360Commerce®

360Store®

Point-of-Sale Developer Guide

Release 7.2.0 LA February 2006

11400 Burnet Road, Suite 5200, Austin, Texas 78758
(512) 491-2600 • (800) 897-8663 • (512) 491-0078 (fax)
www.360Commerce.com

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF CODE SAMPLES

PREFACE

Audience

The audience for this document is developers who develop code for 360Store Point-of-Sale. Knowledge of the following techniques is required:

- Java Programming Language
- Object-Oriented Design Methodology (OOD)
- Extensible Markup Language (XML)

Goals

A developer who reads this document will be able to:

- Understand existing Point-of-Sale code
- Create new Point-of-Sale code
- Extend existing Point-of-Sale code

Feedback

Please e-mail feedback about this document to 360University@360Commerce.com.

Trademarks

The following trademarks may be found in 360Commerce® documentation:

- 360Commerce, 360Store and 360Enterprise are registered trademarks of 360Commerce Inc.
- Unleashed is a trademark of 360Commerce Inc.
- BEETLE is a registered trademark of Wincor Nixdorf International GmbH.
- Dell is a trademark of Dell Computer Corporation.
- IBM, WebSphere and SurePOS are registered trademarks or trademarks of International Business Machines Corporation in the United States, other countries, or both.

Text Conventions

The following table shows the text conventions used in this document:

Table P-1 Conventions

Sample	**Description**
`Courier Text`	Filenames, paths, syntax, and code
Bold text	Emphasis
<*Italics and angle brackets*>	Text in commands which should be supplied by the user

CHAPTER 1

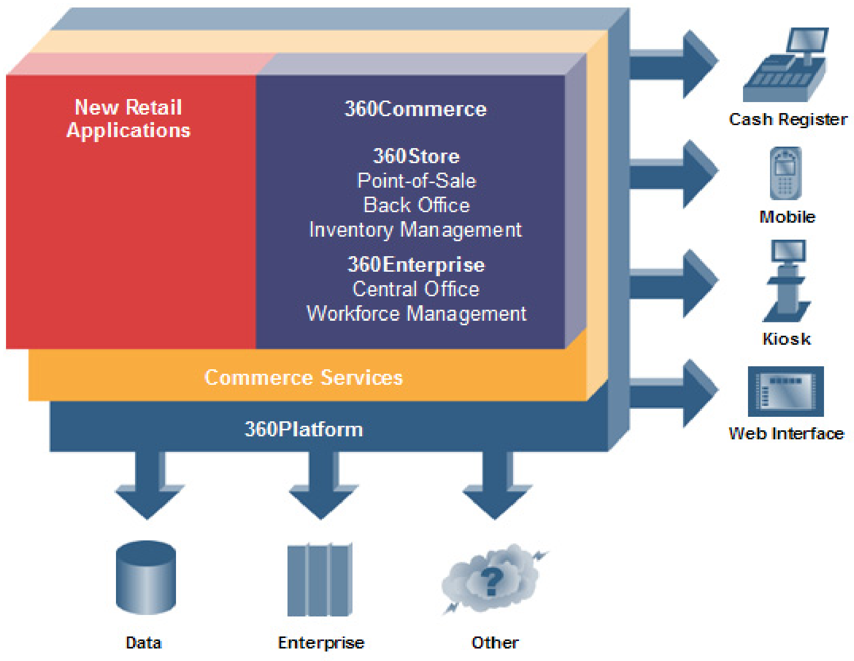
ARCHITECTURE

This chapter contains information about the 360Store® Point-of-Sale architecture. It begins with a general overview of the 360Commerce architecture. Then it describes the layers of the Point-of-Sale architecture, its frameworks, and design patterns.

Overview

Retailers have an increasing demand for enterprise information and customer service capabilities at a variety of points of service, including the Internet, kiosks and handheld devices. The retail environment requires that new and existing applications can be changed quickly in order to support rapidly changing business requirements. 360Platform and Commerce Services enable application developers to quickly build modifiable, scalable, and flexible applications to collect and deliver enterprise information to all points of service. Figure 1-1 shows a high level view of the 360Commerce architecture and components.

Figure 1-1 360Platform Architecture



The following table describes the components in the diagram:

Table 1-1 360Commerce Architecture Components

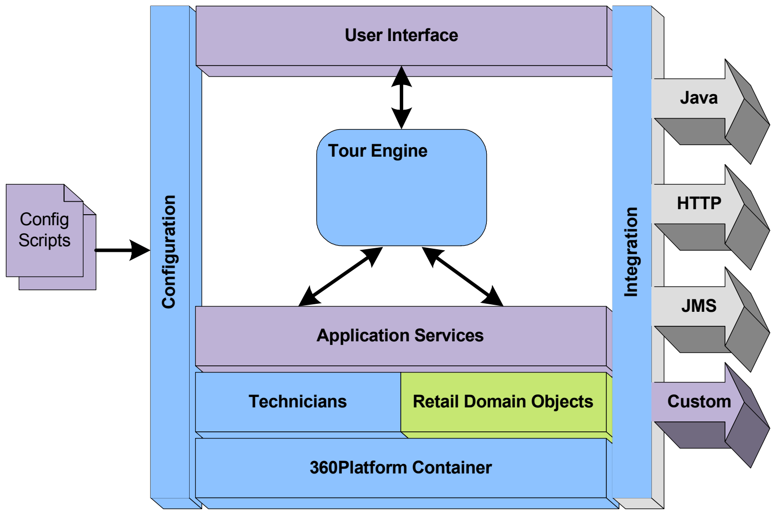
Component	Description
360Platform	360Platform provides services to all 360Commerce applications. It contains the tour framework, UI framework, and Manager/Technician frameworks. 360Platform is not retail-specific.
Commerce Services	Commerce Services implement business logic. Commerce Services define data and behavior for retail applications. This component is referred to as Retail Domain in Point-of-Sale.
360Commerce Applications	All 360Commerce applications leverage the frameworks and services provided by 360Platform and Commerce Services.
External Interfaces	Using frameworks and services, the applications are able to interface to other applications and resources.

Advantages of the 360Commerce architecture include its object-oriented design and scalability. The system is designed to support existing systems and customer extensions. 360Platform frameworks support integration by adhering to retail and technology standards. The multi-tier design of the architecture allows the application to support numerous types of infrastructure.

Point-of-Sale Architecture

360Platform contains reusable, highly customizable components for building and integrating retail applications with user interfaces, devices, databases, legacy systems, and third-party applications. 360Platform also contains integration points for communicating with external resources. The following diagram shows how the Tour engine controls the Point-of-Sale system. This diagram is a more detailed view of the components that form the Commerce Services and 360Platform tiers in the previous diagram.

Figure 1-2 Point-of-Sale Architecture Layers



Beginning with configuration of the UI and Managers/Technicians, events at the user interface are handled by the tour engine, which interacts with tour code (Application Services) and Managers/ Technicians (foundation services) as necessary, capturing and modifying the data stored in Retail Domain objects. Any communication with an integration point is handled by the 360Platform container.

The following table describes the layers of the Point-of-Sale architecture:

Table 1-2 Point-of-Sale Architecture Layers

Component	**Description**
Configuration	Application and system XML scripts configure the layers of the application.
User Interface	This layer provides client presentation and device interaction.
Tour Engine	This mechanism handles the workflow in the application. The tour engine is the controller for Point-of-Sale.
Application Services	This layer provides application-specific business processes. A tour is an application service for Point-of-Sale.
Foundation Services	This layer provides stateless, application-independent technical services. Combined with the Retail Domain objects, it forms the Commerce Services layer. Technicians provide location-transparent services in Point-of-Sale.
Retail Domain Objects	Pure retail-specific business objects that contain application data.
360Platform Container	This is an execution platform and application environment. The Tier Loader is the 360Platform container for Point-of-Sale.
Integration	This layer provides an integration framework for building standard and custom interfaces using standard integration protocols.

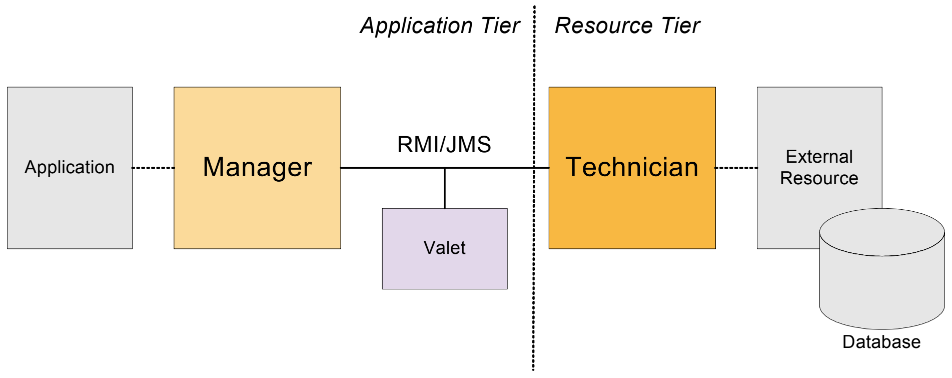
Frameworks

The 360Commerce architecture uses a combination of technologies that make it flexible and extensible, and allow it to communicate with other hardware and software systems. The frameworks that drive the application are implemented by the Java programming language, distributed objects, and XML scripting. Described below, the User Interface, Business Object, Manager/Technician, Data Persistence, and Navigation frameworks interact to provide a powerful, flexible application framework.

Manager/Technician

The Manager/Technician framework is the component of 360Platform that implements the distribution of data across a network. A Manager provides an API for the application and communicates with its Technician, which implements the interface to the external resource. The Manager is always on the same tier, or machine, as the application, while the Technician is usually on the same tier as the external resource. The following figure shows an example of the Manager/Technician framework distributed on two different tiers.

Figure 1-3 Manager/Technician Framework



The following table describes the components:.

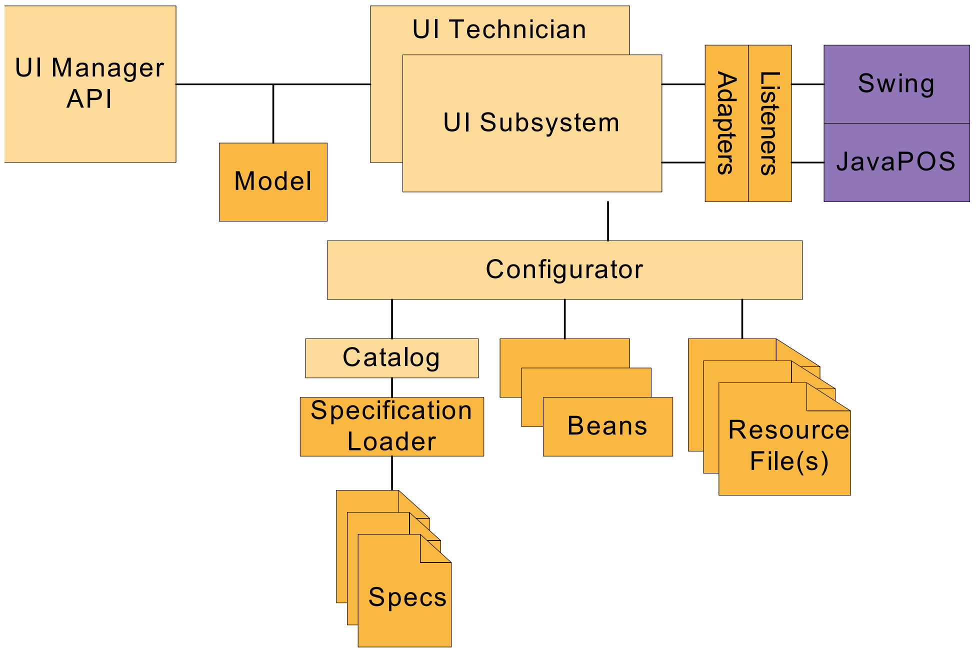
Table 1-3 Manager/Technician Framework Components

Component	Description
Manager	Managers provide a set of local calls to the application. There are various types of managers to handle various types of activity. For example, the Data Manager receives the request to save data from Point-of-Sale. It locates the appropriate Technician that should perform the work and insulates the application from the process of getting the work accomplished. The Manager is available only on the local tier.
Valet	The valet is the object that receives the instructions from the Manager and delivers them to the Technician. The valet handles data transfer across machines with RMI or JMS.
Technician	The Technician is responsible for communicating with the external resource. When a Technician receives a valet, it can handle it immediately or queue it for later action. The Technician can be remote from the Manager or on the local tier.

User Interface

The UI framework includes all the classes and interfaces in 360Platform to support the rapid development of UI screens. In the application code, the developer creates a model that is handled by the UI Manager in the application code. The UI Manager communicates with the UI Technician, which accesses the UI Subsystem. The following figure illustrates components of the UI framework.

Figure 1-4 UI Framework



The components of the UI framework are described in the following table.

Table 1-4 UI Framework Components

Component	**Description**
Resource File(s)	Resource files are text bundles that provide the labels for a screen. They are implemented as properties files. Text bundles are used for localizing the application.
Bean	Beans are reusable Java program building blocks that can be combined with other components to form an application. They typically provide the screen components and data for the workpanel area of the screen.
Specs	Specifications define the components of a screen. Display specifications define the width, height, and title of a window. Template specifications divide displays into areas. Bean specifications define classes and configurators and additional screen elements for a component. Default screen specifications map beans to the commonly used areas and define listeners to the beans. Overlay screen specifications define additional mappings of beans and listeners to default screens.
Specification Loader	Loaders find external specifications and interpret them. The loader instantiates screen specifications such as overlays, templates, and displays, and places the objects into a spec catalog.
Catalog	A Catalog provides the bean specifications by name. The UI Technician requests the catalog from the loader to simplify configurations.
Configurator	The UI framework interfaces with beans through bean configurator classes, which control interactions with beans. A configurator is instantiated for each bean specification. They apply properties from the specifications to the bean, configure a bean when initialized, reset the text on a bean when the locale changes, set the bean component data from a model, update a model from the bean component data, and set the filename of the resource bundle.

