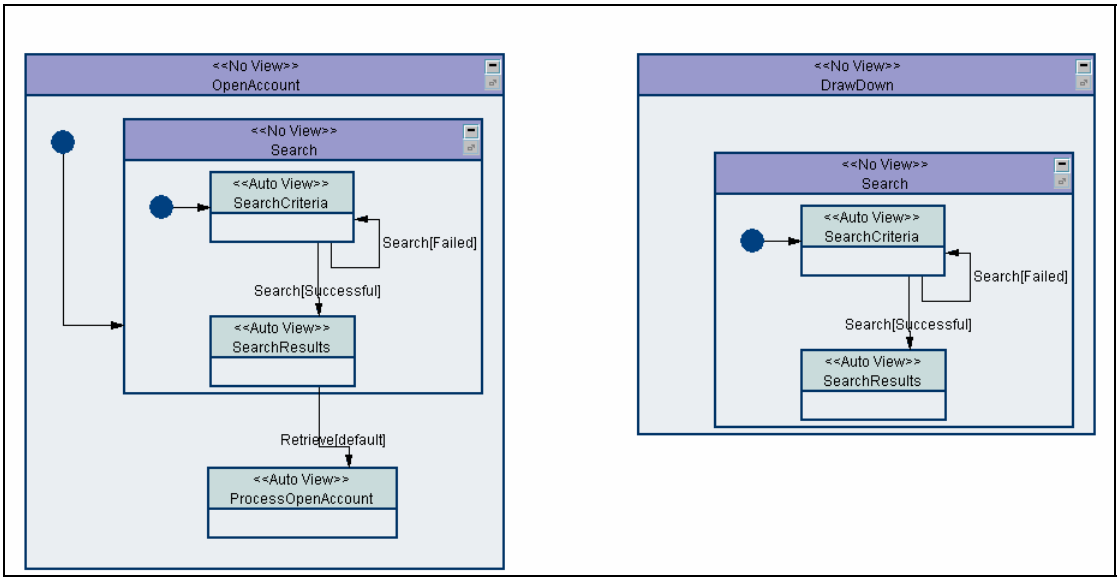
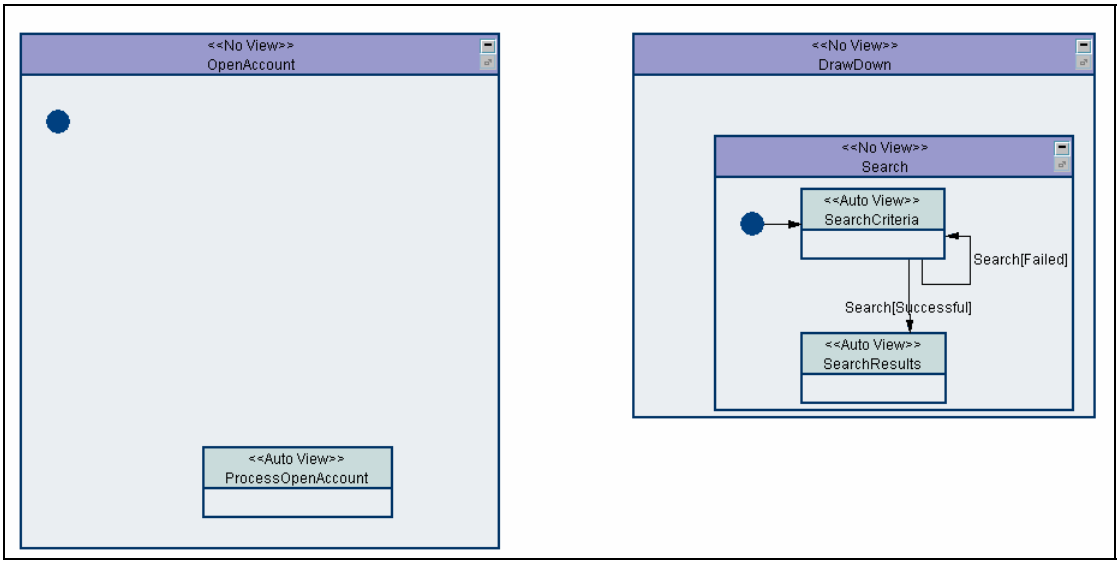


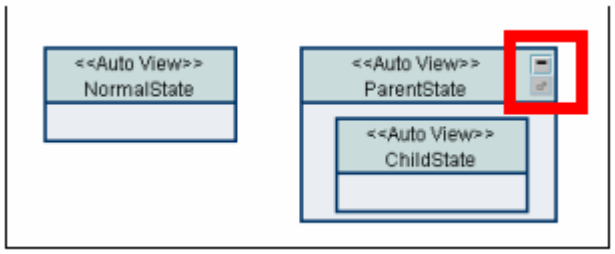
Figure 63. The Search State Cut and Pasted to the DrawDown State



Minimizing and Maximizing Parent States

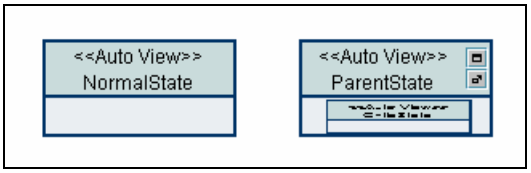
When a state becomes a parent state, two new icon buttons appear in the top right of the state's header, as illustrated in Figure 64.

Figure 64. A Parent State with Maximize and Minimize Buttons Highlighted



Clicking the top button minimizes or maximizes the parent state–the button toggles between the minimize and maximize functions. When you minimize the parent state, the bottom button is enabled as shown in Figure 65.

Figure 65. A Parent State Minimized

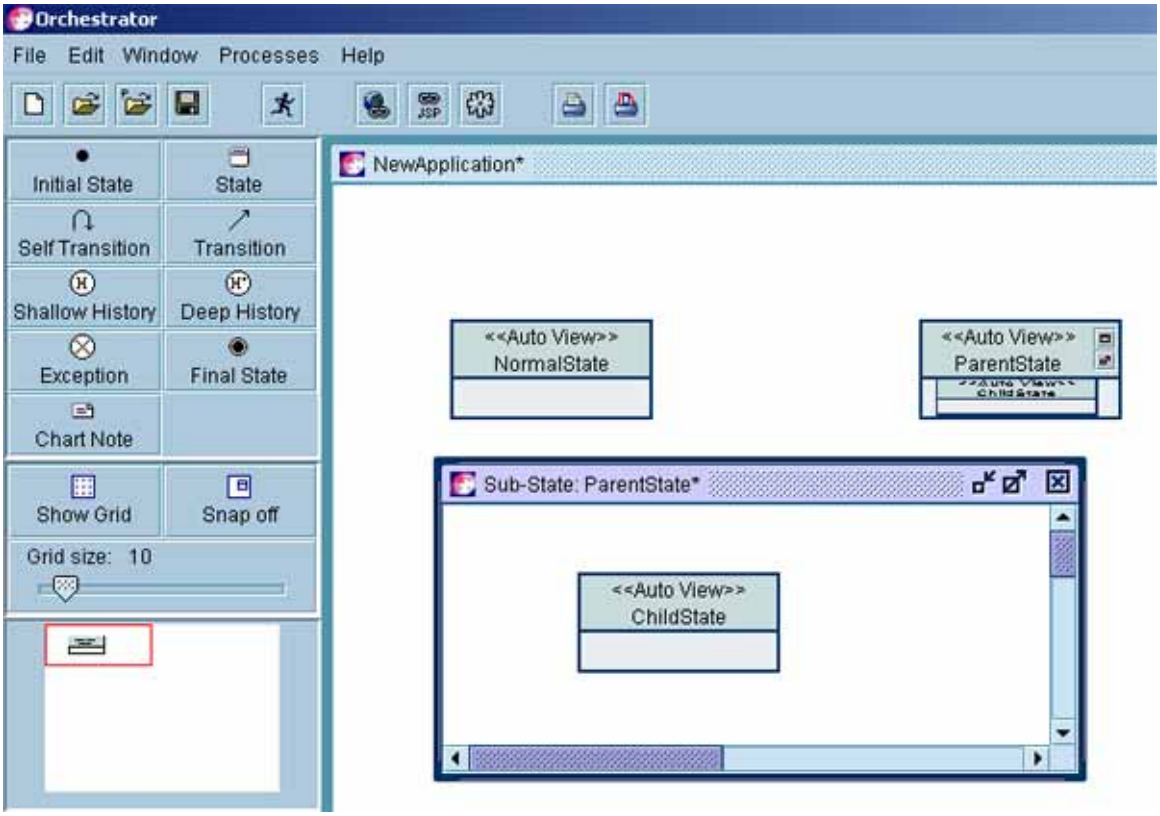


Opening Subcharts

The Screen Orchestrator allows you to open subcharts from the main statechart. This feature is extremely useful when the statechart becomes very large, as is the case for an application such as a teller or call center application (which might have over 200 states and transitions). Building the statechart becomes increasingly difficult as the application grows in size. Subcharts provide separate windows to allow you to more easily edit particular parent states, helping to reduce the clutter from the other states in the application.

Clicking the bottom button on the top right of the parent state opens the parent state in a new window, that is, as a subchart (see Figure 66).

Figure 66. A Parent State Opened as a Subchart



You can draw on a subchart as if it were a normal statechart. Figure 68 shows a new child state and transition added to the parent state subchart.

When you close a subchart, and maximize the parent state from the main statechart window, the states and transitions that you added to the subchart are added to the main statechart. This is shown in Figure 67.

NOTE: When a parent state is minimized, you cannot add states unless you maximize it.

Figure 68. A Subchart with New States and a Transition Added

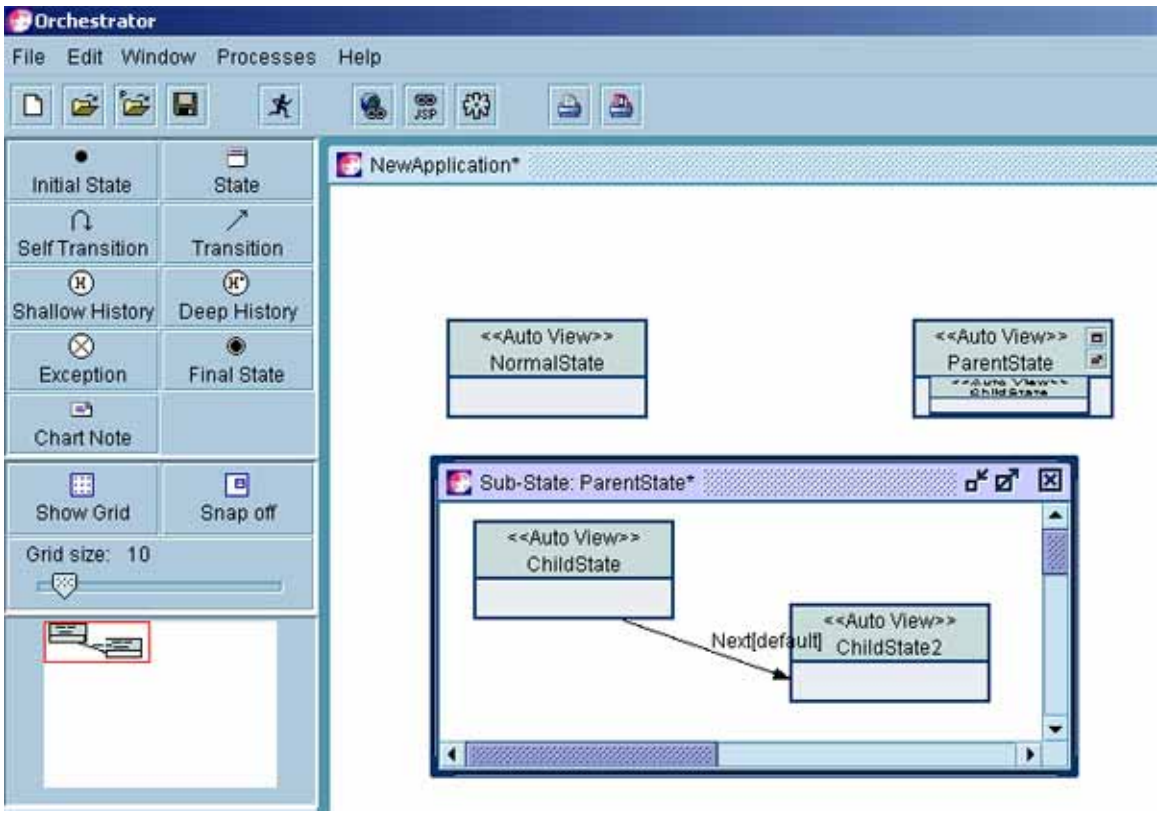
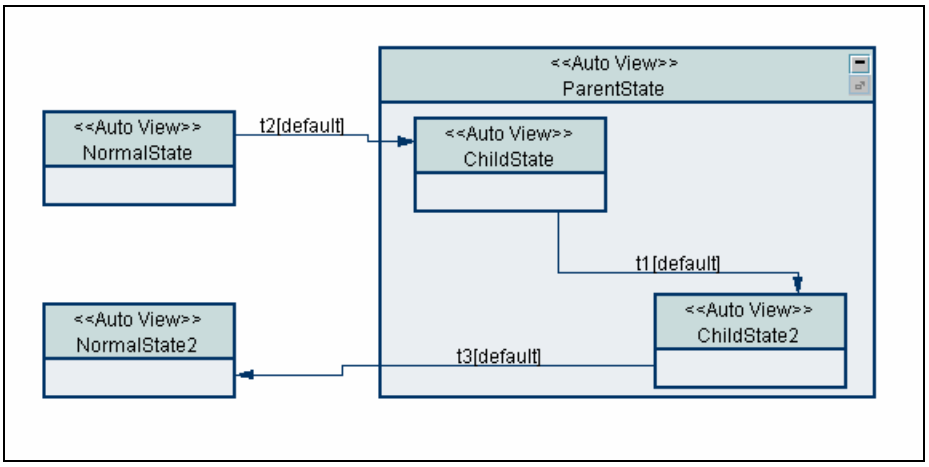


Figure 67. The Parent State Maximized with States and a Transition from the Subchart



Handling of Transitions Leaving and Entering Parent States

When parent states are minimized or opened in a separate subchart, the Screen Orchestrator shows the transitions to child states entering or leaving the parent state in a special way.

When parent states are minimized, a blue box is displayed in the upper-left corner of the parent state and for each transition a line is drawn directly between the other state and the box. For example, Figure 69 and Figure 70 show how transitions are drawn before and after a parent state is minimized.

Similarly, when a parent state is opened in a subchart, the subchart window indicates which transitions leave or enter the parent state by means of a small blue box in the upper-left corner of the subchart window, as shown in Figure 71.

Figure 69. Transitions to and from a Maximized Parent State

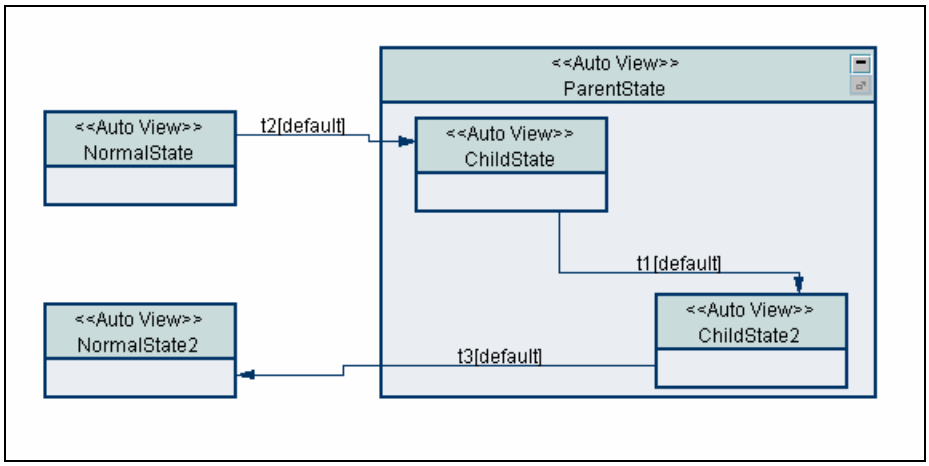


Figure 70. Transitions To and From a Minimized Parent State

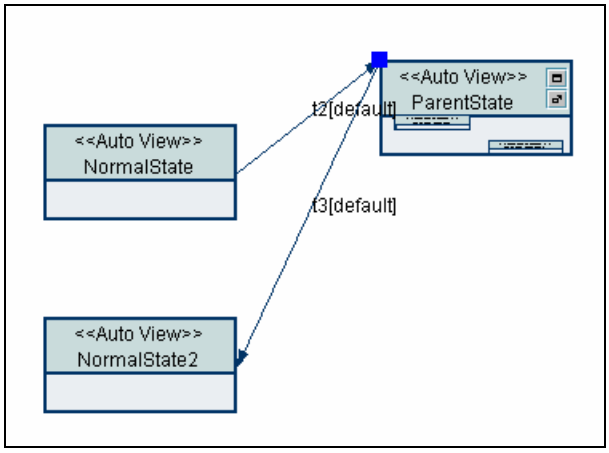
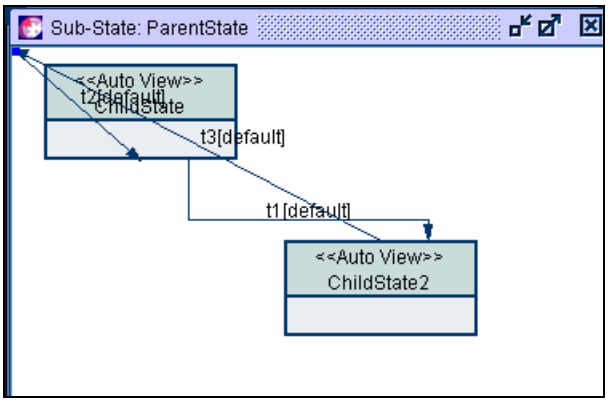


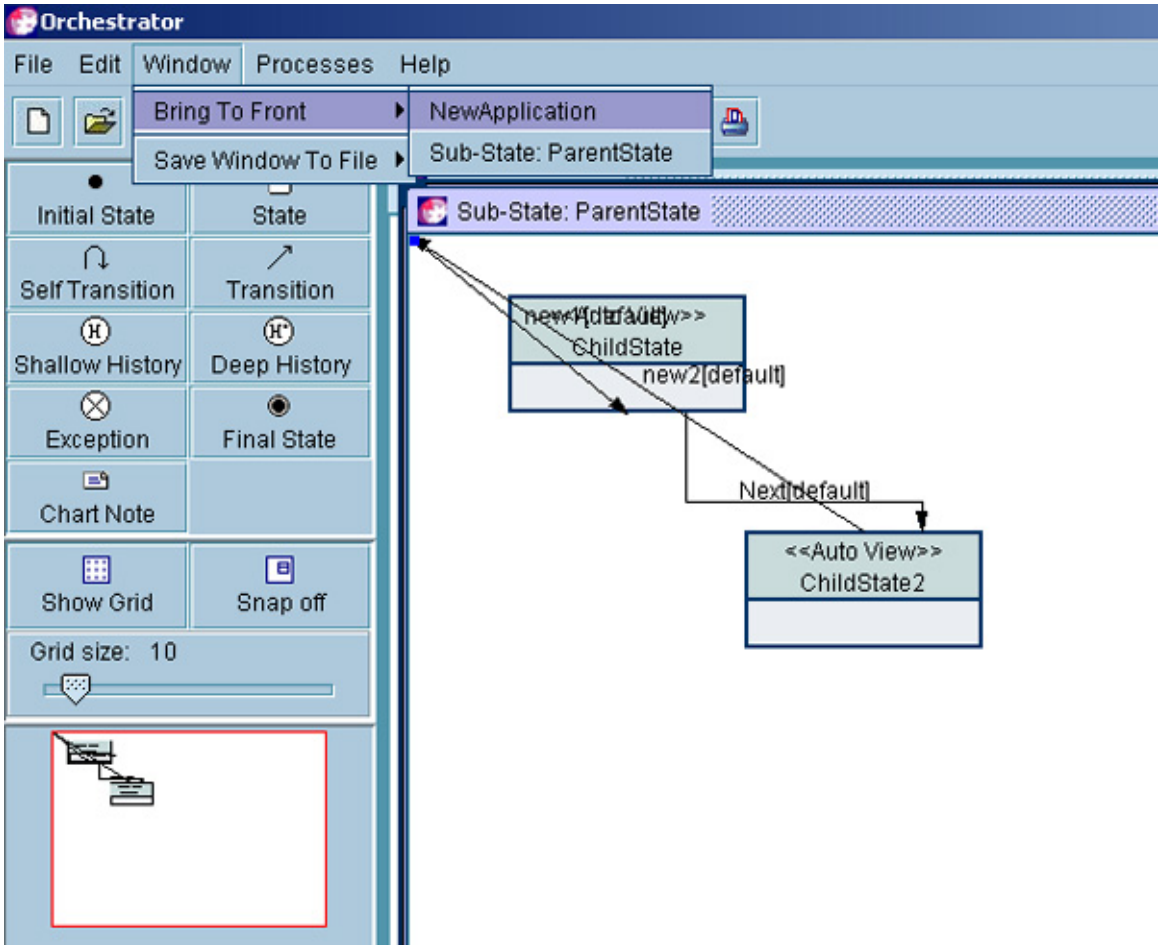
Figure 71. Transitions in a Parent State Opened in a Subchart