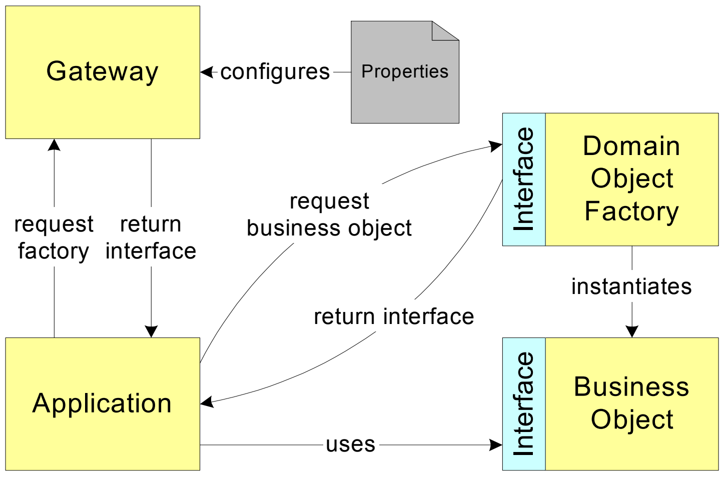
Table 1-4 UI Framework Components

Component	Description
Model	The business logic communicates with beans through screen models. Each bean configurator contains a screen model, and the configurator must determine if any action is to be taken on the model. Classes exist for each model.
UI Manager	The UI Manager provides the API for application code to access and manipulate user interface components. The UI Manager uses different methods to call the UI Technician.
UI Technician	The UI Technician controls the main application window or display. The UI Technician receives calls from Point-of-Sale tours, locates the appropriate screen, and handles the setup of the screens through the UI Subsystem.
UI Subsystem	The UI Subsystem provides UI components for displaying and editing Point-of-Sale screens. The UI subsystem enables application logic to be completely isolated from the UI components. This component is specific to the technology used, such as Swing or JSP.
Adapters	Adapters are used to provide a specialized response to bean events. Adapters can handle the events, or the event can cause the adapter to manipulate a target bean. Adapters implement listener interfaces to handle events on the UI. Adapters come from the Swing API of controls and support JavaPOS-compliant devices.
Listeners	Listeners provide a mechanism for reacting to user interface events. Listeners come from the Swing API of controls and support JavaPOS-compliant devices.

Business Object

The Commerce Services layer of the architecture contains the Business Object framework that implements the instantiation of business objects. The Business Object framework is used to create new business objects for use by Point-of-Sale. The business objects contain data and logic that determine the path or option used by an application.

Figure 1-5 Business Object Framework



The components in the Business Object framework are described in the following table.

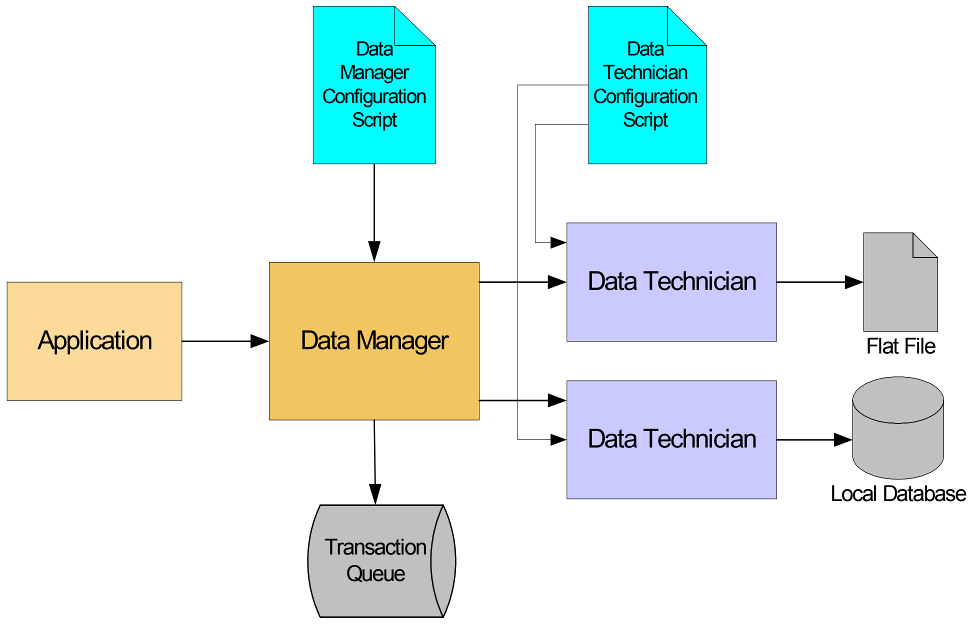
Table 1-5 Business Object Framework Components

Component	Description
DomainGateway	The DomainGateway class provides a common access point for all business object classes. It also configures dates, times, decimals, percentages, currency, and numbers.
Domain Object Factory	The Domain Object Factory returns instances of business object classes. The application requests a Factory from the DomainGateway.
Business Object	Business objects define the attributes for application data. New instances are created using the Domain Object Factory.

Data Persistence

A specific Manager/Technician pair is the Data Manager and Data Technician used for data persistence. The Data Persistence framework illustrates how data gets saved to a persistent resource, such as the database or flat files on the register.

Figure 1-6 Data Persistence Framework



The components in the Data Persistence framework are described in the following table.

Table 1-6 Data Persistence Framework Components

Component	Description
Data Manager	The Data Manager defines the application entry point into the Data Persistence Framework. Its primary responsibility is to contact the Data Technician and transport any requests to the Data Technician.
Data Manager Configuration Script	The Data Manager processes data actions from the application based on the configuration information set in the Data Manager Configuration Script. The Configuration Script defines transactions available to the application.

Table 1-6 Data Persistence Framework Components

Component	Description
Data Technician	The Data Technician provides the interface to the database or flat file. This class is part of the 360Platform framework. It provides entry points for application transactions sent by the Data Manager and caches the set of supported data store operations. It also contains a pool of physical data connections used by the supported data operations.
Data Technician Configuration Script	The Data Technician Configuration Script specifies the types of connections to be pooled, the set of operations available to the application, and the mapping of an application data action to a specific data operation.
Transaction Queue	The Transaction Queue holds data transactions and offers asynchronous data persistence and offline processing for Point-of-Sale. When the database is offline, the data is held in the queue and posted to the database when it comes back online. When the application is online, the Data Manager gets the information from the Transaction Queue to send to the database.

Tour

The Tour framework establishes the workflow for the application. It models application behavior as states, events and transitions. The 360Platform engine is modeled on finite state machine behavior. A finite state machine has a limited number of possible states. A state machine stores the status of something at a given time and, based on input, changes the status or causes an action or output to occur. The Tour framework provides a formal method for defining these nested state machines as a traceable way to handle flow through an application.

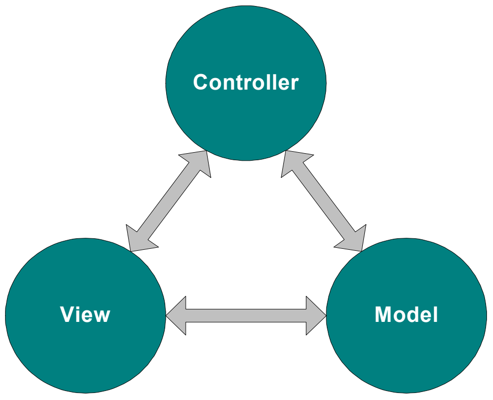
Design Patterns

Design patterns describe solutions to problems that occur repeatedly in object-oriented software development. A pattern is a repeatable, documented method that can be applied to a particular problem. This section describes four patterns used in the architecture of Point-of-Sale: MVC, Factory, Command, and Singleton.

MVC Pattern

The MVC Pattern divides the functionality of an application into three layers: model, view, and controller. Different functionality is separated to manage the design of the application. A model represents business objects and the rules of how they are accessed and updated. The model informs views when data changes and contains methods for the views to determine its current state. A view displays the contents of a model to the user. It is responsible for how the data is presented. Views also forward user actions to the controller. A controller directs the actions within the application. The controller is responsible for interpreting user input and triggering the appropriate model actions. The following diagram illustrates the MVC Pattern.

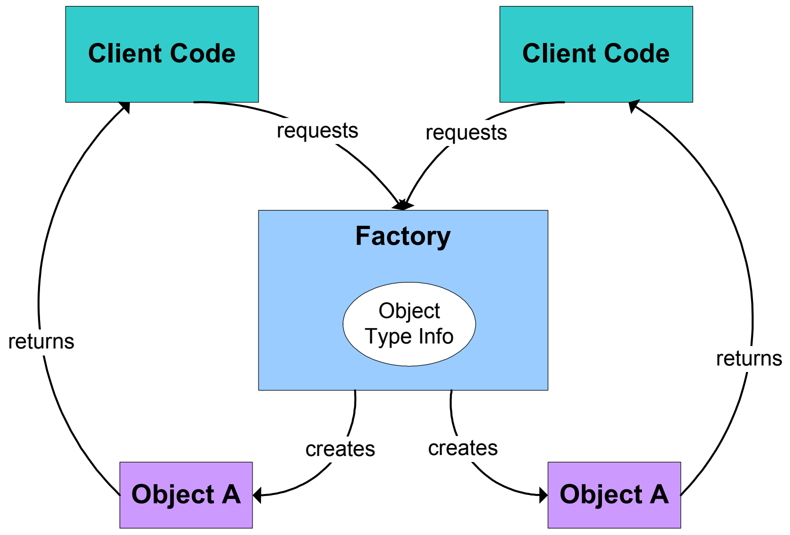
Figure 1-7 MVC Pattern



Factory Pattern

Another design pattern used in Point-of-Sale code is the Factory pattern. The intent of the Factory pattern is to provide an interface for creating families of related or dependent objects without specifying their concrete classes. The application requests an object from the factory, and the factory keeps track of which object is used. Since the application does not know which concrete classes are used, those classes can be changed at the factory level without impacting the rest of the application. The following diagram illustrates this pattern.

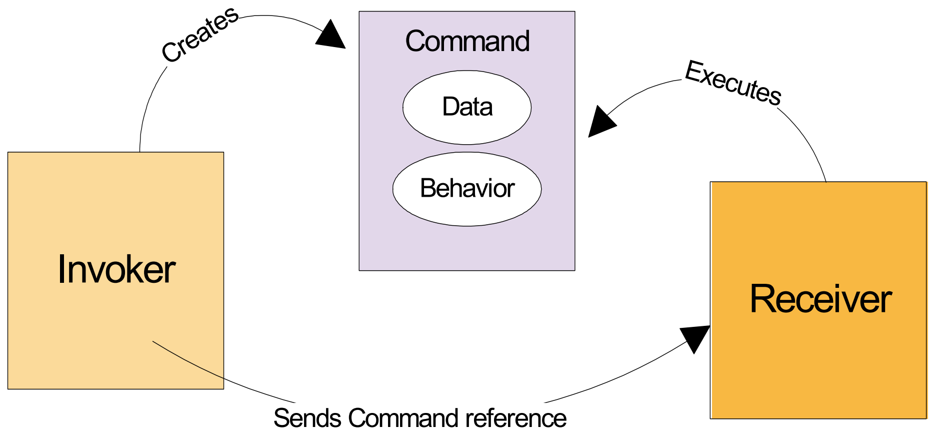
Figure 1-8 Factory Pattern



Command Pattern

Sometimes it is necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request. The Command pattern encapsulates a request as an object. The design abstracts the receiver of the Command from the invoker. The command is issued by the invoker and executed on the receiver. The following diagram illustrates the Command pattern. It is used in the design of the Manager/Technician framework.

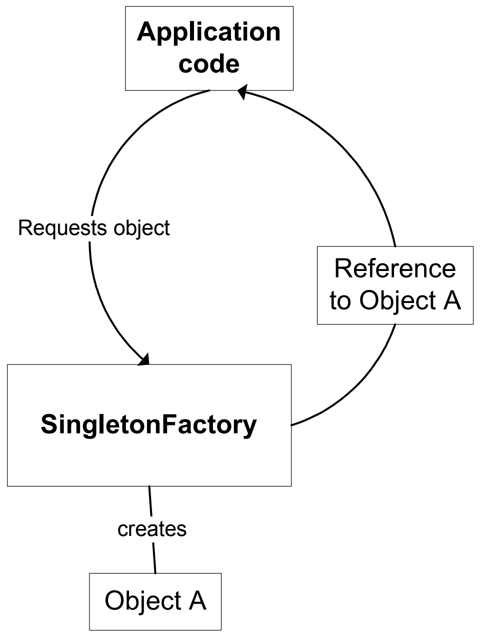
Figure 1-9 Command Pattern



Singleton Pattern

The Singleton pattern ensures a class only has one instance and provides a single, global point of access. It allows extensibility through subclassing. Singletons allow retailers to access the subclass without changing application code. If a system only needs one instance of a class across the system, and that instance needs to be accessible in many different parts of a system, making that class a Singleton controls both instantiation and access. The following patterns illustrates the Singleton pattern:

Figure 1-10 Singleton Pattern



CHAPTER 2

DEVELOPMENT ENVIRONMENT

Overview

A development environment for Point-of-Sale includes all files, tools and resources necessary to build and run the Point-of-Sale application. While development environments may vary depending on the choice of IDE, database, and version control system, configuration of the development environment involves some common steps. This document addresses components that various development environments have in common.

Preparation

The following software resources must be installed and configured before the Point-of-Sale development environment can be set up. Ensure that the following are in place:

- Version control system—The Point-of-Sale source code must be available from a source control system.
- 360Store database—The 360Store database should be installed.
- Eclipse version 3.0 or another IDE—If installing Eclipse, downloads and instructions are available from http://www.eclipse.org/downloads/.
- JDK 1.4—Downloads and instructions are available at http://java.sun.com/downloads/.

Setup

Setting up the development environment requires installing the Point-of-Sale application, populating the database, creating a sandbox, configuring the IDE, and configuring the version control system.

Install Point-of-Sale

Install Point-of-Sale using the installation script. While running the Point-of-Sale installation script, accept the default options even when nothing is selected, except for the options discussed in the following table.

Table 2-1 Point-of-Sale Installation Options

Option	Instruction
Server Tier Type	Choose the Server Tier Type from the following options. • Stand-alone/Collapsed—Choose this option to run the Point-of-Sale client and server functions in one JVM. • N-Tier Client *and* N-Tier Store Server—Choose both of these options to run client and server components on the same machine in separate JVMs.
Database Information	Specify the database type and its location. The default is MySQL 4 in `C:\mysql`.

Build the Database

The tables should be populated with the item, employee, coupon and other retail data that the store needs. If a database is being built from scratch, it needs to be populated with data. The following command can be executed to build the tables and insert a minimal data set.

```
C:\>360store\pos\bin\dbbuild.bat
```

Create a Sandbox

If you plan to retrieve all the source code with the version control system, create a local sandbox with only one directory such as the following.

```
C:\mySandbox\
```

Otherwise, create a local working directory with src, config, and locales\en_US subdirectories. This allows the application code to find all the top-level directories. The following lists the directories that should be created.

```
C:\mySandbox\
C:\mySandbox\src
C:\mySandbox\config
C:\mySandbox\locales\en_US
```

Configure the IDE

The following configuration enables your IDE to build and run the Point-of-Sale application.

1. Set the JRE System Library. In the IDE preferences, point to the JRE included in the JDK installed earlier.

Point to the root of the Java directory in which JDK 1.4 was installed, not the JRE directory in the Point-of-Sale installation directory. For example, if the JDK directory is named `C:\jdk1.4.1`, the JRE Home Directory would be `C:\jdk1.4.1`.

2. Specify the path for the source directories on the build path to be the same as the directory or directories created for the sandbox.
3. Specify the following jars on the build path in the order described in the following table. These directories are the same as the directories in `C:\360store\pos\logs\classpath.log`.

Table 2-2 Build Path

Order	Directory
1	`C:\360store\pos\lib`
2	`C:\360store\pos\lib\locales`
3	`C:\360store\pos\3rdparty\lib`
4	`C:\360store\pos\3rdparty\lib\ibm\surepos750`
5	`C:\360store\pos\3rdparty\special_jars`
6	`C:\360store\360common\lib`
7	`C:\360store\360common\jms\jboss\lib`
8	`C:\360store\360common\3rdparty\lib`
9	`C:\360store\360common\3rdparty\special_jars`

4. Set the launch properties listed in the table below.

   The program arguments differ depending on the Server Tier type chosen during the Point-of-Sale installation. This option is determined by the Server Tier Type selected.

Table 2-3 Launch Properties

Property	Value
main class	`com.extendyourstore.foundation.config.TierLoader`
program arguments	If the Tier type is Stand-alone, the program argument is `classpath:\\config\conduit\CollapsedConduitFF.xml`. If the Tier type is N-Tier Client and N-Tier Server, there are two sets of launch properties. The Store Server launch setting has its program argument set to `classpath:\\config\conduit\StoreServerConduit.xml`. The Client launch setting has its program argument set to `classpath:\\config\conduit\ClientConduit.xml`. Wait for the StoreServerConduit to finish starting before launching the ClientConduit.
classpath	Add the database runtime directory to the classpath. To find this path, open `C:\360store\pos\logs\classpath.log` and search for the local database directory. Also, add the installation config directory. Choose `C:\360Store\posconfig`.

Update Java Security and Policy files

Copy the `java.security` and `java.policy` files dropped by the Point-of-Sale installation, located in `C:\360Store\jre\lib\security`. Paste these files in the `java\jre\lib\security` directory for the JDK that the IDE is referencing.

Configure the Version Control System

Each file from the source code repository should be retrieved to the proper location in your sandbox. To do this, set the workfile location of the root of each of the product components displayed in the version control system, such as 360common. Each workfile location should be set to the local sandbox. For example, if your sandbox is named `C:\mySandbox`, the root of the product components should point to `C:\mySandbox`.

Run Point-of-Sale

To verify the setup, run the Point-of-Sale application using the following steps.

1. Start the 360Store Database.

   Regardless of where the database is located, the service should be started before the Point-of-Sale application is started. If the database was just built, the database server is already running. If it is not running, the batch file provided by the installation script can be executed as follows:
   ```
   C:\>360Store\pos\bin\dbstart.bat
   ```
2. Build the project.
3. Run Point-of-Sale from the IDE.

CHAPTER 3

UI FRAMEWORK

This chapter describes the User Interface (UI) Framework that is part of the 360Platform architecture. The UI Framework encompasses all classes and interfaces included in 360Platform to support rapid development of UI screens. It enables the building of custom screens using existing components.

Overview

For ease of development, the UI Framework hides many of the implementation details of Java UI classes and containment hierarchies by moving some of the UI specification from Java code into XML. This eases screen manipulation and layout changes affecting the look and feel of the entire screen, subsets of screens, and portions of a screen. This table provides a general description of features of the UI Framework.

Table 3-1 UI Framework Features

Feature	Description
Common Design	All UI implementations share code and extend or implement base UI classes that are provided as part of 360Platform. The UI Framework provides basic functionality that does not need to be duplicated within each application.
Reuse	The UI Framework allows bean classes to be independent, thereby supporting their reuse. A UI Technician can be used with multiple applications and UI Framework components can be used across multiple features in an application.
Externally Configurable Screens	The UI Framework enables you to configure screens outside the code to accommodate applications that change frequently. The external screen configurations can be updated to use new 360Platform or application-specific components as they are developed.
Support for Internationalization	The UI Framework provides hooks for implementing internationalization, including language and locale independence.
Extensibility and Flexibility	Additional formats for specifications can be defined without affecting the internal UI Framework classes. Portability is achieved through the use of the Java language and flexible layout managers.

The UI Framework is the set of classes and interfaces that define the elements and behavior of a window-based UI Subsystem. It defines a structure for defining user interfaces. The following table briefly describes the components of the framework. This chapter discusses these components in more detail.

Table 3-2 UI Framework Components

Name	Description
Display	A display is the root container for the UI application window. Displays are any subclass of java.awt.Container that implement EYSRootPaneContainer.
Screen	A screen is a user-level snapshot of a UI window as it relates to an application. The screen is composed of displays, template areas, assignment beans, and listeners. Each of these parts can be individually configured and reassembled to compose the screen.
Template	A template divides the display into areas that contain the layout information used to place the information on the display. Templates can be interchanged to define screen layouts within an application. Each screen specifies the template that is associated with the screen.
Area	An area is a layout placeholder for UI components that operate together to perform a function. Each area contains a layout constraint that dictates how the area is placed on the display.
Bean	A bean is a user interface component or group of components that operate together to provide some useful functionality. For example, a bean could be an input form or group of navigation buttons.
Connection	A connection captures relationships between beans, or between devices and beans. When a bean or device generates an event, another bean responds with a change in behavior or visual display.
Listener	A listener provides a mechanism for reacting to user interface events.

Screens

Generally, for each package in an application, one UI script in the form of an XML file is created to define the screens for the given package. However, because many screens share basic components, certain components are defined in a default UI script. These basic screen components, including displays, templates, and default screens, are defined in `src\com\extendyourstore\pos\config\defaults\defaultuicfg.xml`. Overlay screens are then defined in the UI script for the given package. This section describes the components that are used to build Point-of-Sale screens, except for beans which are described in the next section.

Displays define window properties. They are basic containers with dimensions and a title defined. In Point-of-Sale, only two types of windows can be displayed at the same time—the main application window and a window displaying the Help browser. The following table describes the two types of displays.

Table 3-3 Display Types

Name	Description
EYSPOSDisplaySpec	A 600x800 container for all application screens
HelpDialogDisplaySpec	A 600x800 container for Point-of-Sale Help screens

Templates divide displays into geographical areas. The GridBagLayout is used to define the attributes of each area. The following table describes the typical use of each template.

Table 3-4 Template Types

Name	Typical Use
BrowserTemplateSpec	Back Office screens within the Point-of-Sale application
EYSPOSTemplateSpec	Point-of-Sale screens without required fields
HelpBrowserTemplateSpec	Point-of-Sale help screens
ValidatingTemplateSpec	Point-of-Sale screens with required fields that display an information panel below the work area

Default screens are partially-defined screens that represent elements common to multiple screens. Default screens are based on one display and one template. Default screens map beans to the commonly used areas of the template and define listeners for the bean. These screens are used by overlay specifications that define more specific screen components. For example, almost all screens in the Point-of-Sale application display a status area region. The text displayed in the status region changes, but the StatusPanelSpec bean is the same from screen to screen, so a default screen would assign this bean to the StatusPanel area defined by a template. The following table lists the areas of the template to which beans are assigned, and the display and template used by each of the six types of default screens.

Table 3-5 Default Screen Types

Name	Typical Use	Display	Template
BrowserDefaultSpec	Back Office screens within the Point-of-Sale application	EYSPOSDisplaySpec	BrowserTemplateSpec
DefaultHelpSpec	Point-of-Sale help screens	HelpDialogDisplaySpec	HelpBrowserTemplateSpec
DefaultValidatingSpec	Point-of-Sale screens with required fields that display an information panel below the work area	EYSPOSDisplaySpec	ValidatingTemplateSpec
EYSPOSDefaultSpec	Point-of-Sale screens without required fields	EYSPOSDisplaySpec	EYSPOSTemplateSpec
ResponseEntryScreenSpec	Point-of-Sale screens with information captured in the response area at the top of the screen	EYSPOSDisplaySpec	EYSPOSTemplateSpec

Each screen in Point-of-Sale has an overlay screen defined in a UI script in the package to which it belongs or in a package higher in the hierarchy. For example, the Authorization tour script is found in `src\com\extendyourstore\pos\services\tender\authorization` but the UI script is located in `src\com\extendyourstore\pos\services\tender`. This overlay screen is based on a default screen and defines additional properties for the beans on the areas of the screen. The overlay screen may also specify connections, which are described in "Connections" on page 3-14. The following code sample shows the definition of the ALTERATION_TYPE screen defined in `src\com\extendyourstore\pos\services\alterations\alterationsuicfg.xml`.

Code Sample 3-1 alterationsuicfg.xml: Overlay Screen Definition

```
<OVERLAYSCREEN
        defaultScreenSpecName="EYSPOSDefaultSpec"
        resourceBundleFilename="alterationsText"
        specName="ALTERATION_TYPE">

        <ASSIGNMENT
            areaName="StatusPanel"
            beanSpecName="StatusPanelSpec">
            <BEANPROPERTY
                propName="screenNameTag" propValue="AlterationTypeScreenName"/>
        </ASSIGNMENT>

        <ASSIGNMENT
            areaName="PromptAndResponsePanel"
            beanSpecName="PromptAndResponsePanelSpec">
            <BEANPROPERTY
                propName="promptTextTag" propValue="AlterationTypePrompt"/>
        </ASSIGNMENT>

        <ASSIGNMENT
            areaName="LocalNavigationPanel"
            beanSpecName="AlterationsOptionsButtonSpec">
        </ASSIGNMENT>

</OVERLAYSCREEN>
```

Beans

A screen is composed of beans mapped to specific areas on the screen. All beans are defined in `src/com/extendyourstore/pos/ui/beans`. The beans described in this section are commonly used in screen definitions. Each description provides bean properties that can be defined in assignments of beans to areas. By the Java reflection utility, properties defined in XML files invoke `set()` or `create()` methods in the bean class that accept a single string parameter or multiple string parameters.

The following section covers the PromptAndResponseBean, DataInputBean, NavigationButtonBean, and DialogBean.

PromptAndResponseBean

The PromptAndResponseBean configures and displays the text in the top areas of a Point-of-Sale screen called the prompt region and the response region. This bean is implemented by `src\com\extendyourstore\pos\ui\beans\PromptAndResponseBean.java` and its corresponding model `PromptAndResponseModel.java`.

Bean Properties and Text Bundle

PromptAndResponsePanelSpec is the name of a bean specification that defines the implementation of the PromptAndResponseBean class. The following code sample shows the bean specification available to all screens, defined in `src\com\extendyourstore\pos\config\defaults\defaultuicfg.xml`.

Code Sample 3-2 defaultuicfg.xml: Bean Specification Using PromptAndResponseBean

```
<BEAN
       specName="PromptAndResponsePanelSpec"
       beanClassName="PromptAndResponseBean"
       beanPackage="com.extendyourstore.pos.ui.beans"
       configuratorPackage="com.extendyourstore.pos.ui"
       configuratorClassName="POSBeanConfigurator"
       cachingScheme="ONE">
</BEAN>
```

The following property names and values can be defined in overlay specifications when specifying attributes of a PromptAndResponseBean.

Table 3-6 PromptAndResponseBean Property Names and Values

Item	Description	Example
enterData	Indicates whether data can be entered in the response area	true
promptTextTag	The label tag that corresponds to the text bundle	GiftCardPrompt
responseField	The type of field expected in the response area (see Field Type section for available types)	`com.extendyourstore.pos.ui.beans.AlphaNumericTextField`
maxLength	Maximum length of response area input	15
minLength	Minimum length of response area input	2
zeroAllowed	Indicates whether a zero value is allowed in the response area	true
negativeAllowed	Indicates whether a negative value is allowed in the response area	false
grabFocus	Indicates whether focus should be grabbed when the screen is first displayed	true

These properties can be defined in UI scripts. The following code sample defines an overlay specification that assigns the PromptAndResponsePanelSpec defined above to the PromptAndResponsePanel. This example from `src\com\extendyourstore\pos\services\tender\tenderuicfg.xml` defines the COUPON_AMOUNT overlay screen for the Tender service. The property that retrieves text from a text bundle is highlighted.

Code Sample 3-3 tenderuicfg.xml: PromptAndResponseBean Property Definition

```
<OVERLAYSCREEN>
              defaultScreenSpecName="ResponseEntryScreenSpec"
              resourceBundleFilename="tenderText"
              specName="COUPON_AMOUNT">
       <ASSIGNMENT
            areaName="PromptAndResponsePanel"
            beanSpecName="PromptAndResponsePanelSpec">
            <BEANPROPERTY
                propName="promptTextTag" propValue="CouponAmountPrompt"/>
            <BEANPROPERTY
                propName="responseField"
                propValue="com.extendyourstore.pos.ui.beans.CurrencyTextField"/>
            <BEANPROPERTY
```

```
                propName="maxLength" propValue="9"/>
        </ASSIGNMENT>
...
</OVERLAYSCREEN>
```

The string that should be displayed as the prompt text is defined in a resource bundle. In the resource bundle for the Tender service, which for the en_US locale is defined in `locales\en_US\config\ui\bundles\tenderText_en_US.properties`, the following includes a line that defines the CouponAmountPrompt.

Code Sample 3-4 tenderText_en_US.properties: PromptAndResponseBean Text Bundle Example

```
PromptAndResponsePanelSpec.CouponAmountPrompt=Enter coupon amount and press Next.
```

Tour Code

In the Tour code, bean models are created to hold the data on the bean components. The following table lists some of the important methods in the PromptAndResponseModel class.

Table 3-7 PromptAndResponseModel Important Methods

Method	Description
`boolean isSwiped()`	Returns the flag indicating whether a card is swiped
`void setsScanned(boolean)`	Sets the flag indicating whether a code is scanned
`boolean isResponseEditable()`	Returns the flag indicating whether the response area is editable
`void setGrabFocus(boolean)`	Sets the flag indicating whether focus should stay on the response field

The following sample from `src\com\extendyourstore\pos\services\tender\GetPurchaseOrderAmountSite.java` shows creation of a PromptAndResponseModel, prefilling of data in the model, and display of the model on which the PromptAndResponseModel is set.