Bringing Subcharts to the Front of the Desktop

The Screen Orchestrator allows you to open any number of parent states as subcharts in the desktop area.

Figure 72. The Parent State



If multiple subcharts are open, you can bring any opened window to the front of the desktop by selecting the Window > Bring To Front from the menu bar. This provides a menu option for each window opened in the Screen Orchestrator desktop area. To bring a window to the front, select that window from the menu, see Figure 72.

Multiple User Support

You can use the Screen Orchestrator for defining very large applications involving you and multiple other users.

You can save parent states opened in subchart windows to separate files linked to the main statechart XML file. You can then edit and save changes to this parent state independently of other users.

To save a parent state to a separately linked file

1. Open the parent state as a subchart.
2. Navigate to the Window > Save Window To File menu option.

   A menu of states that you can save is displayed.
3. Select the substate that you want to save to a separate file.
4. Click Yes.

   The state's window title is updated as shown in Figure 74, and a special icon in the header indicates that its contents are contained in a linked file, as shown in Figure 73.

The format of the textual description in the Save Window To File menu option is:

Sub-State: *State Name*

for example:

Sub-State: ParentState.

The format of the filename of the ParentState is:

Application Name.State Name.part

for example:

NewApplication.ParentState.part.

If you rename the application, all its linked state files are also renamed to maintain the link to the main application statechart file.

NOTE: When you edit the parent state in the subchart window, the application statechart is only updated when you either save the changes or close the window.

To open a linked file, select File > Open Part on the menu bar.

Figure 74. A Parent State Window Title Updated to Show the Filename of the Linked File

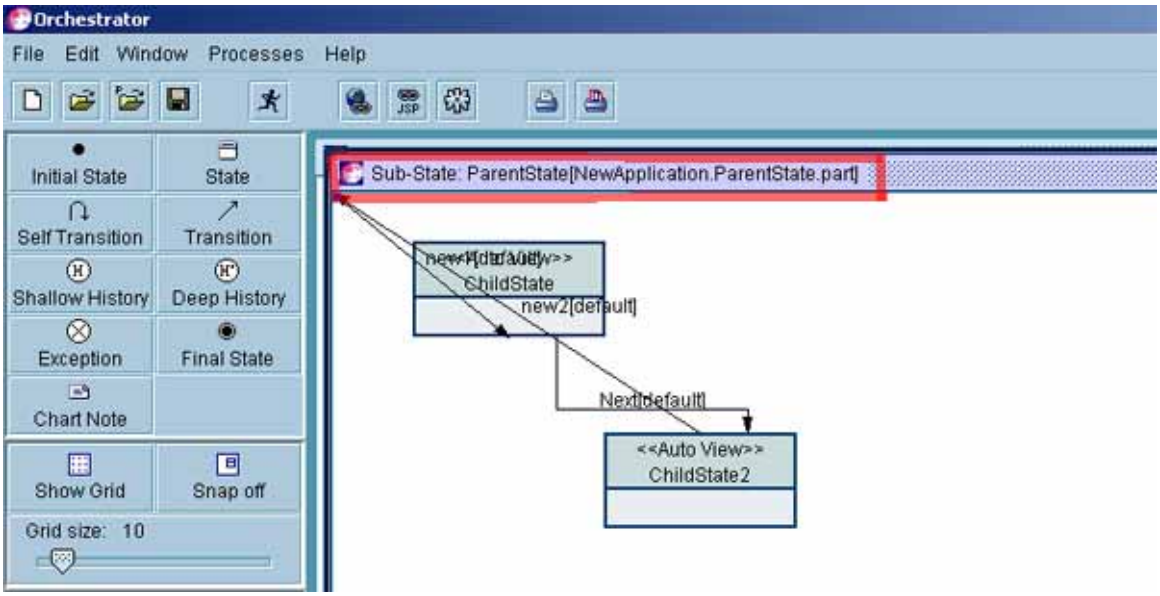
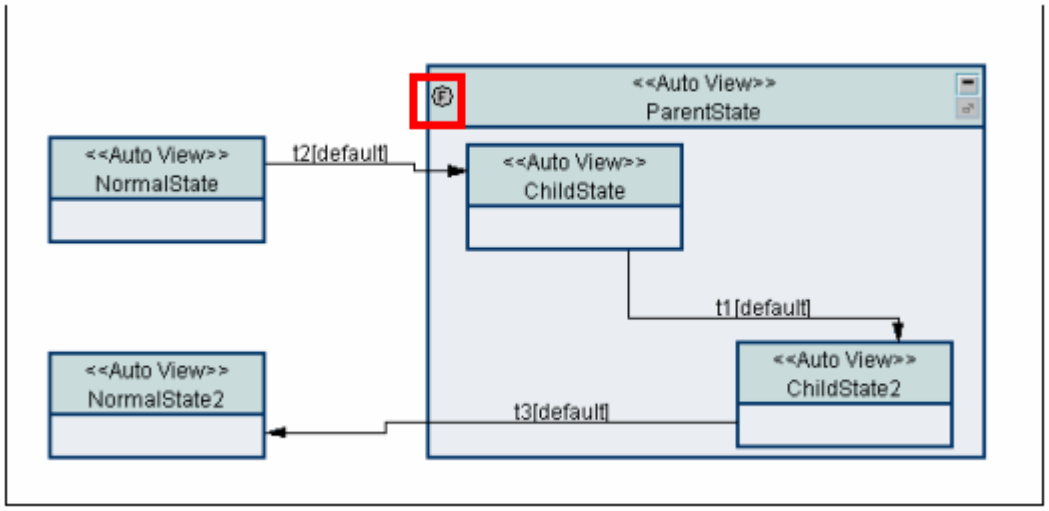


Figure 73. The Parent State Displaying the Special Icon



Users Working on the Same Files

The Screen Orchestrator is very much like any other tool in that it produces a number of flat files. Any user can edit these files and overwrite other user's changes. The only additional feature provided by the Screen Orchestrator is that it does not allow statechart files to be saved if they are marked as read-only. It is your responsibility to make sure that your files are under version control and that you do not overwrite another user's work.

Different users can work on the separate part files that are linked to the main application file, which links all the part files together. A main application file affects all the part files and should only be maintained and modified by a single person.

11 Forward Engineering

This chapter describes the Forward Engineering functionality of Screen Orchestrator that allows you to propagate design model changes to existing statecharts.

This chapter includes the following topics:

About Forward Engineering

The Screen Orchestrator can compare a statechart XML file with a design model XML file and detect any changes made in the design model to tier 1 methods, including addition or removal of parameters, and renaming of parameters or the method itself.

Forward Engineering is facilitated because every tier 1 method and parameter has a Rational Rose Universal Unique Identifier (UUID), which is output with the design model XML file. When you import the design model XML file, the Screen Orchestrator builds a process list containing these UUIDs, and updates the Siebel Processes panel accordingly.

Whenever a new version of the design model is produced, and you are aware that tier 1 methods used by the statechart have been altered, you must update existing statecharts with the new process list produced when you import the design model XML file.

Process of Updating Statecharts

Statecharts that were created using previous versions of the Screen Orchestrator do not contain UUIDs, and a comparison between the design model XML and the statechart XML is not possible. There are therefore two scenarios for updating statecharts, depending on whether they contain UUIDs.

You can update statecharts either by using options in the Forward Engineering menu in Screen Orchestrator, or by using the equivalent commands. In the updated statechart, references to renamed tier 1 methods, and references to parameters that have been removed, added, or renamed, are all updated. However, you must manually update the statechart for any tier 1 methods that have been removed.

Statecharts with No UUIDs

If the statecharts do not contain UUIDs, you must:

1. Produce a report of the differences between the statechart and the design model, and resolve any inconsistencies.
2. Initialize the statechart with UUIDs.

3 Update the statechart with the design model changes.

Statecharts with UUIDs

If the statechart does contain UUIDs you only need to update the statechart with the design model changes.

Updating Statecharts Using the Forward-Engineering Menu Options

Before you update statecharts, use the Model Exporter to produce the design model XML file, see the *Siebel Retail Finance Design Tools Guide* for more information.

Updating a Statechart That Does Not Contain UUIDs

If you have statecharts that do not contain UUIDs you must first import the design model XML file and produce a comparison report. The comparison is made purely on the basis of DataPacket key names and process names.

After you manually correct any inconsistencies between the process list and the statechart, you must initialize the statechart. This initialization copies all UUIDs from the processes list into the statechart, so that future comparisons can be automated.

Finally, you must update the initialized statechart with the changes from the design model.

To update a statechart that does not contain UUIDs

1 Navigate to the Processes > Import new sessions from model > Import Process Definitions screen.

2 Enter the location of the design model XML file, and click OK.

The Siebel Processes panel is updated.