Code Sample 3-5 GetPurchaseOrderAmountSite.java: Creating and Displaying PromptAndResponseModel

```
PromptAndResponseModel responseModel = new PromptAndResponseModel();
Locale locale = LocaleMap.getLocale(LocaleConstantsIfc.USER_INTERFACE;
responseModel.setResponseText(balance.toFormattedString(locale));
POSBaseBeanModel baseModel = new POSBaseBeanModel();
baseModel.setPromptAndResponseModel(responseModel);
ui.showScreen(POSUIManagerIfc.PURCHASE_ORDER_AMOUNT, baseModel);
```

For internationalization, Point-of-Sale can use multiple locales at any given time at a register. There is one default locale, one UI locale based on employee-specific locale, and one customer display and customer receipt locale based on customer-specific locale.

The screen constant, PURCHASE_ORDER_AMOUNT, is mapped to an overlay screen name found in the UI script for the package. The screen constants are defined in `src\com\extendyourstore\pos\ui\POSUIManagerIfc.java`.

The following sample from PurchaseOrderNumberEnteredRoad.java in the same directory shows how to retrieve data from the PromptAndResponseModel in a previous screen. To arrive at this code, a purchase order number is entered and the user presses Next. This line of code gets the purchase order number from the previous screen.

Code Sample 3-6 PurchaseOrderNumberEnteredRoad.java: Retrieving Data From PromptAndResponseModel

```
String poNumber = ui.getInput();
```

DataInputBean

The DataInputBean is a standard bean that displays a form layout containing data input components and labels. This bean is implemented by `src\com\extendyourstore\pos\ui\beans\DataInputBean.java` and its corresponding model `DataInputBeanModel.java`. Field components are commonly defined with the FIELD element when defining a bean with the DataInputBean, as shown in the code sample below.

Bean Properties and Text Bundle

The DataInputBean has two properties that can be defined in UI scripts, which override the settings in the field specifications.

Table 3-8 DataInputBean Property Names and Values

Item	Description	Example
labelTags	Sets the property bundle tags for the component labels	NameLabel,AddressLabel,StateLabel
labelTexts	Sets the text on the component labels	Name,Address,State

The label tag is used for internationalization purposes, so the application can look for the correct text bundle in each language. The label tag overrides the value of the labelText field. The following code from `manageruicfg.xml` shows a field specification defined in a DataInputBean bean specification.

Code Sample 3-7 manageruicfg.xml: Bean Specification Using DataInputBean
```
<BEAN
        specName="RegisterStatusPanelSpec"
        configuratorPackage="com.extendyourstore.pos.ui"
        configuratorClassName="POSBeanConfigurator"
        beanPackage="com.extendyourstore.pos.ui.beans"
        beanClassName="DataInputBean">

        <FIELD fieldName="storeID"
               fieldType="displayField"
               labelText="Store ID:"
               labelTag="StoreIDLabel"
               paramList="storeNumberField"/>
         ...
</BEAN>
```
The strings that should be displayed as labels on the screen are defined in a resource bundle. In the resource bundle for the Manager service, which for the en_US locale is defined in `locales\en_US\config\ui\bundles\managerText_en_US.properties`, the following line of code defines the StoreIDLabel.

Code Sample 3-8 managerText_en_US.properties: DataInputBean Text Bundle Example
```
RegisterStatusPanelSpec.StoreIDLabel=Store ID:
```
Fields do not have to be defined in the UI script. They can be defined in the beans and models instead. In the overlay screen specification, two bean properties that can be set are OptionalValidatingFields and RequiredValidatingFields. If the fields are optional and the user enters information in them, then they are validated. If the user does not enter any information, the fields are not validated. On the other hand, required fields are always validated.

Tour Code

Bean models are created to hold the data managed by the bean. This protects the bean from being changed. A bean can only be accessed by a model in the Tour code. The following table lists some of the important methods in the DataInputBeanModel class.

Table 3-9 DataInputBeanModel Important Methods

Method	Description
`String getValueAsString(String)`	Returns the value of the specified field as a string
`int getValueAsInt(String)`	Returns the value of the specified field as an integer
`void setSelectionValue(String, Object)`	Sets the value of the specified field in a vector to the specified value
`void setSelectionChoices(String, Vector)`	Sets the value of the specified field to the specified vector of choices
`void clearAllValues()`	Clears the values of all the fields

The following sample from `src\com\extendyourstore\pos\services\admin\parametermanager\SelectParamStoreSite.java` shows creation of a DataInputBeanModel and prefilling of data in the model.

Code Sample 3-9 SelectParamStoreSite.java: Creating and Displaying DataInputBeanModel

```
DataInputBeanModel beanModel = new DataInputBeanModel();
 beanModel.setSelectionChoices("choiceList", vChoices);
 beanModel.setSelectionValue("choiceList", (String)vChoices.firstElement());
```

The following sample from Tour code shows how to retrieve data from the DataInputBeanModel. In this example from `src\com\extendyourstore\pos\services\admin\parametermanager\StoreParamGroupAisle.java`, after the model is created and displayed by the code from the previous code sample, the model is retrieved from the UI Manager, and data is retrieved from the model.

Code Sample 3-10 StoreParamGroupAisle.java: Retrieving Data from DataInputBeanModel

```
DataInputBeanModel model =
       (DataInputBeanModel)ui.getModel(POSUIManagerIfc.PARAM_SELECT_GROUP);
ParameterCargo cargo = (ParameterCargo)bus.getCargo();
 String val = (String)model.getSelectionValue("choiceList");
cargo.setParameterGroup(val);
```

NavigationButtonBean

The NavigationButtonBean represents a collection of push buttons and associated key stroke equivalents. This bean is implemented by `src\com\extendyourstore\pos\ui\beans\NavigationButtonBean.java` and its corresponding model `NavigationButtonBeanModel.java`. The global navigation area and the local navigation area both use the NavigationButtonBean.

Bean Properties and Text Bundle

The LocalNavigationPanel and GlobalNavigationPanel bean specifications both use the NavigationButtonBean. Bean properties are described only for the GlobalNavigationPanelSpec because the LocalNavigationPanelSpec typically sets its properties in the bean specification and not the overlay specification.

LocalNavigationPanel

The only property available to the NavigationButtonBean in XML is used to enable and disable buttons. When setting the states of buttons on a LocalNavigationPanel, the buttons are usually defined with the BUTTON element in the bean specification as in the following code sample. In fact, any bean that extends NavigationButtonBean, such as ValidateNavigationButtonBean, can define its buttons in the bean specification.

This example from `src\com\extendyourstore\pos\services\customer\customericfg.xml`, defining the CustomerOptionsButtonSpec bean specification for the Customer service, shows how button text on a NavigationButtonBean is defined in a UI script.

Code Sample 3-11 customericfg.xml: Bean Specification Using NavigationButtonBean

```
<BEAN
        specName="CustomerOptionsButtonSpec"
        configuratorPackage="com.extendyourstore.pos.ui"
        configuratorClassName="POSBeanConfigurator"
        beanPackage="com.extendyourstore.pos.ui.beans"
        beanClassName="NavigationButtonBean">

       <BUTTON
            actionName="AddBusiness"
            enabled="true"
            keyName="F4"
            labelTag="AddBusiness"/>
...
</BEAN>
```

The string that should be displayed on the buttons on the GlobalNavigationPanel is defined in a resource bundle. In the resource bundle `customerText_en_US.properties`, the following entry defines the label for the AddBusiness button.

Code Sample 3-12 customerText_en_US.properties: NavigationButtonBean Text Bundle Example

```
CustomerOptionsButtonSpec.AddBusiness= Add Business
```

GlobalNavigationPanel

The GlobalNavigationButtonBean extends the NavigationButtonBean. The following code sample shows the GlobalNavigationPanel bean specification defined in `src\com\extendyourstore\pos\config\defaults\defaultuicfg.xml`. The bean class is a subclass of NavigationButtonBean.

Code Sample 3-13 defaultuicfg.xml: Bean Specification Using GlobalNavigationButtonBean

```
<BEAN
        specName="GlobalNavigationPanelSpec"
        configuratorPackage="com.extendyourstore.pos.ui"
        configuratorClassName="POSBeanConfigurator"
        beanPackage="com.extendyourstore.pos.ui.beans"
        beanClassName="GlobalNavigationButtonBean"
        cachingScheme="ONE">
...
</BEAN>
```

The following property names and values can be defined in UI scripts when specifying attributes of a GlobalNavigationButtonBean.

Table 3-10 GlobalNavigationButtonBean Property Names and Values

Item	Description	Example
manageNextButton	Indicates whether the bean should manage the enable property of the Next button	true
buttonStates	Sets the buttons with the action names listed to the specified state	Help[true],Clear[false],Cancel[fal se],Undo[true],Next[false]

These properties can be defined in overlay specifications, as in the following code sample from `tenderuicfg.xml`.

Code Sample 3-14 tenderuicfg.xml: GlobalNavigationButtonBean Property Definitions

```
<OVERLAYSCREEN>
              defaultScreenSpecName="EYSPOSDefaultSpec"
              resourceBundleFilename="tenderText"
              specName="TENDER_OPTIONS">
        <ASSIGNMENT
            areaName="GlobalNavigationPanel"
            beanSpecName="GlobalNavigationPanelSpec">
            <BEANPROPERTY
                propName="manageNextButton"
                propValue="false"/>
            <BEANPROPERTY
                propName="buttonStates"
                propValue="Help[true],Clear[false],Cancel[false],Undo[true],Next[false]"/>
        </ASSIGNMENT>
...
</OVERLAYSCREEN>
```

Tour Code

In the Tour code, bean models are created to hold the data on the bean components. The following table lists some of the important methods in the NavigationButtonBeanModel class.

Table 3-11 NavigationButtonBeanModel Important Methods

Method	Description
`ButtonSpec[] getNewButtons()`	Returns an array of new buttons
`void setButtonEnabled(String, boolean)`	Sets the state of the specified action name of the button (the name of the letter the button mails)
`void setButtonLabel(String, String)`	Sets the label of the button using the specified action name of the button (the name of the letter the button mails)

The following sample from `src\com\extendyourstore\pos\services\tender\PricingOptionsSite.java` shows creation of a NavigationButtonBeanModel, prefilling of data in the model, and display of the model on which the NavigationButtonBeanModel is set.

Code Sample 3-15 PricingOptionsSite.java: Creating and Displaying NavigationButtonBeanModel

```
NavigationButtonBeanModel navModel = new NavigationButtonBeanModel();
navModel.setButtonEnabled("TransDiscAmt",true);
navModel.setButtonEnabled("TransDiscPer",true);
model.setLocalButtonBeanModel(navModel);
ui.showScreen(POSUIManagerIfc.PRICING_OPTIONS, model);
```

The screen constant, PRICING_OPTIONS, is mapped to an overlay screen name found in the UI script for the package. The screen constants are defined in `src\com\extendyourstore\pos\ui\POSUIManagerIfc.java`.

DialogBean

The DialogBean provides dynamic creation of dialog screens. This bean is implemented by `src\com\extendyourstore\pos\ui\bundles\DialogBean.java` and its corresponding model `DialogBeanModel.java`.

Bean Properties and Text Bundle

DialogSpec is the name of a bean specification that defines an implementation of the DialogBean class. The following code sample shows the bean specification defined in `src\com\extendyourstore\pos\services\common\commonuicfg.java`.

Code Sample 3-16 commonuicfg.java: Bean Specification Using DialogBean

```
<BEAN
       specName="DialogSpec"
       configuratorPackage="com.extendyourstore.pos.ui"
       configuratorClassName="POSBeanConfigurator"
       beanPackage="com.extendyourstore.pos.ui.beans"
       beanClassName="DialogBean">
       <BEANPROPERTY propName="cachingScheme" propValue="none"/>
</BEAN>
```

The DialogBean does not have any properties that can be defined in UI scripts. Therefore, all its properties are defined in Tour code discussed in the next section. The following code sample defines the message displayed in the dialog. This example from `src\com\extendyourstore\pos\services\inquiry\InquirySlipPrintAisle.java` shows how text on a DialogBean is defined in Java code.

Code Sample 3-17 InquirySlipPrintAisle.java: DialogBean Label Definition

```
DialogBeanModel model = new DialogBeanModel();
model.setResourceID("Retry");
```

The resourceID corresponds to the name of the text bundle. For all dialog screens in the en_US locale, `dialogText_en_US.properties` contains the bundles that define the text on the screen, as shown in the following code.

Code Sample 3-18 dialogText_en_US.properties: DialogBean Text Bundle Example

```
DialogSpec.Retry.title=Device is offline
DialogSpec.Retry.description=Device offline
DialogSpec.Retry.line2=<ARG>
DialogSpec.Retry.line5=Press the Retry button to attempt to use the device again.
```

Tour Code

In the Tour code, bean models are created to hold the data on the bean components. The following table lists some of the important methods in the DialogBeanModel class.

Table 3-12 DialogBeanModel Important Methods

Method	Description
setResourceID(String)	Used to locate screen text in the text bundle
setArgs(String [])	Sets a string of arguments to replace <ARG> tags in the text bundle

Table 3-12 DialogBeanModel Important Methods

Method	Description
setButtonLetter(int, String)	Sets the specified letter to be sent when the specified button is pressed
setType(int)	Sets the flag indicating whether focus should stay on the response field

The following sample from `src\com\extendyourstore\pos\services\tender\LookupStoreCreditSite.java` shows creation of a DialogBeanModel, prefilling of data in the model, and display of the model on which the DialogBeanModel is set.

Code Sample 3-19 LookupStoreCreditSite.java: Creating and Displaying DialogBeanModel

```
DialogBeanModel dialogModel = new DialogBeanModel();
DialogModel.setResourceID(“InvalidCashAmount”);
dialogModel.setArgs(new String[] ={cashAmt});
dialogModel.setType(DialogScreensIfc.ACKNOWLEDGEMENT);
dialogModel.setButtonLetter(BUTTON_OK, "Failure");
ui.showScreen(POSUIManagerIfc.DIALOG_TEMPLATE, dialogModel);
```

The screen constant, DIALOG_TEMPLATE, is mapped to an overlay screen name found in the UI script for the package. The screen constants are defined in `src\com\extendyourstore\pos\ui\POSUIManagerIfc.java`.

When setting the dialog type, refer to the following table that lists the available dialog types as defined by constants in `src\com\extendyourstore\pos\ui\DialogScreensIfc.java`. For each dialog type, the buttons on the dialog are specified. In most cases, the letter sent by the button has the same name as the button, except for the two types noted.

Table 3-13 Dialog Types

Dialog Type	Button(s)	Details
ACKNOWLEDGEMENT	Enter	Button sends OK letter
CONFIRMATION	Yes, No	
CONTINUE_CANCEL	Continue, Cancel	
ERROR	Enter	Button sends OK letter, Screen displays red in the title bar
RETRY	Retry	
RETRY_CANCEL	Retry, Cancel	
RETRY_CONTINUE	Retry, Continue	
SIGNATURE		Places a signature panel to capture the customer’s signature

When setting a letter to a button, refer to the following table that lists the available button types also defined in `DialogScreensIfc.java`. These constants are used as arguments to DialogBean methods that modify button behavior.

Table 3-14 Button Types

Button	ButtonID
Enter, OK	BUTTON_OK
Yes	BUTTON_YES
No	BUTTON_NO

Table 3-14 Button Types

Button	ButtonID
Continue	BUTTON_CONTINUE
Retry	BUTTON_RETRY
Cancel	BUTTON_CANCEL

Field Types

This section defines field types available to all beans. The following field types may be used by all the beans, but DataInputBean is the only bean that understands the FIELD element. In other words, DataInputBean is the only bean that defines fields in bean specifications.

These field types correspond to `create()` methods in UIFactory.java, such as `createCurrencyField()` and `createDisplayField()`. The application framework uses reflection to create the fields. Therefore, the field names in the following table can be set as the fieldType attribute in an XML bean specification using the DataInputBean class. The corresponding parameter list is a list of strings that can be set as the paramList attribute.

Table 3-15 Field Types and Descriptions

Name	Description	Parameter List Strings (no spaces allowed)
alphaNumericField	allows letters and/or numbers, no spaces or characters	name,minLength,maxLength
constrainedPasswordField	text where the view indicates something was typed, but does not show the original characters	name,minLength,maxLength
constrainedTextAreaField	multi-line area that allows plain text, with restrictions on length	name,minLength,maxLength,columns,wrapStyle,lineWrap
constrainedField	allows letters, numbers, special characters, and punctuation, with restrictions on length	name,minLength,maxLength
currencyField	allows decimal numbers only, representing currency, with two spaces to the right of the decimal	name,zeroAllowed,negativeAllowed,emptyAllowed
decimalField	allows decimal numbers only	name,maxLength,negativeAllowed,emptyAllowed
displayField	display area that allows a short text string or an image, or both	name
driversLicenseField	allows alphanumeric text that can contain '*' or ' '	name
EYSDateField	allows only whole numbers and the special character /—the format is MM/DD/YYYY	name
EYSTimeField	allows only whole numbers and the special character:—the format is HH:MM	name
nonZeroDecimalField	allows non-zero decimal numbers only	name,maxLength

Table 3-15 Field Types and Descriptions

Name	Description	Parameter List Strings (no spaces allowed)
numericField	allows integers only, no special characters or letters	name,maxLength,minLength
nonZeroNumericField	allows non-zero integers only	name,maxLength,minLength
textField	allows letters, numbers, special characters, and punctuation	name
validatingTextField	line of text that can be validated by length requirements	name

Connections

Connections configure the handling of an event in the UI Framework. They are used to define inter-bean dependencies and behavior and to tie the bean event responses back to the business logic. When one bean generates an event, another bean can be notified of the event. Connections have a source bean, a Listener Type for the target, and a target bean.

Connections attach a source bean to a target bean, which receives event notifications from the source bean. The Listener Type specifies which type of events can be received. The XML in the following sections are found in `com\extendyourstore\pos\services\tender\tenderuicfg.xml`. Other listeners used in Point-of-Sale include ConfirmCancelAction, HelpAction, and CloseDialogAction.

ClearActionListener

ClearActionListener is an interface that extends ActionListener in Swing to make it unique for its use in Point-of-Sale. The following code shows how this listener is used in an overlay specification. Adding the ClearActionListener allows the Clear button to erase the text in the selected field in the work area when the Clear button on the GlobalNavigationPanelSpec is clicked.

Code Sample 3-20 tender.xml: ClearActionListener XML tag

```
<CONNECTION
        listenerInterfaceName="ClearActionListener"
        listenerPackage="com.extendyourstore.pos.ui.behavior"
        sourceBeanSpecName="GlobalNavigationPanelSpec"
        targetBeanSpecName="CreditCardSpec"/>
```

DocumentListener

DocumentListener is an interface defined in Swing. The following code shows how this listener is used in an overlay specification. Adding the DocumentListener allows the Clear button on the GlobalNavigationPanelSpec to be disabled until input is entered in the selected field on the work area.

Code Sample 3-21 tender.xml: DocumentListener XML tag

```
<CONNECTION
        listenerInterfaceName="DocumentListener"
        listenerPackage="javax.swing.event"
        sourceBeanSpecName="CreditCardSpec"
        targetBeanSpecName="GlobalNavigationPanelSpec"/>
```

ValidateActionListener

ValidateActionListener is an interface that extends ActionListener in Swing to make it unique for its use in Point-of-Sale. The following code shows how this listener is defined in an overlay specification. Adding the ValidateActionListener allows the CreditCardSpec to recognize when the Next button on the GlobalNavigationPanelSpec is clicked, resulting in the validation of the required fields on the work area. If the required fields are empty, an error dialog appears stating that the required field(s) must have data.

Code Sample 3-22 tender.xml: ValidateActionListener XML tag

```
<CONNECTION
        listenerInterfaceName="ValidateActionListener"
        listenerPackage="com.extendyourstore.pos.ui.behavior"
        sourceBeanSpecName="GlobalNavigationPanelSpec"
        targetBeanSpecName="CreditCardSpec"/>
```

The fields that are required must be specified for this listener in the overlay specification for the target bean, as in the following XML from `tenderuicfg.xml`.

Code Sample 3-23 tenderuicfg.xml: ValidateActionListener Required Fields

```
<ASSIGNMENT
   areaName="WorkPanel"
   beanSpecName="CreditCardSpec">
   <BEANPROPERTY
      propName="RequiredValidatingFields" propValue="CreditCardField,ExpirationDateField"/>
   </ASSIGNMENT>
```

Text Bundles

Currently, over forty text bundles exist for the Point-of-Sale application. Many of these bundles are service-specific. A properties file with the same name exists for every language, located in `locales\<locale name>\config\ui\bundles` with the locale name appended to the filename. For example, the Customer service would have its text defined in the `customerText_en_US.properties` file in English, and the text would be similarly defined in the `customerText_es_PR.properties` file in Spanish. The following examples show the same text bundle in different languages.

Code Sample 3-24 customerText_en_US.properties: Text Bundle in English

```
Common.Add=Add Customer
Common.AddBusiness=Add Business
```

Code Sample 3-25 customerText_es_PR.properties: Text Bundle in Spanish

```
Common.Add=Añadir Cliente
Common.AddBusiness=Añadir Negocio
```

A similarly named properties file would exist for each locale. Because they are discussed earlier in the chapter, service-specific bundles and the dialogText bundle are not described in this section.

receiptText

From `src\com\extendyourstore\pos\config\bundles\BundleConstantsIfc.java`, the following code sets a string constant for the receiptText bundle.

Code Sample 3-26 BundleConstantsIfc.java: String Constant for receiptText

```
public static String RECEIPT_BUNDLE_NAME = "receiptText";
```

In Tour Code, methods to print the receipt exist which call methods on the Utility Manager to get specified text. The following code is from the `printDocument()` method in `src\com\extendyourstore\pos\receipt\GiftCardInquirySlip.java`.

Code Sample 3-27 GiftCardInquirySlip.java: Tour Code to Print Receipt

```
UtilityManager utility = (UtilityManager)
       Gateway.getDispatcher().getManager(UtilityManagerIfc.TYPE);
Properties slipProps = utility.getBundleProperties(BundleConstantsIfc.RECEIPT_BUNDLE_NAME,
                                        UtilityManagerIfc.RECEIPT_BUNDLES,
                                        LocaleMap.getLocale(LocaleConstantsIfc.RECEIPT));
String title = slipProps.getProperty("GiftCardTitle", "Gift Card Inquiry").toString();
String giftCardNumber = slipProps.getProperty("GiftCardAccount", "Gift Card #").toString();
...define additional properties...
printLineCentered(title);
printLine("");
printLine(blockLine(new StringBuffer(" " + giftCardNumber), new StringBuffer(cardNumber)));
```

In the `receiptText_<locale>.properties` file, the corresponding text is defined.

Code Sample 3-28 receiptText_en_US.properties: Text Bundle

```
Receipt.GiftCardTitle=BALANCE INQUIRY
Receipt.GiftCardAccount=Account #
```

parameterText

In overlay specifications, the parameterText bundle is specified to define the text for particular screens. For example, the following code from `src\com\extendyourstore\pos\services\admin\parametermanager\parameteruicfg.xml` defines text for the PARAM_SELECT_PARAMETER overlay screen. On this screen, the names of the parameters found in the parameterText properties file are displayed.

Code Sample 3-29 parameteruicfg.xml: Overlay Specification Using parameterText

```
<OVERLAYSCREEN
        defaultScreenSpecName="EYSPOSDefaultSpec"
        resourceBundleFilename="parameterText"
        specName="PARAM_SELECT_PARAMETER">
```

In the utility package, the ParameterManager is used to retrieve parameter values. The following code from `src\com\extendyourstore\pos\utility\GiftCardUtility.java` shows how a parameter is retrieved from the ParameterManager. The handle to the ParameterManager, `pm`, is passed into the method but originally retrieved by the code `ParameterManagerIfc pm = (ParameterManagerIfc)bus.getManager(ParameterManagerIfc.TYPE);`

Code Sample 3-30 GiftCardUtility.java: Tour Code to Retrieve Parameter

```
public static final String DAYS_TO_EXPIRATION_PARAMETER = "GiftCardDaysToExpiration";
daysToExpiration = pm.getIntegerValue(DAYS_TO_EXPIRATION_PARAMETER);
```

In the `parameterText_<locale>.properties` file, the corresponding text is defined. This text is displayed on the Parameter List screen when viewing Security options and choosing the Tender parameter group.

Code Sample 3-31 parameterText_en_US.properties: Text Bundle

```
Common.GiftCardDaysToExpiration=Days To Giftcard Expiration
```

The value of the parameter is defined in `config\parameter\application\application.xml` by the code sample below. Each parameters belongs to a group, a collection of related parameters.

Code Sample 3-32 application.xml: Definition of Parameter

```
<PARAMETER name="GiftCardDaysToExpiration"
         type="INTEGER"
         final="N"
         hidden="N">
         <VALIDATOR class="IntegerRangeValidator"
            package="com.extendyourstore.foundation.manager.parameter">
            <PROPERTY propname="minimum" propvalue="1" />
            <PROPERTY propname="maximum" propvalue="9999" />
         </VALIDATOR>
         <VALUE value="365"/>
</PARAMETER>
```

CHAPTER 4

TOUR FRAMEWORK

Overview

The Tour framework is a component of the 360Platform layer of the Point-of-Sale architecture. The Tour framework implements a state engine that controls the workflow of the application. Tour scripts are a part of this framework; they define the states and transitions that provide instructions for the state engine that controls the workflow. Java classes are also part of this framework; they implement the behavior that is accessed by the tour engine, based on instructions in the tour scripts. The Tour Guide application assists with this development effort by generating the tour scripts rapidly and creating stubs of the necessary Java classes.

Tour Components

The tour metaphor helps the user visualize how the 360Platform engine interacts with application code. In the following description of the metaphor, the words in *italics* are part of a simple tour script language that 360Platform uses to represent the application elements.

Tour Metaphor

For a moment, imagine that you are a traveler about to embark on a journey. You have the itinerary of a business traveler (changeable at any time), your luggage, and transportation. In addition, you have a video camera (*TourCam*) to record your tour so you can remember it later.

You leave on your journey with a specific goal to achieve. Your itinerary shows a list of tours that you can choose from to help you accomplish your task. Each tour provides a tour *bus* with a *cargo* compartment and a driver. Each driver has a *map* that shows the various *service regions* that you can visit. These regions are made up of *site*s (like cities) and transfer *station*s (bus stations, airports, etc.). The maps show a finite number of lanes, which are either *road*s joining one site to another or *aisles* within one site. To notify the driver to start the bus and drive, you must send a *letter* to the driver. The driver reads the name on the letter and looks for a lane that matches the letter.

When a matching letter is found, the driver looks for a traffic *signal* on the road. If there is no signal, the driver can traverse the road. If there is a signal, the driver can traverse the road only if the signal is green.

If the signal is red, the driver attempts to traverse the next alternative road that matches the letter. If the driver cannot find any passable road, he or she returns to the garage. When you arrive at a site or traverse a lane, you may perform an action to achieve your goal, like take a picture of the countryside.

Upon arriving at a transfer station, you immediately transfer to another service, and you load a portion of your cargo onto a *shuttle* and board the shuttle. The shuttle takes you and your cargo to the bus that runs in the map of the other tour. Upon arrival at the new bus, you unload the shuttle and load the new bus. Then the new driver starts the bus and your journey begins in the new tour. When the transfer tour itinerary is complete, you load whatever cargo you want to keep onto a shuttle and return to the original tour bus. At that time, you unload the shuttle and continue your tour.

These tour script components map to terms in the metaphor. The tour metaphor provides labels and descriptions of these components that improve understanding of the system as a whole. The following table includes a metaphor description and a technical description for the basic metaphor components.

Table 4-1 Metaphor Components

Name	Metaphor Description	Technical Description
Service	A group of related cities, for example "A Mediterranean Tour"	An implementation of workflow and behavior for a set of functionality
Bus	The vehicle that provides transportation from city to city	The entity that follows the workflow between the sites
Cargo	The baggage that the traveler takes with him/her from city to city	The data that follows the workflow, modified as necessary
Site	A city	A function point in the workflow
Road	A path the bus takes to get from one city to another	A transition that takes place based on an event that changes the state
Aisle	A path the traveler takes while staying on the same bus in the same city	An action that takes place based on an event, without leaving the current state
Letter	A message the bus driver receives instructing him/her to perform an action	A message that causes a road or aisle to be taken

When given a use case, create a tour script by identifying components for the tour metaphor. Strategies for identifying components are listed in the table below. The following sections describe each component in more detail.