3 Open the statechart to be updated.

4 Navigate to the Forward Engineering > Log Differences > Generate Forward Engineering Report screen.

5 Enter the location for the report file, and click OK.

6 Using the report, correct any inconsistencies between the statechart and the design model XML.

7 Click Forward Engineering > Initialize State Chart.

A new version of the statechart is created in the \resources\xml\initialize directory. The currently open statechart is not updated to prevent loss or corruption of data caused by mistakes that might have been made in the manual updating.

8 Close the current statechart and open the new initialized version of the statechart.

9 Click Forward Engineering > Update State Chart.

The Screen Orchestrator compares the processes in the statechart against the process list based upon UUIDs, and the updated statechart is saved in the \resources\xml\merge directory. If Screen Orchestrator finds that a tier 1 method currently in use has been removed from the model, it generates ands displays a report.

10 Update the new statechart with any methods reported as removed.

Updating a Statechart That Does Contain UUIDs

If you have statecharts that already contain UUIDs, you only need to import the design model XML file and update the statechart.

To update a statechart that does contain UUIDs

1 Navigate to the Processes > Import new sessions from model > Import Process Definitions screen.

2 Enter the location of the design model XML file, and click OK.

The Siebel Processes panel is updated.

3 Open the statechart to be updated.

4 Click Forward Engineering > Update State Chart.

The Screen Orchestrator compares the processes in the statechart against the process list based upon UUIDs, and the updated statechart is saved in the \resources\xml\merge directory. If Screen Orchestrator finds that a tier 1 method currently in use has been removed from the model, it generates ands displays a report.

5 Update the new statechart with any methods reported as removed.

Updating Statecharts Using Commands

You can use the Forward Engineering functionality in batch mode from the command line.

In each of the commands, you specify the locations of the design model XML file (processes.xml) and the statechart XML file. For the statechart XML file, you can specify either a statechart file or a directory. In the case of a directory, Screen Orchestrator processes all of the valid statecharts in the directory and its subdirectories.

Updating a Statechart That Does Not Contain UUIDs

If the statecharts do not contain UUIDs, you must first initialize them with UUIDs.

To update a statechart that does not contain UUIDs

1 Generate a comparison report by entering the command:

```
java com.eontec.statechart.forwardeng.ForwardEngApp –report statechart_location
processes.xml_location
```

A comparison report is generated in the \resources\xml\reports directory.

2 Using the report, correct any inconsistencies between the statecharts and the design model XML.

3 Initialize the statecharts with UUIDs, by entering the command:

`java com.eontec.statechart.forwardeng.ForwardEngApp –setup` *`statechart_location processes.xml_location`*

The initialized statecharts are created in the \resources\xml\initialize directory.

4 Updated the initialized statecharts by entering the command:

`java com.eontec.statechart.forwardeng.ForwardEngApp` *`statechart_location processes.xml_location`*

The updated statecharts are saved in the \resources\xml\merge directory.

Updating a Statechart That Does Contain UUIDs

If you have statecharts that already contain UUIDs, you only need to update the state chart. Enter the command:

`java com.eontec.statechart.forwardeng.ForwardEngApp` *`statechart_location processes.xml_location`*

The updated statecharts are saved in the \resources\xml\merge directory.

12 Writing A Swing Application

This chapter describes briefly what is required for applications deployed using the Swing application programming interface (API) to use the state machine. It includes the following topics:

About Writing a Swing Application

The javax.swing package, which is part of the Java Foundation Classes, provides a set of graphical user interface (GUI) components. These components are commonly known as Swing components. In contrast with the Abstract Windows Toolkit (AWT), which Swing has largely supplanted, the Swing components are written entirely in Java.

The state machine framework supports applications developed using the Swing API as well as those developed as Web applications using the Servlet API.

Designing Swing applications is virtually identical to designing Web applications. You should give the same concern to the behavior and flow control through the application in the statechart, and you can use the same Controller and Process integration classes.

Deploying a Swing application using the state machine requires you to use the classes and interfaces in the com.bankframe.fe.statemachine.ext.connectors.swing package.

The structure of your application is:

- An application main class acting as a Window or Applet for the application. The main class contains:
  - A ViewContainer, which is a container within which all the application views are displayed
  - A StateMachineEventDispatcher, which listens for StateMachineEvents fired from your views
- The View classes, which all implement IView and StateMachineEventSource. Whenever the user does anything that triggers an event on the statechart, the view must fire a StateMachineEvent.

Processing Sequence

The sequence for processing an event is:

1. The view class fires a StateMachineEvent.
2. The StateMachineEventDispatcher receives the event and forwards it to the RequestManager in the state machine.
3. The RequestManager returns the new view class.
4. The StateMachineEventDispatcher registers itself as a StateMachineEventListener on the view, so that it receives the next StateMachineEvent that is fired.
5. The StateMachineEventDispatcher passes the view to the ViewContainer.
6. The ViewContainer displays the new view.

The important classes and interfaces are described in the following sections.

StateMachineEvent Class

The StateMachineEvent class takes on the role of the Request. All user events that are to be processed by the state machine must be fired from the view as StateMachineEvents.

StateMachineEventSource Interface

The StateMachineEventSource interface must be implemented by all View classes in addition to the IView interface. The interface contains methods for adding and removing StateMachineEventListeners to the view.

StateMachineEventDispatcher Class

The StateMachineEventDispatcher manages the StateMachineEvents fired by the views, passing them into the state machine. StateMachineEventDispatcher also gives the resultant view to the ViewContainer for display.

ViewContainer Interface

The ViewContainer interface marks the JContainer that holds and displays the views.

Writing the Application Main Class

The application main class has the following responsibilities:

- **Creating and Displaying a ViewContainer.** The application main class must create and display some class that implements ViewContainer. This is where all the application views are displayed.
- **Creating a StateMachineEventDispatcher.** The application main class must create a StateMachineEventDispatcher. The dispatcher requires a ViewContainer and the state machine configuration properties. The main class can also set a logger, user session manager, and application manager if necessary. (In general, these can be loaded automatically based on the values in the configuration properties.)
- **Firing the First Event.** To start the application, the application main class must fire the first StateMachineEvent into the StateMachineEventDispatcher. Create a StateMachineEvent with **this**

as the target and a null event name. The state machine locates the start state for the application and gives the appropriate view to the ViewContainer.

If you complete these steps correctly, it does not matter whether the main class is an applet, frame, or neither.

Writing the ViewContainer Class

The ViewContainer class has one very simple responsibility; it must display the views that are given to it through the displayView(IView) method.

The ViewContainer is normally a JPanel or other JContainer. When it receives a view it must check that the view is a JComponent, remove or hide the previous view, and display the new one.

The view might not be a JComponent, and could be any class. When writing the ViewContainer, be aware of the types of view that are written for the application, and make sure you have a way of displaying all of them. For example: another type of view that might be supplied is a JDialog, in which case the ViewContainer should call the show() method to display the dialog.

NOTE: You should use only modal dialogs.

Writing the View Classes

As in the servlet environment, the view classes have two very simple and closely-related responsibilities. They must display information suitable to the current state, and they must present controls (for example, buttons) to the user to allow them to fire events.

The view classes must implement two interfaces, as described in the following sections.

IView Interface

The IView interface includes three methods that you must implement.

- The first method is a variant of the build method inherited from the com.bankframe.fe.statemachine.base.apps.IView interface. You can define this method with the following block of code:

  ```
  public void build(RequestContext requestContext, IState currentState) {

  com.bankframe.fe.statemachine.ext.apps.View.
  build(requestContext, currentState, this);

  }
  ```

- The second method is a variant of the build method, defined in the com.bankframe.fe.statemachine.ext.apps.IView interface. This method must be implemented to populate the view with the values displayed to the user.
- The third method is the populateFromProperties method, mentioned in Managing ViewProperties on page 106.

StateMachineEventSource Interface

You can implement the two methods in the StateMachineEventSource interface using the standard code:

```
// listenerList is an instance of javax.swing.event.EventListenerList

public void addStateMachineEventListener(StateMachineEventListener listener) {

listenerList.add(StateMachineEventListener.class, listener);

}

public void removeStateMachineEventListener(StateMachineEventListener listener) {

listenerList.remove(StateMachineEventListener.class, listener);

}
```

When you need to fire a StateMachineEvent, use the following code as a guide:

```
StateMachineEvent event = new StateMachineEvent(this, eventName);

// set the parameters as required in the event.

StateMachineEventDispatcher.fireStateMachineEvent(event, listenerList);
```

Managing ViewProperties

ViewProperties are the means through which you can include information in the statechart to be used by the View class. For example, viewProperties contains the jspName for the JSPView and the stylesheetURI for the XSLTAutoView.

You can use viewProperties by implementing the populateFromProperties method in your view to read values from the viewProperties and copy them to attributes that are later used in the build method.

Adding a View Class to the Screen Orchestrator

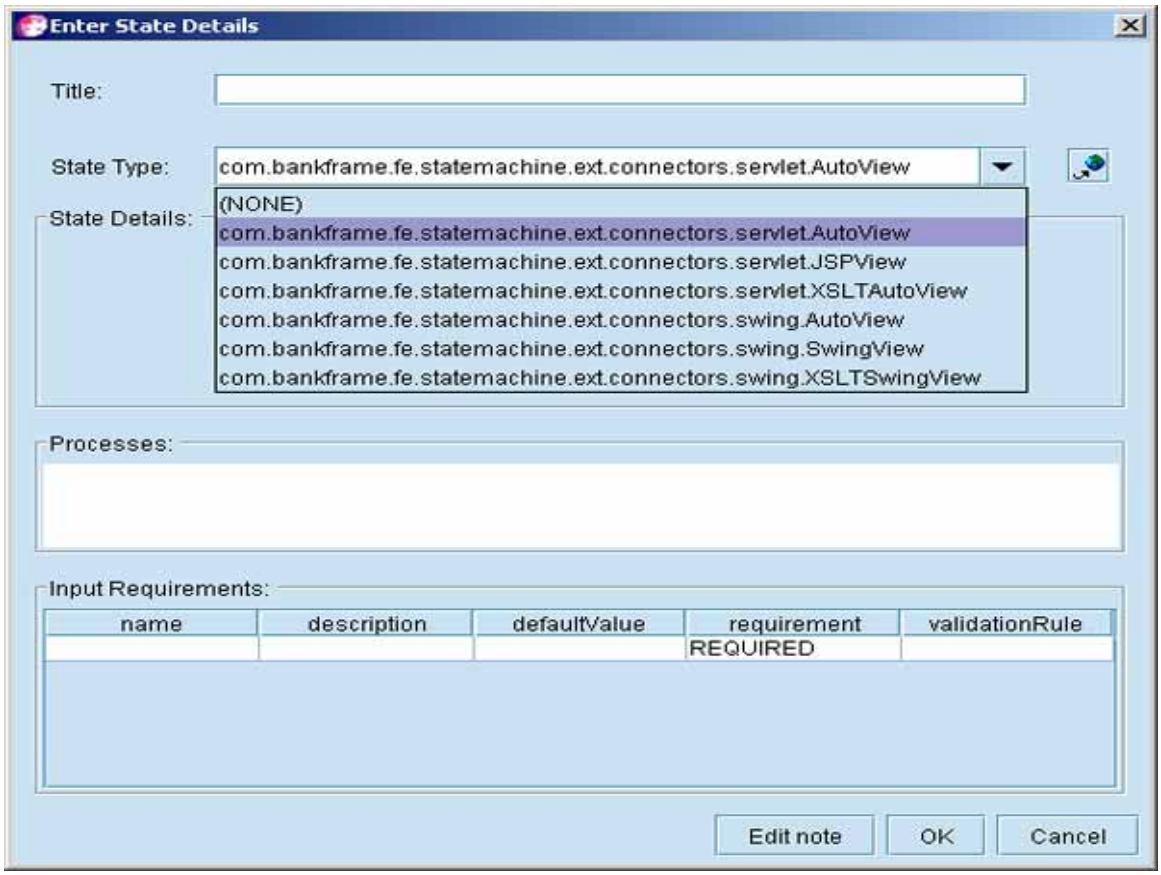
When you create a state in the Screen Orchestrator, the Enter State Details screen is displayed as shown in Figure 75.

Using this screen you specify the type of view that this state represents. The State Type field contains the fully-qualified class name for the view. By default the Screen Orchestrator provides the following types:

- None
- AutoView (JSP)
- JSPView
- XSLTAutoView

- AutoView (Swing)
- SwingView
- XSLTSwingView

Figure 75. The Enter State Details Screen



To add new Swing view classes to your statechart, you enter the fully-qualified class name into the State Type field in the Enter State Details screen, and click the Register button. This is also the case for JSP and XSLT views.

A view type class must implement the com.bankframe.fe.statemachine.ext.apps.IView interface. Additionally, you can define a BeanInfo class for your new view type. The BeanInfo class allows you to customize the view and is written according to the JavaBeans standard. When you select the new view type from the State Type field, the BeanInfo class for the view type is loaded by the Screen Orchestrator and its attributes displayed in the View Details area of the Enter State Details screen.

Examples illustrating how you can create an IView class and an associated BeanInfo class are given in the following sections.

The JSPView Class

The following code illustrates how the JSPView class was written.

```
public class JSPView extends View {

       protected String jspName;
       protected String requestURL;

    /**
     * The JSP can expect an attribute in the request with the key
     * STATE_ATTRIBUTE_NAME that contains the IState implementor for the
     * current state.
     * <br>
     * The value of STATE_ATTRIBUTE_NAME is "State"
     */
    public static String STATE_ATTRIBUTE_NAME = "State";

    /**
     * The JSP can expect an attribute in the request with the key
     * VIEW_ATTRIBUTE_NAME that contains the instance of JSPView that was
     * used.
     * You might use this to build subclasses of JSPView that perform extra
     * processing of the data in the ResponseData, exposing the results of that
     * processing through methods on the view.
     * <br>
     * The value of VIEW_ATTRIBUTE_NAME is "View"
     */
    public static String VIEW_ATTRIBUTE_NAME = "View";

    /**
     * The JSP can expect an attribute in the request with the key
     * INPUTS_ATTRIBUTE_NAME that contains the instance of Inputs that was
     * used.
     * <br>
     * You can use this in the JSP to gain access to the data from the
     * incoming request, the user session, and the response data populated
```

```
 * by the controller.
 * <br>
 * The value of INPUTS_ATTRIBUTE_NAME is "Inputs"
 */
   public static String INPUTS_ATTRIBUTE_NAME = "Inputs";

/**
 * The JSP can expect an attribute in the request with the key
 * REQUEST_CONTEXT_ATTRIBUTE_NAME that contains the current RequestContext.
 * <br>
 * The value of REQUEST_CONTEXT_ATTRIBUTE_NAME is "RequestContext"
 */
public static String REQUEST_CONTEXT_ATTRIBUTE_NAME = "RequestContext";

   public static String RESPONSE_DATA_ATTRIBUTE_NAME = "ResponseData";
   /**
    * Constructor for JSPView.
    */
   public JSPView() {
          super();
   }

   /**
   * @see com.bankframe.fe.statemachine.ext.apps.View#build(IState, Inputs,
   RequestContext)
    */
   public void build(
          IState state,
          Inputs inputs,
          RequestContext requestContext) throws StateMachineUserException {
    HttpServletRequest request = ((Request)inputs.getRequest()).getRequest();
    Response response = (Response)requestContext.getResponse();
    requestURL = request.getRequestURL().toString();
```

```
 request.setAttribute(STATE_ATTRIBUTE_NAME, state);
 request.setAttribute(VIEW_ATTRIBUTE_NAME, this);
 request.setAttribute(INPUTS_ATTRIBUTE_NAME, inputs);
 request.setAttribute(REQUEST_CONTEXT_ATTRIBUTE_NAME, requestContext);
 request.setAttribute(RESPONSE_DATA_ATTRIBUTE_NAME,
response.getResponseData());
 RequestDispatcher dispatcher = request.getRequestDispatcher(jspName);
 try {
     dispatcher.include(request, response.getResponse());
 } catch (ServletException ex) {
     throw new StateMachineUserException(ex);
 } catch (IOException ex) {
     throw new StateMachineUserException(ex);
 }
}

/**
 * Returns the jspName.
 * @return String
 */
public String getJspName() {
       return jspName;
}

/**
 * Sets the jspName.
 * @param jspName The jspName to set
 */
public void setJspName(String jspName) {
       this.jspName = jspName;
}

/**
```

```
 * Returns the jspName.
 * @return String
 * @deprecated
 */
public String getJSPName() {
	return jspName;
}

/**
 * Sets the jspName.
 * @param jspName The jspName to set
 * @deprecated
 */
public void setJSPName(String jspName) {
	this.jspName = jspName;
}

/**
 * Returns the requestURL.
 * @return String
 */
public String getRequestURL() {
	return requestURL;
}

/**
 * @see
com.bankframe.fe.statemachine.ext.apps.IView#populateFromProperties(Properties)
 */
public void populateFromProperties(Properties viewProperties) {
	if (viewProperties != null) {
		if (viewProperties.getProperty("jspName") != null) {
			setJspName(viewProperties.getProperty("jspName"));
```

```
                }
            }
        }
}
```

The JSPViewBeanInfo Class

The following code illustrates how the JSPView class sample was written.

```
public class JSPViewBeanInfo extends SimpleBeanInfo {


        protected PropertyDescriptor[] propertyDescriptors;
        protected BeanDescriptor beanDescriptor;


        /**
         * Constructor for JSPViewBeanInfo.
         */
        public JSPViewBeanInfo() throws IntrospectionException {
                super();
                PropertyDescriptor jspNameDescriptor = new PropertyDescriptor("jspName",
                JSPView.class, "getJspName", "setJspName");
                PropertyDescriptor requestURLDescriptor = new
                PropertyDescriptor("requestURL", JSPView.class, "getRequestURL", null);
                propertyDescriptors = new PropertyDescriptor[]{jspNameDescriptor,
                requestURLDescriptor};
                beanDescriptor = new BeanDescriptor(JSPView.class,
                GenericCustomizer.class);
        }


        /**
         * Returns the propertyDescriptors.
         * @return PropertyDescriptor[]
         */
        public PropertyDescriptor[] getPropertyDescriptors() {
                return propertyDescriptors;
        }
```

```
	/**
	 * Returns the beanDescriptor.
	 * @return BeanDescriptor
	 */
	public BeanDescriptor getBeanDescriptor() {
		return beanDescriptor;
	}
}
```

The JSPViewBeanInfo class allows you to enter the JSP filename for that particular state. When you select JSPView in the State Type field in the Enter State Details screen, the State Details area is customized as a result of loading the JSPViewBeanInfo class. This is illustrated in Figure 76 where welcome.jsp is entered in the jspname field.

The Swing Application Requirements

The state machine comes complete with two different connector packages, designed to allow deployment of applications within Swing or servlet environments.