Table 4-2 Component Identification Strategies

Component	How to Identify
Service	A service generally corresponds to a set of related functionality.
Site	Sites generally correspond to points in the workflow that need input from outside the tour. Outside input sources include the user interface, the database, and devices among others.
Road	At a site, look at the ways control can be moved to another site. There is one road for each of these cases.
Aisle	At a site, there might be a task that you want to handle in a separate module and then return to the site when the task is complete. There is one aisle for each of these cases.
Letter	Letters generally correspond to buttons on a UI screen and responses from the database and devices. Look for the events that move control from one site to another or prompt additional behavior within a site to help identify letters.

Follow the naming conventions in the Development Standards when deciding the names for the components. It is important to understand that the tour metaphor is not only used to describe the interaction of the components, but the component's names are used in the code. By convention, a site named GetTender has a Java class in the package named GetTenderSite.java that performs the work done at the site.

Service and Service Region

Tours provide a way of grouping related functionality to minimize maintenance and increase reusability. All tours provide a bus to maintain state and cargo for data storage. All sites, lanes, and stations contained within a tour have access to these resources. A service is essentially a tour, but the terms *service* and *service region* are used by the Tour Guide application to refer to a tour. Furthermore, in the Point-of-Sale source code, the tours are found in the `src\com\extendyourstore\pos\services` directory. Generally, this chapter uses the word *tour* to refer to a tour. The word *service* and phrase *service region* are used in this section because they are elements in the XML code.

The service region contains all functionality related to running the application when no exceptions are encountered. The following code sample from `src\com\extendyourstore\pos\services\tender\tender.xml` shows the definition of a service and service region in a tour script.

Code Sample 4-1 tender.xml: Definition of Service and Service Region

```
<SERVICE name="Tender" package="com.extendyourstore.pos.services.tender" tourcam="ON">
        <SERVICECODE>
        ...definition of letters, siteaction classes, and laneaction classes...
        </SERVICECODE>
        <MAP>
                <REGION region="SERVICE" startsite="GetTender">
                ...definition of sites, stations, and lanes...
                </REGION>
        </MAP>
</SERVICE>
```

As shown in the code sample, there are two main sections of a tour script. The SERVICECODE element defines the Java classes in the tour and the letters that may be sent in the tour code or by the user. The MAP element links the classes and letters to the sites and lanes. In the following sections, code samples are shown from both sections of the tour script.

Bus

The bus object is passed as a parameter to all tour methods called by the tour engine. Methods can be called on the bus to get access to the cargo, managers and other state information. The following code sample from `src\com\extendyourstore\pos\services\tender\GetCheckInfoSite.java` shows a reference to the bus.

Code Sample 4-2 GetCheckInfoSite.java: Retrieving Cargo from Bus

```
TenderCargo cargo = (TenderCargo) bus.getCargo();
```

Tourmap

One problem of tour scripts is that they can be difficult to customize for a particular retailer's installation. The new tourmap feature allows customizations to be made more easily on existing tour scripts. Tour components and tour scripts can be referenced by logical names in the tour script and mapped to physical names in a tourmap file, making it easier to use the product tour and just change the pieces that need to be changed for a customer implementation. In addition, with tourmaps, components and scripts can be overridden based on a country, so files specific to a locale are implemented when appropriate.

The tourmap does not allow you to modify the structure of the tour, specifically the following:

- does not allow you to add or remove sites
- does not allow you to add or remove roads and aisles
- does not allow you to specify a tour spanning multiple files (i.e. “tour inheritance”)

Of particular note is the last bullet: the tourmap does not allow you to assemble fragments of xml into one cohesive tour script. After the application is loaded, there is only be one tour script that maps to any logical name.

The functionality of tourmapping is implemented via one or more tourmap files. Multiple tourmap files can be specified via the `config\tourmap.files` properties. `tourmap.files` is a comma delimited list of tourmap files. As each file is loaded, the application checks the `country` property of the tourmap file. The order of files is significant because later files override any values specified in previous files. A file that overrides a similarly-named file is called an overlay.

Each tourmap file begins with a root element, `tourmap`, which has an optional `country` attribute. The `tourmap` elements contains multiple `tour` elements, each one of which describes a tour's logical name, its physical file, and any overlays to apply. For instance, a simple tourmap might look like the following:

Code Sample 4-3 Sample Tourmap

```
<?xml version="1.0" encoding="UTF-8"?>
<tourmap
   country="CA"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="com/extendyourstore/foundation/tour/dtd/tourmap.xsd">

    <tour name="testService">
       <file>classpath://com/extendyourstore/foundation/tour/engine/tourmap.testservice.xml</file>
       <SITE
          name="siteWithoutAction"
          useaction="com.extendyourstore.foundation.tour.engine.actions.overlay.OverlaySiteAction"/>
       <SITEACTION
          class="SampleSiteAction"
          replacewith="com.extendyourstore.foundation.tour.engine.actions.overlay.OverlaySiteAction"/>

    </tour>
</tourmap>
```

In this instance, the tour with the logical name “`testService`” references the file `com\extendyourstore\foundation\tour\engine\tourmap.testservice.xml`. Additionally, the values for SITE and SITEACTION are replaced. Note, however, that because of the `country` in the `tourmap` element, this only happens when the default locale of the application is a Canadian locale, if `locale=Locale.CANADA_FRENCH` or `locale=Locale.CANADA`.

Tourmaps are used not only to override XML attributes, but they are used also when the workflow needs to be changed.

Cargo

Cargo is data that exists for the length of the tour in which it is used. Any data that needs to be used at different tour components such as sites and aisles needs to be stored on the cargo. Cargo always has a Java class. The following code sample from `tender.xml` defines the Tender cargo.

Code Sample 4-4 tender.xml: Definition of Cargo

```
<CARGO class="TenderCargo">
</CARGO>
```

With the concept of a tourmap, a cargo class can be overridden with another class. This allows you to override the class name for a customer implementation yet still keep the same workflow for the customer as in the product. The following tourmap definition specifies the class to override and the new class to use in place of the original class. Note that replacewith is a fully qualified classname, with both package and classname specified, unlike the class attribute.

Code Sample 4-5 tourmap.xml: Example of Overriding Cargo Class

```
<CARGO class="TenderCargo" replacewith="com.extendyourstore.cargo.SomeCargo"/>
```

Sites

Sites correspond to nodes in a finite state machine and cities in the tour metaphor. Sites are usually used as stopping places within the workflow. Arrival at a site usually triggers access to an external interface, such as a graphical user interface, a database, or a device. Sites always have a corresponding siteaction class.

The `tender.xml` code sample below contains the site information from the two main parts of a tour script: the XML elements SERVICECODE and MAP, respectively.

Code Sample 4-6 tender.xml: Definition of Site Class

```
<SITEACTION  class="GetTenderSite"/>
```

Code Sample 4-7 tender.xml: Mapping of Site to SiteAction

```
<SITE  name="GetTender" siteaction="GetTenderSite">
  ...definition of lanes...
</SITE>
```

With the concept of Tourmap, a site's `siteaction` can be overridden with another class. This allows you to override the class name for a customer implementation yet still keep the same workflow for the customer as in the product. The following tourmap definition specifies the class to override and the new class to use in place of the original class. Note that replacewith is a fully qualified classname, with both package and classname specified, unlike the class attribute.

Code Sample 4-8 tourmap.xml: Overriding Siteaction With Tourmap

```
<SITEACTION class="GetTenderSite" replacewith="com.extendyourstore.actions.SomeOtherSiteAction"/>
```

System Sites

System sites are defined by the 360Platform engine but can be referenced within a tour script. For example, a road defined by a tour script can have a system site as its destination. Each system site must have a unique name in the tour script file. The following code from `tender.xml` shows the definition of two system sites. The Final system site stops a bus and returns it to the parent bus, and LastIndexed resumes the normal bus operation after an exception.

Code Sample 4-9 tender.xml: Definition of System Sites

```
<REGION>
       <MAP>
       ...definition of sites, lanes, and stations...
       <SYSTEMSITE name="Final" action="RETURN" />
       <SYSTEMSITE name="LastIndexed" action="BACKUP" />
       </MAP>
</REGION>
```

Letters

Letters are messages that get sent from the application code or the user interface to the tour engine. Letters indicate that some event has occurred. Typically, letters are sent by the external interfaces, such as the graphical user interface, database, or device to indicate completion of a task.

Lanes are defined as roads and aisles. When the system receives a letter, it checks all lanes defined within the current site or station to see if the letter matches the letter for a lane. If no matching lane is found, the letter is ignored. Letters do not have a Java class associated with them.

Standard letter names are used in the application, such as Success, Failure, Undo, and Cancel. The following code sample shows `tender.xml` code that defines letters. The definition is added to the SERVICECODE XML element.

Code Sample 4-10 tender.xml: Definition of Letter

```
  <LETTER  name="Credit"/>
```

Roads

Roads provide a way for the bus to move between sites and stations. Each road has a name, destination, and letter that activates the road. A road may or may not have a laneaction class, depending on whether the road has behavior; only roads that have behavior require a class. Roads are defined within site definitions because they handle letters received at the site.

Following is `tender.xml` code that shows the definition of a road. The definition is added to the SERVICECODE XML element. After the first code sample is another sample that maps the road to a site and letter, which is contained in the MAP section of the tour script.

Code Sample 4-11 tender.xml: Definition of Road Class

```
<LANEACTION  class="ValidCreditInfoEnteredRoad"/>
```

Code Sample 4-12 tender.xml: Mapping of Road to Site

```
<SITE  name="GetCreditInfo" siteaction="GetCreditInfoSite">
  <ROAD
    name="ValidCreditInfoEntered"
    letter="Valid"
    laneaction="ValidCreditInfoEnteredRoad"
    destination="GetTender"
    tape="ADVANCE"
    record="OFF"
    index="OFF">
  </ROAD>
  ...other lanes defined...
</SITE>
```

With the concept of Tourmap, a road's laneaction can be overridden with another class. This allows you to override the class name for a customer implementation yet still keep the same workflow for the customer as in the product. The following tourmap definition specifies the class to override and the new class to use in place of the original class. Note that replacewith is a fully qualified classname, with both package and classname specified, unlike the class attribute.

Code Sample 4-13 tourmap.xml: Example of Overriding Site Laneaction

```
<LANEACTION class="ValidCreditInfoEnteredRoad"
replacewith="com.extendyourstore.actions.SomeOtherLaneAction"/>
```

Common Roads

The COMMON element is defined in the REGION element of the tour script. The COMMON element can contain roads that are available to all sites and stations in a tour. Common roads have the same attributes as roads defined within a site, but they are defined outside of a site so they can be accessed by all sites. If a common road and a tour road are both activated by the same letter, the site road is taken. The following is an example that differentiates common roads from tour roads.

Code Sample 4-14 Example of Common Road

```
<MAP>
     <REGION region="SERVICE" startsite="Example">
          <COMMON>
               <ROAD name="QuitSelected" letter="exit"
                    destination="NamedIndex"
                    tape="REWIND"/>
                    <COMMENT>
                    </COMMENT>
               </ROAD>
          </COMMON>
          <SITE name="RequestExample" siteaction="RequestExampleSite">
               <ROAD name="ExampleSelected" letter="next"
                     laneaction="ExampleSelectedRoad"
                     destination="ShowExample"
                     tape="ADVANCE"
                     record="OFF"
                     index="ON"/>
                    <COMMENT>
                    </COMMENT>
               </ROAD>
     </REGION>
</MAP>
```

Aisles

Aisles provide a means for moving within a site and executing code. Aisles are used when a change is required but there is no reason to leave the current site or station. Each aisle contains a name, a letter, and a laneaction. Aisles always require a Java class because they must have behavior since they do not lead to a different site or station like roads.

Following is the `tender.xml` code that shows the definition of an aisle. The definition is added to the SERVICECODE XML element. The second code sample from the same tour script maps an aisle to the site and letter, which is contained in the MAP section.

Code Sample 4-15 tender.xml: Definition of Aisle Class

```
<LANEACTION  class="CardInfoEnteredAisle"/>
```

Code Sample 4-16 tender.xml: Mapping of Aisle to Site

```
<SITE  name="GetCreditInfo" siteaction="GetCreditInfoSite">
  <AISLE
```

```
    name="CardInfoEntered"
    letter="Next"
    laneaction="CardInfoEnteredAisle">
  </AISLE>
...other lanes defined...
</SITE>
```

With the concept of Tourmap, an aisle's laneaction can be overridden with another class. This allows you to override the class name for a customer implementation yet still keep the same workflow for the customer as in the product. The following tourmap definition specifies the class to override and the new class to use in place of the original class. Note that replacewith is a fully qualified classname, with both package and classname specified, unlike the class attribute.

Code Sample 4-17 tourmap.xml: Example of Overriding Aisle Laneaction

```
<LANEACTION class="CardInfoEnteredAisle" replacewith="com.extendyourstore.actions.SomeOtherLaneAction"/
>
```

Stations and Shuttles

Transfer stations are used to transfer workflow to another tour and return once the tour workflow has completed. A transfer station describes a location where another tour is started and the passenger exits one bus and enters the bus for another tour.

Transfer stations specify the name of the nested tour and define data transport mechanisms called shuttles. Shuttles are used to transfer cargo to and from the nested tour. These shuttles are either launch shuttles or return shuttles. Launch shuttles transfer cargo to the nested tour and the return shuttles transfer newly acquired cargo from the nested tour to the calling tour. Shuttles have Java classes associated with them, but stations do not.

The following code samples from `src\com\extendyourstore\pos\services\tender\tender.xml` contain the station and shuttle information from the SERVICECODE and MAP elements in the tour script, respectively.

Code Sample 4-18 tender.xml: Definition of Shuttle Class

```
  <SHUTTLE  class="TenderAuthorizationLaunchShuttle"/>
```

Code Sample 4-19 tender.xml: Mapping of Station to Service and Shuttle Classes

```
  <STATION
    name="AuthorizationStation"
    servicename="classpath://com/extendyourstore/pos/services/tender/authorization/Authorization.xml"
    targettier="APPLICATIONTIER"
    launchshuttle="TenderAuthorizationLaunchShuttle"
    returnshuttle="TenderAuthorizationReturnShuttle">
    ...lane definitions to handle exit letter from nested service...
  </STATION>
```

The servicename can be defined as a logical name like “authorizationService” and mapped to a filename is the tourmap file. The shuttle names can also be overridden in the tourmap file. This allows you to override the class name for a customer implementation yet still keep the same workflow for the customer as in the product. The code samples below illustrate this.