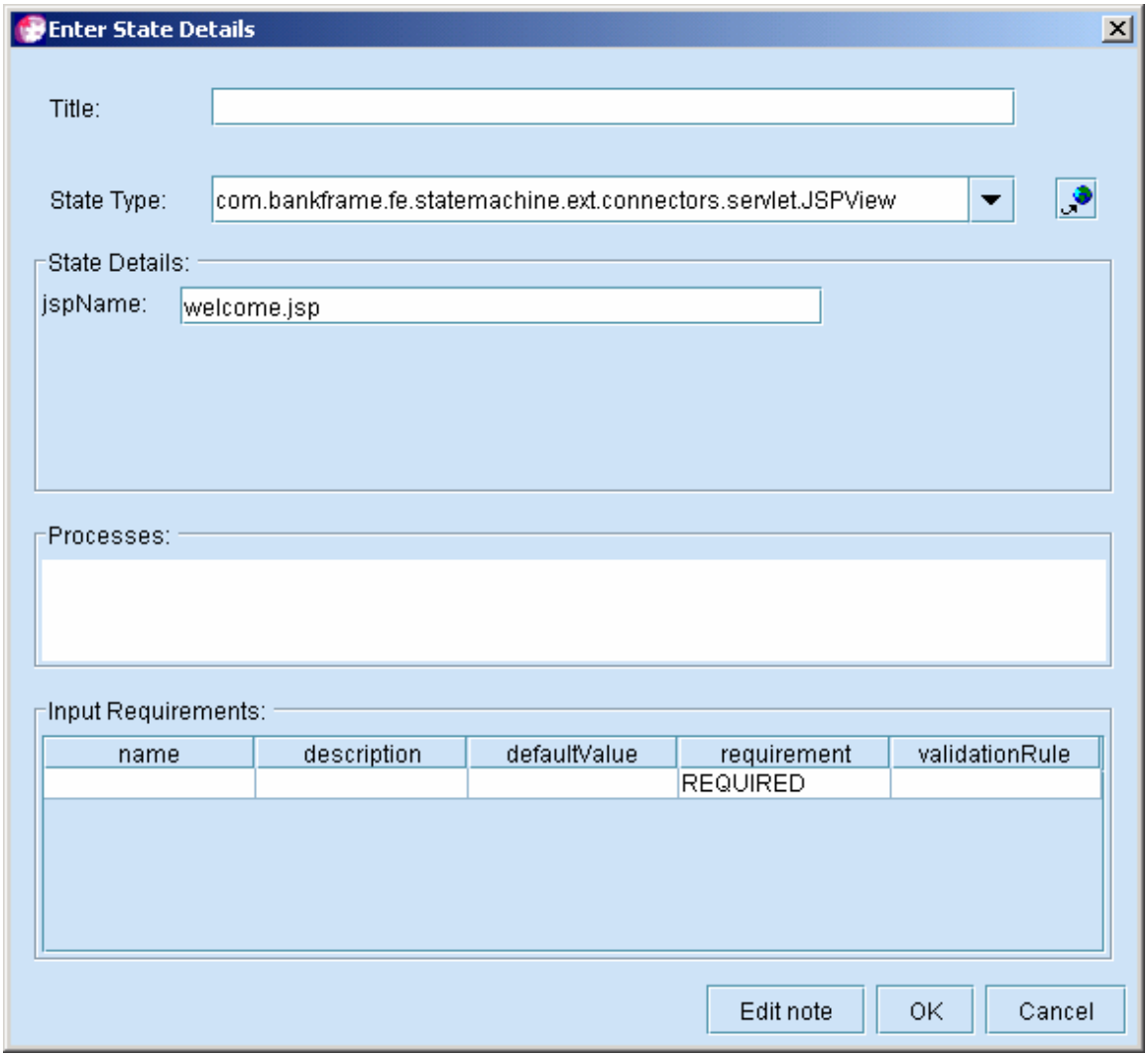
This section describes how to write a Swing application based on the state machine.

Before you set up your application you should have already created your statechart for the application and identified your views and controllers. If your application requires any special controller classes, read Chapter 6, Creating Controller Classes. On startup there are a few steps your application needs to take to use the state machine, as described in the following sections.

The ViewController Interface

Your application must designate an object to contain and display the views as they are produced. This object is probably an instance of JPanel or another JContainer, and must implement the ViewContainer

Figure 76. The Enter State Details Screen Customized with JSPViewBeanInfo



interface.

The com.bankframe.fe.statemachine.ext.connectors.swing.ViewContainer interface has one method that must be implemented and that is the displayView(IView view) method. When the state machine calls this method, the container class that implements the method must display the specified view to the user.

For more information about using this interface, see the MCA Services API documentation.

Setting the Application Properties

The Swing application then needs to set up the properties required by the UserSessionManagerFactory, ApplicationManagerFactory, ApplicationManager, and RequestContext. The default values for these classes when using running a Swing application are:

```
com.bankframe.fe.statemachine.base.UserSessionManager=
com.bankframe.fe.statemachine.ext.sessionmanagers.inmemory.UserSessionManager


com.bankframe.fe.statemachine.base.ApplicationManager=
com.bankframe.fe.statemachine.ext.apploaders.sax.ApplicationManager
```

The State Machine Events

After the application properties are set, your Swing application must create an instance of the StateMachineEvent Dispatcher class passing in the ViewContainer and Properties as parameters to the constructor of the class. Finally, the applications must create a StateMachineEvent and pass it into the StateMachineEventDispatcher. The StateMachineEventDispatcher and StateMachine Event classes are in the com.bankframe.fe.statemachine.ext.connectors.swing package. This initial event can have the ViewContainer as its target and a null event name. This causes the state machine to load the application, locate the start state, build the appropriate view, and pass it back into the ViewContainer through the displayView method.

To complete the circle and make sure all subsequent events are properly handled, there are two remaining details. The views for the application must be sources of StateMachineEvents, implementing the StateMachineEventSource interface. The view must be able to recognize those user actions that are events described on the statechart and fire StateMachineEvents appropriately. The ViewContainer must make sure that all views it displays have the StateMachineEventDispatcher registered as a StateMachineEventListener with the view. Hence, when the view fires a StateMachineEvent, the dispatcher receives it, passes it to the RequestManager, and passes the result view back to the ViewContainer.

Example of a Swing Application

The Screen Orchestrator provides a preview function for loading the currently opened statechart and stepping through the statechart. This functionality is provides by using a Swing AutoView class. This simple example of a Swing application uses the state machine. The PreviewFrame class is used in this section to provide a simple example of how a Swing application is created using the state machine.

The following is the code for a PreviewFrame class used by the Screen Orchestrator:

```
/**
```

```
 * The PreviewFrame class.
 * This class provides a swing frame for running a preview of a drawn statechart.
 * @author  Brian O'Byrne
 */
public class PreviewFrame extends JFrame implements ViewContainer {

    private StateMachineEventDispatcher eventDispatcher;
    private JScrollPane scrollPane;
    private JPanel viewportComponent;

       /**
        * The PreviewFrame constructor.
       * @param appDoc Document is the XML document representation of the statechart
       to be previewed.
        */
    public PreviewFrame(Document appDoc) {
       this(appDoc, "State Chart Editor Preview");
    }

       /**
        * The PreviewFrame constructor.
        * @param app Application is the statechart application to be previewed.
        */
    public PreviewFrame(Application app) {
       this(app, "State Chart Editor Preview");
    }

       /**
        * The PreviewFrame constructor.
        * @param appDoc Document the statechart xml.
        * @param title String
        */
    public PreviewFrame(Document appDoc, String title) {
```

```
       super(title);
        initComponents();
        Properties applicationProperties = new Properties(System.getProperties());

applicationProperties.setProperty("com.bankframe.fe.statemachine.base.ApplicationManag
er", "com.eontec.statechart.preview.ApplicationManager");

applicationProperties.setProperty("com.bankframe.fe.statemachine.base.UserSessionManag
er", "com.bankframe.fe.statemachine.ext.sessionmanagers.inmemory.UserSessionManager");
        applicationProperties.setProperty(RequestManager.VIEW_OVERRIDE_KEY,
"com.bankframe.fe.statemachine.ext.connectors.swing.AutoView");
        try {
            eventDispatcher = new StateMachineEventDispatcher(this,
applicationProperties);
            ApplicationManager appManager =
(ApplicationManager)eventDispatcher.getApplicationManager();
            appManager.loadApplication(appDoc);
            appManager.getDefaultApplication();
             eventDispatcher.handleStatemachineEvent(new StateMachineEvent(this,
       null));
        } catch (StateMachineException ex) {
            ex.printStackTrace();
        }
    }

       /**
        * The PreviewFrame constructor.
        * @param app Application the statechart.
        * @param title String
        */
    public PreviewFrame(Application app, String title) {
       super(title);
       initComponents();
       Properties applicationProperties = new Properties(System.getProperties());
```

```
applicationProperties.setProperty("com.bankframe.fe.statemachine.base.ApplicationManag
er", "com.eontec.statechart.preview.ApplicationManager");


applicationProperties.setProperty("com.bankframe.fe.statemachine.base.UserSessionManag
er", "com.bankframe.fe.statemachine.ext.sessionmanagers.inmemory.UserSessionManager");

        applicationProperties.setProperty(RequestManager.VIEW_OVERRIDE_KEY,
"com.bankframe.fe.statemachine.ext.connectors.swing.AutoView");

        try {

            eventDispatcher = new StateMachineEventDispatcher(this,
applicationProperties);

            ApplicationManager appManager =
(ApplicationManager)eventDispatcher.getApplicationManager();

            appManager.loadApplication(app);

            appManager.getDefaultApplication();

            eventDispatcher.handleStatemachineEvent(new StateMachineEvent(this,
null));

        } catch (StateMachineException ex) {

            ex.printStackTrace();

        }

    }


       /**
        * This method initializes the frame.
        */

    private void initComponents() {

        addWindowListener(new java.awt.event.WindowAdapter() {

            public void windowClosing(java.awt.event.WindowEvent evt) {

                exitForm(evt);

            }

        });

        this.getContentPane().setLayout(new BorderLayout());

        scrollPane = new JScrollPane();

        JButton closeButton = new JButton();

        closeButton.setActionCommand("CLOSE_BUTTON_CMD");
```

```
    closeButton.setText("Close Preview");
    closeButton.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent actionEvent) {
                exitForm(actionEvent);
          }
    });
    viewportComponent = new JPanel();
    viewportComponent.setLayout(new BorderLayout());
    JPanel viewportComponentFiller = new JPanel();
    viewportComponent.add(viewportComponentFiller, BorderLayout.CENTER, 0);
    viewportComponent.add(new JPanel(), BorderLayout.NORTH, 1);
    scrollPane.setViewportView(viewportComponent);
    scrollPane.setBackground(null);
    this.getContentPane().add(scrollPane, BorderLayout.CENTER);
    this.getContentPane().add(closeButton, BorderLayout.SOUTH);

   this.setIconImage(ImageLoader.getImageIcon("STATE_MACHINE_ICON").getImage());
    this.setSize(new Dimension(550,600));
}

/**
 * This method hides the preview frame.
 */
private void exitForm(java.awt.event.WindowEvent evt) {
    this.hide();
}

/**
 * This method hides the preview frame.
 */
private void exitForm(ActionEvent evt) {
   this.hide();
}
```

```
    /**
     * This method will display the specified view in the preview frame.
     * @see
com.bankframe.fe.statemachine.ext.connectors.swing.ViewContainer#displayView(IView)
     */
    public void displayView(IView view) {
        ((StateMachineEventSource)view).addStateMachineEventListener(eventDispatcher);
        viewportComponent.remove(1);
        viewportComponent.add((Component)view, BorderLayout.NORTH, 1);
        validate();
        repaint();
    }

    /**
     * This method adds a StateMachineProcessingListener to the statemachine event
    dispatcher.
     * @param listener StateMachineProcessingListener
     */
    public void addStateMachineProcessingListener(StateMachineProcessingListener
    listener) {
        this.eventDispatcher.addStateMachineProcessingListener(listener);
    }

    /**
     * This method removes a StateMachineProcessingListener to the statemachine
    event dispatcher.
     * @param listener StateMachineProcessingListener
     */
    public void removeStateMachineProcessingListener(StateMachineProcessingListener
    listener) {
        this.eventDispatcher.removeStateMachineProcessingListener(listener);
    }
```

```
    /**
     * This method adds a collection of StateMachineProcessingListeners to the
    statemachine event dispatcher.
     * @param listeners Collection
     */
  public void addStateMachineProcessingListener(Collection listeners) {
    this.eventDispatcher.addStateMachineProcessingListener(listeners);
  }

    /**
     * This method removes a collection of StateMachineProcessingListener to the
    statemachine event dispatcher.
     * @param listeners Collection
     */
  public void removeStateMachineProcessingListener(Collection listeners) {
    this.eventDispatcher.removeStateMachineProcessingListener(listeners);
  }

}
```

The important things to look at in the code example are the constructors for the class. They create the application properties for the state machine and set them specifically for this application. The preview frame provides its own ApplicationManager; this class is used to load the specified statechart application or XML document in this instance.

```
applicationProperties.setProperty("com.bankframe.fe.statemachine.base.ApplicationManag
er", "com.eontec.statechart.preview.ApplicationManager");

applicationProperties.setProperty("com.bankframe.fe.statemachine.base.UserSessionManag
er", "com.bankframe.fe.statemachine.ext.sessionmanagers.inmemory.UserSessionManager");

      applicationProperties.setProperty(RequestManager.VIEW_OVERRIDE_KEY,
"com.bankframe.fe.statemachine.ext.connectors.swing.AutoView");
```

The code also sets a view override, which informs the state machine that all views specified in the statechart must be ignored and only the Swing AutoView class must be used as views for the preview frame. This override is not required in your Swing application, as you want your application to load the views that you specify.

Next, the constructor creates a StateMachineEventDispatcher using the previously highlighted properties.

```
eventDispatcher = new StateMachineEventDispatcher(this, applicationProperties);
```

The PreviewFrame class implements the ViewController interface and can therefore be used in the constructor of the StateMachineEventDispatcher class.

The displayView method, which is required to be implemented by the PreviewFrame, has the responsibility of displaying the next view and registering the event dispatcher with the view.

```
public void displayView(IView view) {

((StateMachineEventSource)view).addStateMachineEventListener(eventDispatcher);

viewportComponent.remove(1);

viewportComponent.add((Component)view, BorderLayout.NORTH, 1);

validate();

repaint();

}
```

The AutoView instances, which are IView interfaces, fire events to the state machine using the registered event dispatcher.

13 Validating Input Requirements

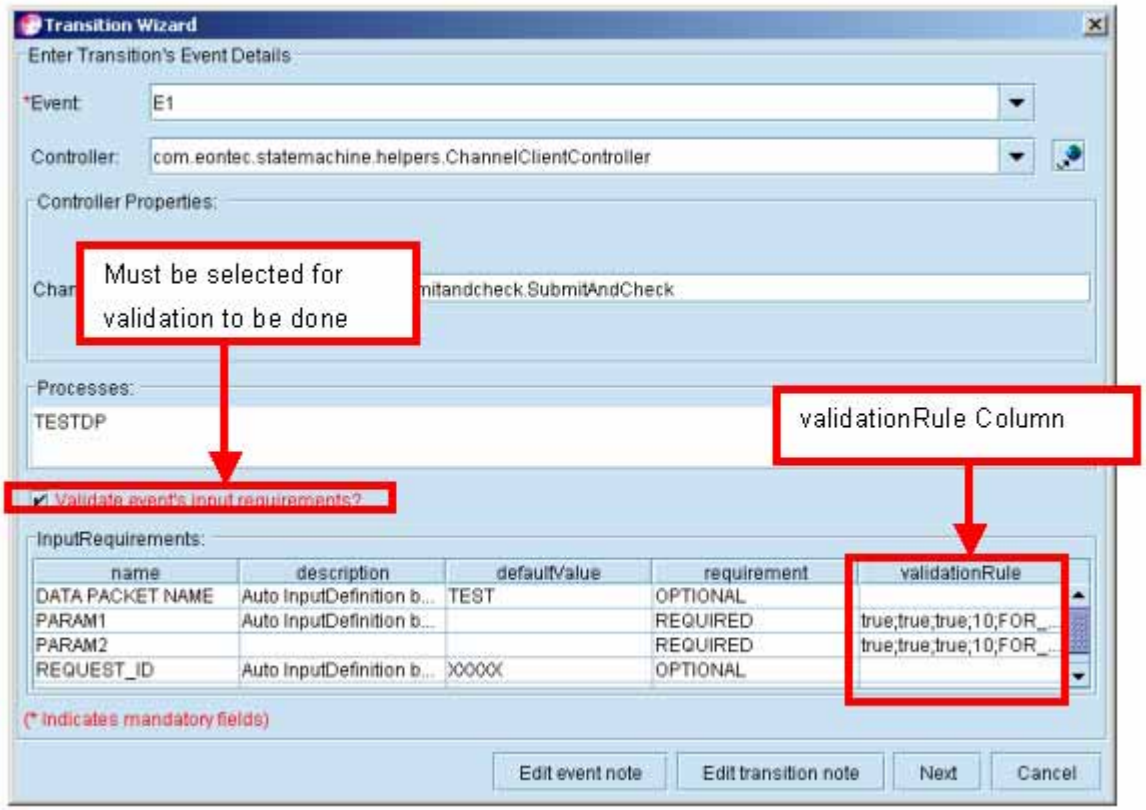
You can use the Screen Orchestrator to define validation rules for various input requirements so that the state machine can execute these rules before any event is handled. This capability is a very useful feature for Web-based applications where no validation can be done on the actual JSP (for example no dynamic scripting is allowed on the page), and the form submitted to the state machine must be validated before any processing is done.

This chapter includes the following topics:

Defining the Validation Rules

You specify validation rules for input requirements in the Transition Wizard, as shown in Figure 77.

Figure 77. Validation Rules in the Transition Wizard



To specify a validation rule for an input requirement

1. Select the Validate event's input requirements? check box.
2. Double-click in the validationRule cell for the input requirement.

   The Specify validation rule for input requirement screen is displayed.
3. Complete the Validation Rule details. The fields are described in the following table.

Field	Description
Mandatory	Select this check box if the input field is a required field.
Exact Length	Select this check box f the input field must be of a required length.
Maximum Length	Type the maximum length for the input.
Rule	Select the validation rule to be applied.
Key/Pattern	Select if the field must contain a specific value or be of a particular pattern.
Name of value	Displays the parameter name.

4. Click OK.

How the State Machine Handles the Validation Check

After you have used the Screen Orchestrator to define the validation rules, you can use the state machine to run the actual application. When an event is submitted to the state machine, the first task of the state machine is to determine whether validation of the event's inputs is required before the event is processed and its transition followed.

If validation is required, the state machine reads the rules for each input requirement and then validates each input based on the specified rule. Each input is tested in turn and a record is built up of all the inputs that fail validation. If no input fails validation, the state machine proceeds as normal. However, if any of the inputs fail validation, the record of failed inputs and their validation exceptions are added to the request as a collection of FAILED_VALIDATION_ERRORS. The state machine then returns the user to the last displayed state. The view for that state can then display the failed validation rules to the user. Figure 78 and Figure 79 show the Screen Orchestrator's preview frame running a test application and failing input validations.