Code Sample 4-20 tourmap.xml: Example of Mapping Servicename

```
<tour name=”authorizationService”>
       <file>classpath://com/extendyourstore/pos/services/tender/authorization/Authorization.xml</file>
</tour>
```

Code Sample 4-21 tourmap.xml: Example of Overriding Shuttle Name

```
<SHUTTLE class="TenderAuthorizationLaunchShuttle"
replacewith="com.extendyourstore.shuttles.NewShuttle"/>
```

Nested tours operate independently, with their own XML script and Java classes. Stations and shuttles simply provide the functionality to transfer control and data between two independent tours.

Signals

Signals direct the tour to the correct lane when two or more lanes from the same site or station are activated by the same letter. The lane that has a signal that evaluates to true is the one that is traversed. Each signal has an associated Java class. Signal classes evaluate the contents of the cargo and do not modify data.

The following code sample lists the `tender.xml` code that relates to the definition of two roads with Light signals defined. The definition is added to the SERVICECODE XML element, whereas the road description is added to the MAP XML element. The negate tag negates the Boolean value returned by the specified signal class.

Code Sample 4-22 tender.xml: Definition of Traffic Signal

```
  <SIGNAL  class="IsAuthRequiredSignal"/>
```

Code Sample 4-23 tender.xml: Signal Processing With Negate Tag

```
<STATION>
  name="AuthorizationStation"
  <ROAD
    name="AuthorizationRequested"
    letter="Next"
    destination="AuthorizationStation"
    tape="ADVANCE"
    record="OFF"
    index="OFF">
    <LIGHT  signal="IsAuthRequiredSignal"/>
  </ROAD>
  <ROAD
    name="BalancePaid"
    letter="Next"
    destination="CompleteTender"
    tape="ADVANCE"
    record="OFF"
    index="OFF">
    <LIGHT  signal="IsAuthRequiredSignal" negate="Y"/>
  </ROAD>
  ...additional lane definitions...
</STATION>
```

Exception Region

Continuing the tour metaphor, the bus could break down at any time. If the bus driver detects that the bus has broken down, the bus driver takes the bus to the nearest Garage system site. Once the bus is in the garage, the mechanic assumes control of and diagnoses the breakdown.

- If the mechanic is able to restore the cargo to a valid state, the mechanic informs the bus driver by traversing to the Resume system site. The bus driver subsequently resumes driving by resetting the bus at the site where the breakdown occurred.
- If the mechanic is not successful in repairing the bus, the mechanic stops the bus, and mails the parent tour a letter informing it of the breakdown.

- If there is no mechanic within the tour, the bus driver stops the bus, and mails the parent tour a letter informing it of the breakdown. The bus completes its tour when it arrives at the final site.

The exception region includes the functionality for handling exceptions. It can contain sites, roads, and stations just like the service region. There are two ways to exit the exception region: at the Return system site or the Resume system site. Return shuts down the application, and Resume starts the application at the last visited site or station in the service region.

The mechanic operates within the exception region of the tour. Any exception that occurs within the tour region where the bus driver operates is converted to an Exception letter and is passed to the mechanic. When the exception is being processed, the mechanic assumes control of the bus and processes all incoming letters. If the application developer has created an exception region for the mechanic, the Exception letter is processed using application-specific actions and traffic lights. However, if the exception region does not exist, the mechanic stops the bus and informs the parent bus of the problem.

Depending on the application definition, recovery from exceptions can result in a rollback, resumption, or a restart of the bus.

Role of Java Classes

All the code samples in this chapter have been from tour scripts. Tour scripts exist in the form of one XML file per tour. The tour script refers to Java classes that implement specific behavior, such as the siteaction and laneaction attributes. A tour has the following Java classes:

- One for the cargo
- One for each site
- One for each aisle
- One for each road that implements behavior
- One for each shuttle
- One for each signal

The Tour Guide application can generate Java stubs for these classes, but the code in the methods for the sites, roads, aisles, and cargo classes needs to be written. The following table lists methods that the tour engine looks for when it arrives at a specified place in the tour.

Table 4-3 System-called Methods

Class	**Method(s)**
Site	`arrive()`, `depart()`
Road (if behavior)	`traverse()`
Aisle	`traverse()`
Shuttle	`load()`, `unload()`
Signal	`roadClear()`
Cargo	<none>

Tour Cam

TourCam allows you to navigate backward through your application in a controlled manner while requiring minimal programming to accomplish the navigation. It provides the ability to back up from a tour or process by tracking the state of the cargo and the location of the tours. TourCam is turned on or off at the tour level. If there is no reason to back up, TourCam should not be turned on.

The ability to backup or restore data to a previous state is accomplished using TourCam. TourCam is used to record the bus path through the map, as well as the associated cargo changes. TourCam is described using the TourCam metaphor. The words in *italics* in the following paragraphs are the TourCam-specific terms.

A bus driver *records* the progress along the bus route using TourCam. The bus driver records snapshots of the passenger cargo immediately before traversing a road. Each snapshot is mounted in a frame within the current *tape*. The frame is stamped with the current road. Using this method, the bus driver can retrace steps through the map. If the frame is *indexed*, the driver stops at that index when retracing his steps.

The bus driver may adjust the TourCam tape while the bus traverses a road between sites.

- The bus driver can *advance* the current TourCam tape, and add the next road and snapshot of the cargo as a frame in the tape.
- The bus driver can *discard* the current TourCam tape, and replace it with a blank tape.
- The bus driver can *rewind* the current tape to restore the cargo to be consistent with a previously visited site.
- The bus driver can *splice* the current TourCam tape by removing all frames that were recorded since a previously visited site.

When the passenger wants to back up, they instruct the bus driver to traverse a road whose destination is the *Backup* system site. The backup road can inform the bus driver to rewind or splice the TourCam tape while retracing its path along the last recorded road. Similarly, the passenger can instruct the bus driver to traverse a road to a specific, previously visited site. That road effectively backs up the bus when it instructs the bus driver to rewind or splice the TourCam tape.

When the passenger wants to end the trip, they instruct the bus driver to travel down a road whose destination is the Return system site. The final road may advance or discard the TourCam tape. A passenger may return to the tour if they back into the parent transfer station. If the TourCam tape is advanced, a return visit retraces the path through the map in reverse order. If the TourCam tape is discarded, all return visits start at the start site, as if the passenger were visiting the tour for the first time.

Attributes

The TourCam processing model places all undo actions on roads and treats sites and stations as black boxes. The tour attribute that turns TourCam on or off is tourcam. The following code from `tender.xml` shows the location in the tour script where the tourcam is set. The default value is OFF.

Code Sample 4-24 tender.xml: Definition of tourcam

```
<SERVICE
    name="Tender"
    package="com.extendyourstore.pos.services.tender"
    tourcam="ON">
```

The rest of the TourCam attributes are set on the road element in the MAP section of the tour script. The following code from `tender.xml` shows a road definition with these attributes set.

Code Sample 4-25 tender.xml: Definition of Road With TourCam Attributes

```
<SITE name="GetGiftCertificateInfo" siteaction="GetGiftCertificateInfoSite">
        <ROAD name="GiftCertificateInfoEntered"
              letter="Next"
              laneaction="GiftCertificateInfoEnteredRoad"
              destination="GetTender"
              tape="ADVANCE"
              record="OFF"
              index="OFF">
        ...definitions of lanes...
</SITE>
```

The following table lists TourCam attributes and their values.

Table 4-4 Road Tag Element Attributes

Tag	Description	Values	Default
tape	Indicates what tour action to take when traversing the road.	ADVANCE – Adds a frame representing this road to the tourcam tape DISCARD – Discards the entire tour cam tape REWIND – Back up to the site specified by the ‘destination’ while calling the backup method on all roads SPLICE – Back up to the site specified by the ‘destination’ without calling the backup method on any roads	ADVANCE
record	Indicates that a snapshot of the cargo should be recorded and saved on the tourcam tape	ON—Record a snapshot OFF – Do not record a snapshot	ON
index	Indicates that an index should be placed on the tourcam tape when this road is traversed	ON—Place an index on the tape OFF—Do not place an index on the tape	ON
namedIndex	Indicates that a named index should be placed on the tourcam tape when this road is traversed	Any string value is allowed	None
destination	Used when the tape has a value of REWIND or SPLICE to indicate where the tourcam should back the bus up to	<SITENAME>—The name of a site to back up to. The site must be in the current tour. LastIndexed – The backup should end at the site that is the origin of the first road found with an unnamed index. NamedIndex – The backup should end at the at the site that is the origin of the first road found with the named index specified by the *named Index*.	None

Each of the following combinations describes a combination of settings and how it is useful in different situations. The following tables describe the forward and backward TourCam settings:

Table 4-5 Forward TourCam Settings

Settings	Behavior

`ADVANCE`
`index=ON`
`record=ON`
	This combination permits you to return to the site without specifying it as a destination and storing the state of the cargo. Use this combination if you are entering data and making decisions. The UI provides a method for backing up to the previous step.

`tape=ADVANCE`
`index=OFF`
`record=ON`
	This combination allows you to track visited sites, and allows you to attach undo behavior. However, you cannot back up to this site. A common scenario for use would be for performing external lookups and the user must backup to the site that started the lookup. This combination is used, rather than the following combination, when changes made to the cargo that must be reversible.

`tape=ADVANCE`
`index=OFF`
`record=OFF`
	This combination is useful for sites that require external setup from another site, but do not result in a significant change in cargo. You cannot back up to a site that uses these settings and you cannot restore cargo at this site. As with the previous combination, these settings are used for sites that perform external lookups.

`tape=ADVANCE`
`index=ON`
`record=OFF`
	This combination is used when a site does not do anything of significance to cargo. You would use this setting if a site prompts to choose an option from a list and there is a default, or to respond to a yes/no dialog and you want to ensure the data collected at the site is reset.

`tape=ADVANCE`
`namedIndex=LOGIN`
	This combination is used when you want the application to be able to return to a specific index, even if the backup begins in a child tour.

`tape=DISCARD`	This combination is used when you want the application flow only to go forward from this site. For example, after a user tenders a credit card for a sale, the user cannot backup to enter, delete or modify items. This setting does not permit you to backup or restore cargo to a previously recorded site.

Table 4-6 Backup Tour Cam Settings

Settings	Behavior

`destination=BACKUP`
`tape=REWIND`
	This combination returns the application to the previously marked site and makes the snapshot available for undo. This is the preferred method of performing a full backup with restore.

`destination=`*`site`*
`tape=REWIND`
	This combination backs up the application until it reaches the specified site. It is only used if the site to which you want to backup does not directly precede the current site or you know that you always want to backup to the specified site. These settings could produce unpredictable results if new sites are later inserted in the map between the current site and the target backup site.

`destination=LastIndexed`
`tape=SPLICE`
	This combination returns the application to the previously marked site without restoring the cargo. These settings are used in scenarios when the cargo is inconsequential.

Table 4-6 Backup Tour Cam Settings

Settings	Behavior

`destination=`*`site`*
`tape=SPLICE`
	This combination backs up the application to the specified site without restoring the cargo. It is used when the cargo is inconsequential, or when you want to loop back to a base site in a tour without permitting backup or undoing cargo after returning to the base site.
	For example, the application starts from a menu and permits the user to back up until a series of steps are complete, but not afterward. In this case, the final road from the last site returns to the menu. The need to use this combination might indicate a design flaw in the tour. The developer should question whether the series of sites that branch from the menu should be a separate tour. If the answer is no, this combination is the solution.

`destination=`*`NamedIndex`*
`namedIndex=LOGIN`
	This combination backs up the application to the origin of the road with the specified named index. This is used to back up to a specific index, even if it was set in a parent tour.

Letter Processing

In the absence of TourCam, processing of letters is straightforward. If the letter triggers a lane, the bus simply traverses the lane. With TourCam enabled, the processing of letters must consider the actions required to retrace the path of the bus. If the letter triggers an aisle, the bus traverses the aisle. There is no backup over an aisle. If the letter triggers a road, `tape=advance` or `tape=discard` indicate a forward direction, and `tape=rewind` or `tape=splice` indicate a backward direction. The destination of the road element is used to indicate the backup destination when `tape=rewind` or `tape=splice`. It can be one of the following values: "LastIndexed", "NamedIndex", or <sitename>.

Cargo Restoration

One of the primary strengths of TourCam is the ability to restore the bus' cargo to a previous state. TourCamIfc provides a mechanism for the bus driver to make and subsequently restore a copy of the cargo when specified by road attributes. Classes that implement TourCamIfc must implement the `makeSnapshot()` and `restoreSnapshot()` methods. An example of this is `src\com\extendyourstore\pos\services\inquiry\giftreceipt\GiftReceiptCargo.java`.

Code Sample 4-26 GiftReceiptCargo.java: TourCamIfc Implementation

```
public class GiftReceiptCargo implements CargoIfc, TourCamIfc
{
        ...body of GiftReceiptCargo class...
        public SnapshotIfc makeSnapshot()
        {
               return new TourCamSnapshot(this);
        }
        public void restoreSnapshot(SnapshotIfc snapshot) throws ObjectRestoreException
        {
               GiftReceiptCargo savedCargo = (GiftReceiptCargo) snapshot.restoreObject();
               this.setPriceCode(savedCargo.getPriceCode());
               this.setPrice(savedCargo.getPrice());
     }
}
```

SnapshotIfc provides a mechanism to create a copy of the cargo. The class that implements SnapshotIfc is responsible for storing information about the cargo and restoring it later, by calling `restoreObject()`.

A shuttle allows the optional transfer of cargo from the calling tour to the nested tour during backups. If defined, this shuttle is used during rewind and splice backup procedures. The classname for the shuttle is specified in the tour script via the `backupshuttle` attribute of the station element.