Figure 78. The Screen Orchestrator Preview Frame for Testing an Application

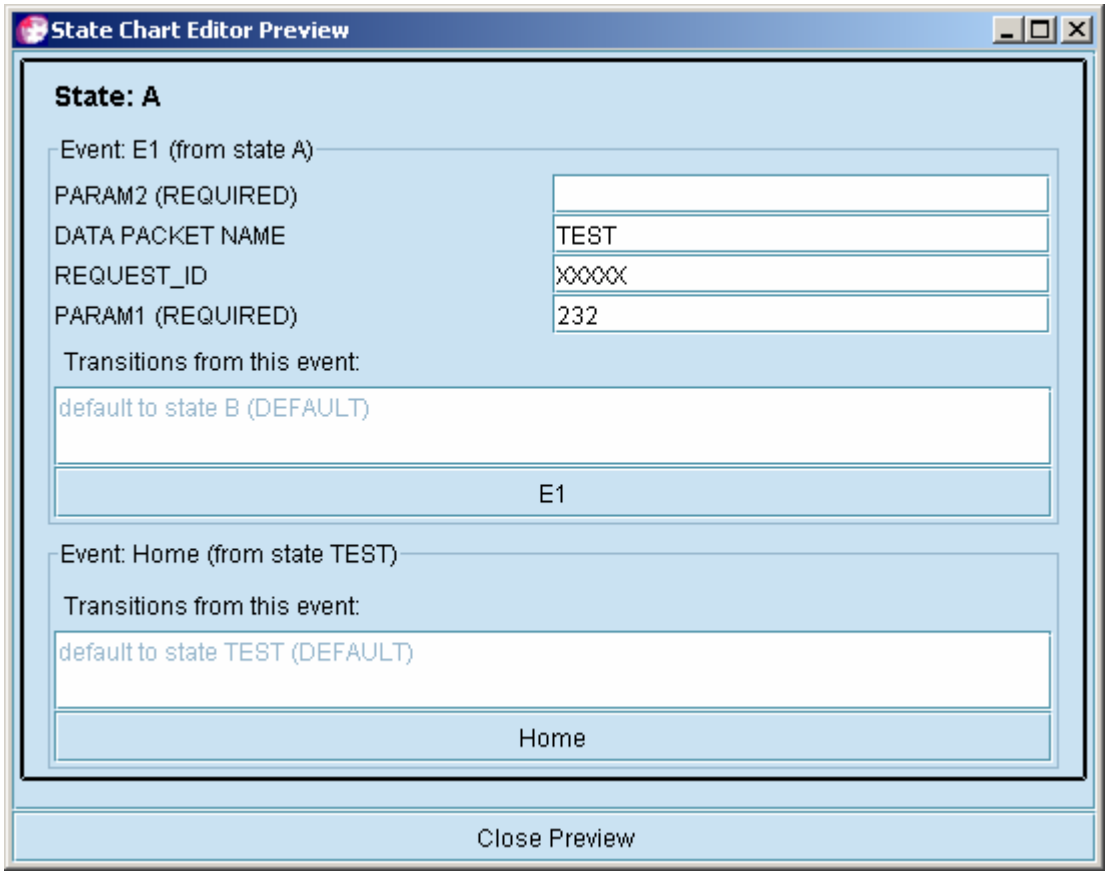
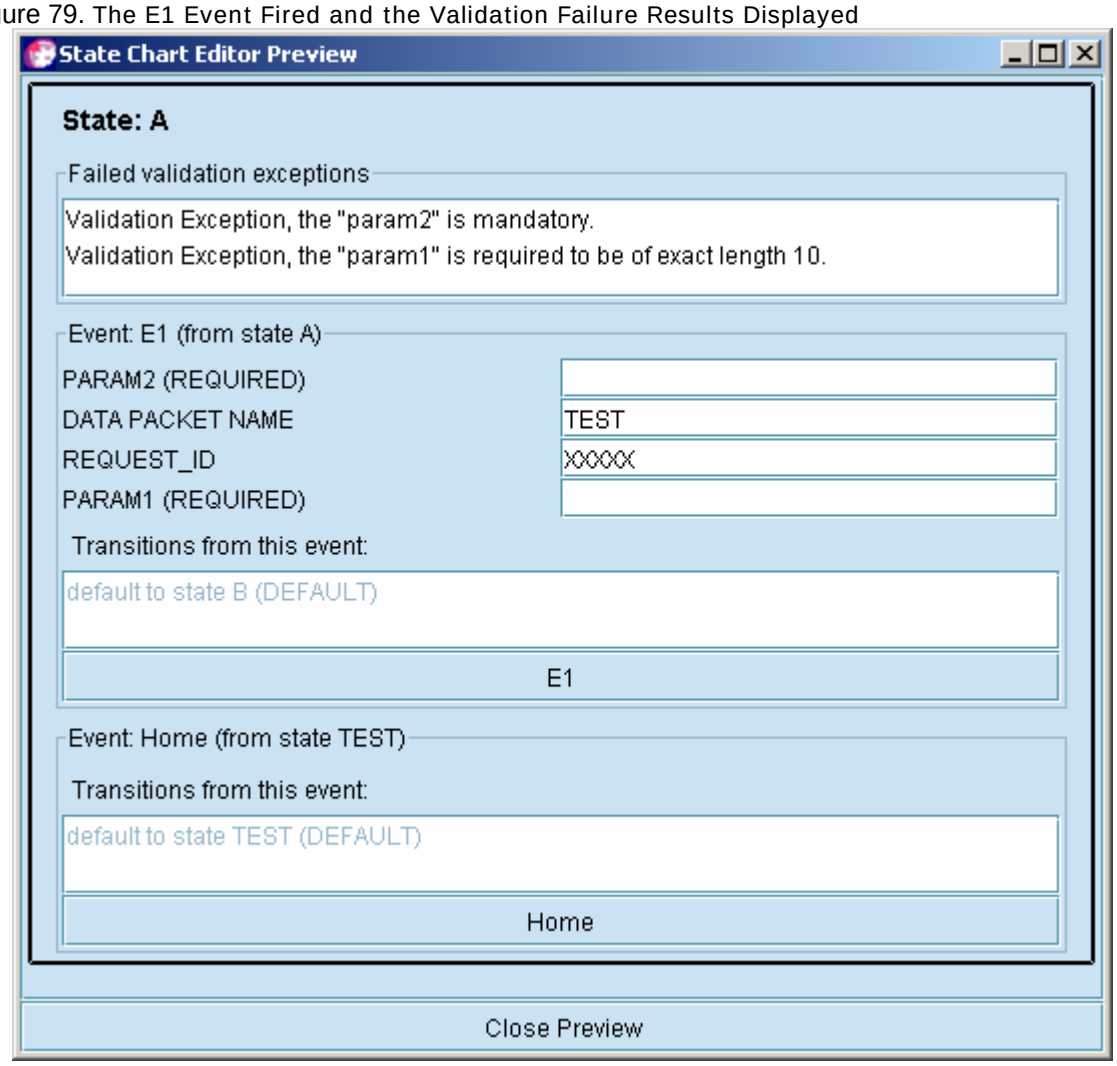


Figure 79. The E1 Event Fired and the Validation Failure Results Displayed



14 Generating JSPs and Swing Panels

This chapter describes how to generate the Java Server Pages (JSP) and Swing panels associated with autoviews in the Screen Orchestrator. It contains the following topics:

About Generating JSPs and Swing Panels

The state machine has a concept of autoviews. If no view exists for a particular state, the state machine can supply an autoview for that state at run time.

Using the Screen Orchestrator, you can generate the actual JSPs or Swing panels that the autoview would create. The files are very useful for providing initial starting points for view states for developing the application. You can draw your statechart and generate a starting set of JSPs or Swing panels from which you can test and develop the initial application. You can then edit the JSPs or Swing panels as required.

When you generate the JSPs and Swing panels, XSLT style sheets are used, and you can modify these style sheets to change the look and feel of the files that are generated. The JSP style sheet is the *orchestrator-install-dir*\resources\jspTemplate.xsl file, while the Swing panel style sheet is the *orchestrator-install-dir*\resources\panelTemplate.xsl file.

When you generate JSPs, the filename for each state is the jspName specified in the JSPView state. If the state is an autoview, the JSP filename is the state name postfixed with a .jsp extension.

When you generate Swing panels, the filename is based on the fully-qualified classname specified for each SwingView's state panelName. If the state is an autoview, the panel name is based on the state name, and the package name defaults to temp.

Testing Whether JSPs Can Be Compiled

To test whether your JSPs can be compiled, click the JSP button on the Screen Orchestrator toolbar.

This does not generate any JSPs, it only tests whether they can be compiled.

Generating JSP and Swing Panel Files

You can generate all the JSPs and Panels for the currently opened statechart.

To generate JSP and Swing panels

1. Click the generator button (the cog wheel icon) on the Screen Orchestrator toolbar.

The Confirm file generation? screen is displayed.

2 Complete the file generation details as described in the following table.

Field	Description
Generate files to following directory:	If required, click Change and specify a directory if you want to generate the files to a directory other that the default directory.
Use chart's specified HTML directory	Select this check box to generate the files to the chart's specified HTML directory.
JSPs	Select the check box to generate the JSPs.
Panels	Select the check box to generate the Swing panels.

3 Click OK.

4 If a file that that you specified already exists, confirm whether you want to overwrite that particular file and all files in the folder.

15 MCA Services Timing Points

This chapter describes how you can use timing points at various points in the state machine for performance testing purposes. It contains the following topics:

About Timing Points

A timing point is code that is used to time events or actions within Siebel Retail Finance code. In the state machine, timing points are used to measure the time taken to handle requests and perform actions associated with transitions. The data recorded by timing points is logged to file or disk.

For more information about timing points, see the *MCA Services Developer Guide* and the MCA Services API documentation.

Timing Points in the State Machine

To record the overall time for a state machine request, use the following timing points:

- In the `com.bankframe.fe.statemachine.ext.connectors.servlet.EntryServlet` class for JSP/HTML applications, the `public void delegateToRequestManager(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException;` method records the overall time to handle an HTML request to the state machine
- In the `com.bankframe.fe.statemachine.ext.connectors.swing.StateMachineEventDispatcher` class for a Swing application, the `public void handleStatemachineEvent(StateMachineEvent evt)` method records the overall time to handle a Swing request to the state machine

There are also additional timing points in the com.bankframe.fe.statemachine.ext.apps.Controller class:

- public com.bankframe.fe.statemachine.base.apploaders.IStateTransition getResult(RequestContext requestContext, om.bankframe.fe.statemachine.base.apploaders.IEvent event) throws StateMachineUserException;
- public void doSideEffects(RequestContext requestContext, com.bankframe.fe.statemachine.base.apploaders.IStateTransition transition) throws StateMachineUserException;

These timing points determine how long it takes for the state machine to get the correct transition to follow and to do the actions for that chosen transition.

If any of the methods are overwritten, you should also put the timing code into the overwritten methods.

Timing points are added in the code as follows:

```
Object[] objects = new Object[]{TimingPointConstants.TIMING_POINT_SUBSYSTEM,
"NAMEOFSUBSYSTEM", TimingPointConstants.TIMING_POINT_TYPE,
"",TimingPointConstants.TIMING_POINT_MAJOR_TYPE,
TimingPointUtil.MAJORTYPE_SERVLET_STRING, TimingPointConstants.TIMING_POINT_REQUEST,
"REQUESTVALUE"};

TimingPoint tp=TimingPointFactory.getTimingPoint(new TimingPointProperties(objects));
```

The following code is used to exit from the timing point:

```
tp.exit(this);
```

The descriptions used in the timing point log for the state machine classes are:

- **EntryServlet.** request round trip time.
- **StateMachineEventDispatcher.** request round trip time.
- **Controller getResult method.** round trip time.
- **Controller doSideEffects method.** round trip time.

The MCA 1.2 and later versions have timing code for measuring the time for executing MCA requests. To determine the correct state machine round trip time, subtract this time from the state machine times, if requests are being sent to the EJB server by the state machine controller classes