Code Sample 4-27 Sample Backupshuttle Definition

```
<STATION servicename="foo.xml"
        launchshuttle="MyLaunchShuttle"
        backupshuttle="MyBackupShuttle"\/>
```

Tender Tour Reference

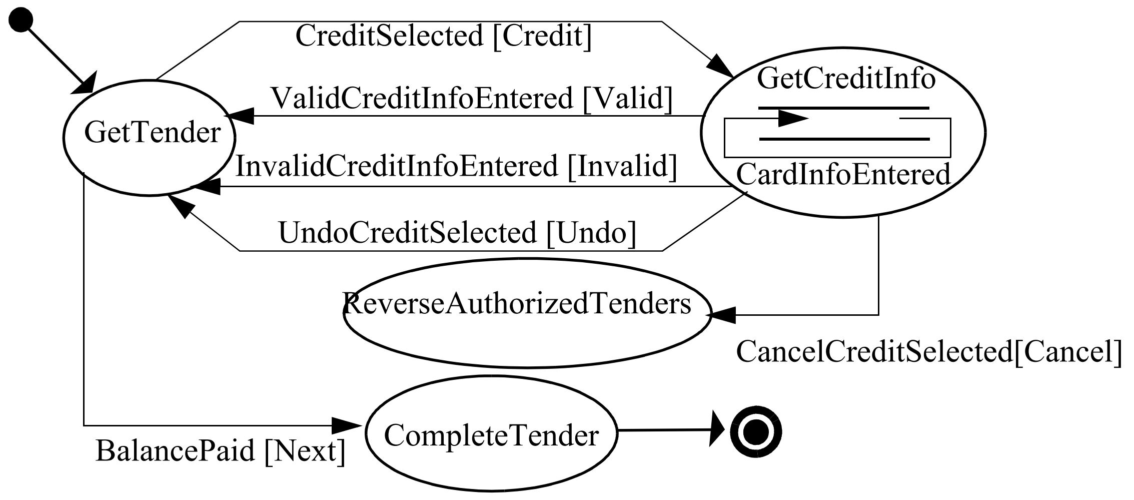
The files in the Tender package can be found in `src\com\extendyourstore\pos\services\tender`. The following table provides resources in the Tender package that are common to all tours.

Table 4-7 Tender Package Components

Resource	Filename	Description
Tour script	`tender.xml`	This file defines the components (sites, letters, roads, etc.) of the Tender tour and the map of the Tender tour.
Tour screens	`tenderuicfg.xml`	This configuration file contains bean specifications and overlay screen specifications for the Tender tour.
Starting site	`GetTenderSite.java`	Tender types are displayed from this site. If the selected tender requires input, it is entered via another site, which then returns control to this site. When the balance due is paid, control is returned to the calling service.
Cargo	`TenderCargo.java`	This class represents the cargo for the Tender tour.
Stations	Names (stations do not have classes): AuthorizationStation PINPadStation AddCustomer AddBusinessCustomer FindCustomer SecurityOverrideStation LinkCustomerStation	These stations provide access to other tours. Each of these stations define one or more shuttle classes which are part of the Tender package. The workflows are defined in other packages, but can be called from the Tender tour. For example, AuthFailedRoad is defined in the Tender tour because it handles the exit letter from the Authorization tour. However, `Authorization.xml`, the workflow for the Authorization tour, is located in `src\com\extendyourstore\pos\services\tender\authorization`.

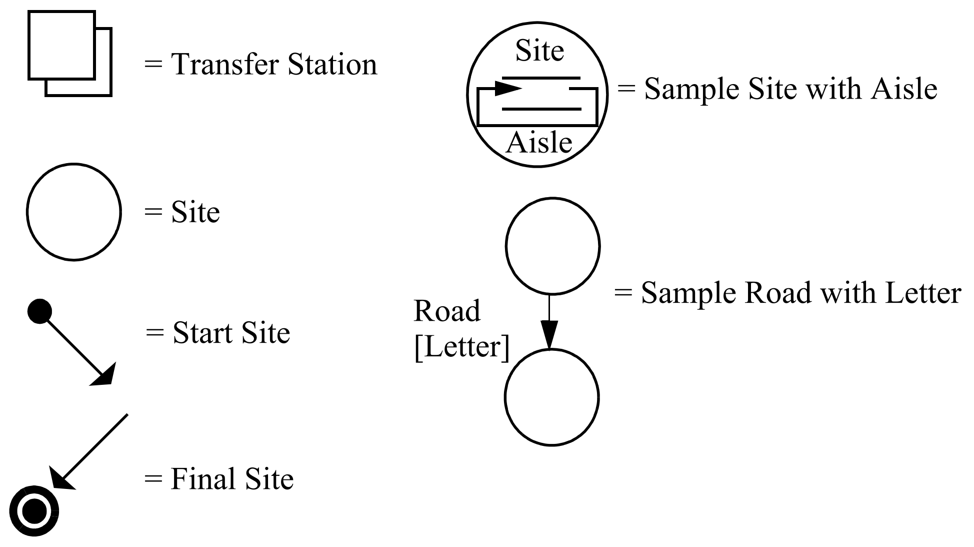
The Tender package is unique in that the workflow is generally similar for all the tender type options available from the main site. For example, if the user chooses to pay by check or credit card, the workflow is similar. When the user cancels the form of payment, the 360Platform engine is directed to the ReverseAuthorizedTenders site. When the user decides to undo the operation, the engine is directed back to the GetTender site. The workflow for the credit card tender option is shown in the following figure.

Figure 4-1 Workflow Example: Tender with Credit Card Option


CreditSelected [Credit]
GetCreditInfo
ValidCreditInfoEntered [Valid]
GetTender
InvalidCreditInfoEntered [Invalid]
CardInfoEntered
UndoCreditSelected [Undo]
ReverseAuthorizedTenders
CancelCreditSelected[Cancel]
CompleteTender
BalancePaid [Next]

The symbols are explained in the figure below.

Figure 4-2 Workflow Symbols



CHAPTER 5

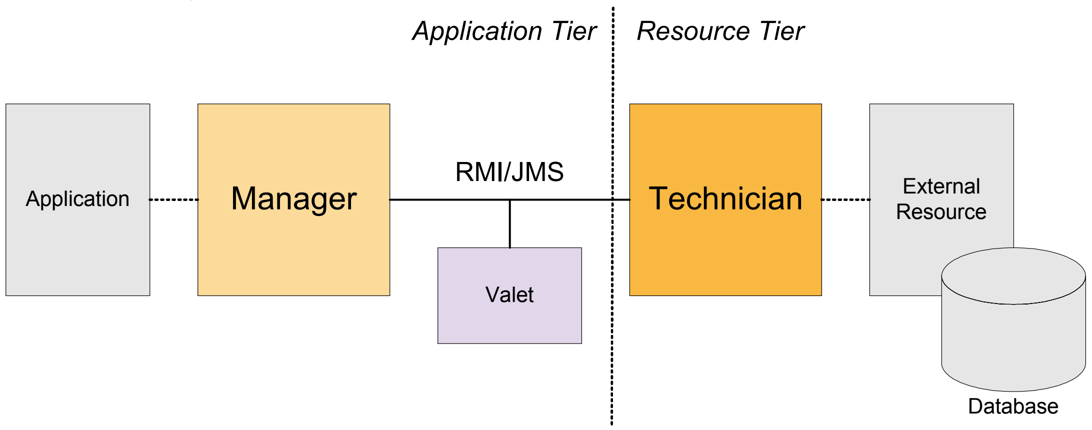
MANAGER/TECHNICIAN FRAMEWORK

This chapter describes the Manager/Technician pair relationship and how it is used to provide business and system services to the application. It also describes how to build a Manager and Technician and provides sample implementation and sample code.

Overview

360Platform provides the technology for distributing business and system processes across the enterprise through plug-in modules called Managers and Technicians. Manager and Technician classes come in pairs. A Manager is responsible for communicating with its paired Technician on the same or different tiers. The Technician is responsible for performing the work on its tier. By design, Managers know how to communicate with Technicians through a pass-through remote interface called a valet. The valet is the component that handles data transfer. The valet can travel across networks. It receives the instructions from the Manager and delivers them to the Technician. A valet follows the Command design pattern, described in the Architecture chapter.

Figure 5-1 Manager, Technician and Valet



There is a M:N relationship between instances of Managers and Technicians. Multiple Managers may communicate with a Technician, or one Manager may communicate with multiple Technicians. While most Managers have a corresponding Technician, there are cases such as the Utility Manager where no corresponding Technician exists.

There are three Manager/Technician categories. These types have different usages and are started differently. The three types are:

- Global—These Managers and Technicians are shared by all tours. They provide global services to applications.
- Session—These Managers and Technicians perform services for a single tour. They are started by each tour and exist for the length of the tour.
- Embedded—Thread Manager is embedded inside the 360Platform engine. It is essential to the operation of the engine. This is currently the only embedded Manager.

Examples of each type are listed in the following table.

Table 5-1 Manager/Technician Type Examples

Manager/Technician Type	Examples
Global	Data Journal Log Resource Tax Timer Tier Trace XML
Session	Device Parameter Session UI Web DomainInterface TenderAuth
Embedded	Thread

Session Managers are started up by the tour bus when a tour is invoked and can only be accessed by the bus in the tour code. Global Managers, on the other hand, can be used at any time and are not specific to any tour. Each type of Manager has a specific responsibility. This table lists the functions of some of the Managers.

Table 5-2 Manager Names and Descriptions

Manager Name	Description
Data	The Data Manager is the system-wide resource through which the application can obtain access to persistent resources. The Data Manager tracks all data stores for the system, and is the mechanism by which application threads obtain logical connections to those resources for persistence operations.
Device	The Device Manager defines the Java interfaces that are available to an application or class for accessing hardware devices, like printers and scanners.

Table 5-2 Manager Names and Descriptions

Manager Name	Description
Journal	The Journal Manager is the interface that is used to write audit trail information, such as start transaction, end transaction, and other interesting register events.
Log	The Log Manager is the interface that places diagnostic output in a common location on one tier for an application, regardless of where the actual tours run.
Parameter	The Parameter Manager is the interface that provides access to parameters used for customization and runtime configuration of applications.
Thread	The Thread Manager is a subsystem that provides system threads as a pooled resource to the system.
Tier	The Tier Manager interface starts a tour session and mails letters to existing tour sessions. The Tier Manager enables the engine to start a tour on any tier specified in a transfer station, regardless of where that tier runs. In addition, the Tier Manager enables a bus to mail a letter to any other existing Bus in the system on any tier.
Timer	The Timer Manager provides timer resources to applications that require them. It does not have a Technician because all timers are local on the tier where they are used.
User Interface	The UI Manager is a mechanism for accessing and manipulating user interface components. The user interface subsystem within a state machine application must also maintain a parallel state of screens, so the appropriate screens can be matched with the application state at all times. The user interface subsystem within a distributed environment must enable application logic to be completely isolated from the user interface components.
XML	The XML Manager locates a specified XML file, parses the file, and returns an XML parse tree.

New Manager/Technician

When creating a new Manager and Technician pair, you must create a Manager and Technician class, a Valet class, and interfaces for each class. Managers are the application client to a Technician service, Technicians do the work, and the valet tells the Technicians what work to do. Managers can be considered proxies for the services provided by the Technicians. Technicians can serve as the interfaces to resources. Managers communicate with Technicians indirectly using valets. Valets can be thought of as commands to be executed remotely by the Technician. Samples for the new classes that need to be created are organized together in the next section.

Requesting services from the Managers only requires familiarity with the interface provided by Managers. However, building a new Manager/Technician pair requires implementing the interfaces for both the new Manager and Technician, as well as creating a Valet class.

Manager Class

A Manager merely provides an API to tour code. It behaves like any other method except that the work it performs may be completed remotely. The input to a Manager is usually passed on to the valet that in turn, passes it on to the Technician, which actually performs the work.

The Manager class provides methods for sending valets to the Technician. The `sendValet()` method makes a single attempt to send a valet to the Manager's Technician. The `sendValetWithRetry()` method attempts to send the valet to the Manager's Technician, and if there is an error, reset the connection to the Technician and then try again.

Managers must implement the ManagerIfc, which requires the following methods:

Table 5-3 ManagerIfc Methods

Method	Description
`MailboxAddress getAddress()`	Gets address of Manager
`Boolean getExport()`	Returns if this Manager is exportable
`String getName()`	Gets name of Manager
`void setExport(Boolean)`	Sets whether the Manager is exportable
`void setName(String)`	Sets name of Manager
`void shutdown()`	Shuts this Manager down
`void startUp()`	Starts this Manager

Often, a subclass of Manager can use these methods exactly as written. Unlike the Technicians, Managers seldom require special startup and shutdown methods, because most Managers have no special resources associated with them.

Manager Configuration

You can provide runtime configuration settings for each Manager using a conduit script. The Dispatcher that loads Point-of-Sale configures the Managers by reading properties from the conduit script and calling the corresponding `set()` method using the Java reflection utility. All properties are set by the Dispatcher before the Dispatcher calls `startUp()` on the Manager.

Every Manager should have the following.

- Name—Tour code typically locates a Manager using its name. Often this name is the same as the name of the class and may be defined as a constant within the Manager. This is what the `getName()` method returns.
- Class—This is the name of the class, minus its package.
- Package—This is the Java package where the class resides.

Managers may have an additional property file defined that specifies additional information such as the definition of transaction mappings. If a separate configuration script is defined, the `startup()` method must read and interpret the configuration script. The following sample from `config\conduit\CollapsedConduitFF.xml` shows this.

Code Sample 5-1 CollapsedConduitFF.xml: Data Manager Configuration

```
<MANAGER name="DataManager" class="DataManager"
```

```
        package="com.extendyourstore.foundation.manager.data">
  <PROPERTY propname="configScript"
            propvalue="classpath://config/manager/PosDataManager.xml" />
</MANAGER>
```

Technician Class

Technicians implement functions needed by Point-of-Sale to communicate with external or internal resources, such as the UI or the store database. Technicians must implement the TechnicianIfc, which requires the following methods:

Table 5-4 TechnicianIfc Methods

Method	Description
`MailboxAddress getAddress()`	Gets address of Technician
`Boolean getExport()`	Checks if this Technician is exportable
`String getName()`	Gets name of Technician
`void shutdown()`	Shuts this Technician down
`void startUp()`	Starts up Technician process

Often, a subclass of Technician can use these methods exactly as written. The most likely methods to require additional implementation are `startUp()` and `shutdown()`, which needs to handle connections with external systems.

Technician Configuration

The Technician is configured within the conduit script. Each Technician should have the following:

- Name—A Manager typically locates its Technician using its name. Often this name is the same as the name of the class and may be defined as a constant within the Technician. This is what `Technician.getName()` returns.
- Class—The name of the class, minus its package
- Package—The Java package where the class resides
- Export—This should be Y if the Technician may be accessed by an external Java process; N otherwise. The value returned by `Technician.getExport()` is based on this. In Technicians, it indicates whether the Technician can be remotely accessed from another tier.
- commScheme (optional) —Specifies the communication scheme used to communicate with the Technician. The default is RMI.
- encryptValets (optional) —Specifies whether the valets should be encrypted during network transmission. The default is N.
- compressValets (optional) —Specifies whether the valets should be compressed during network transmission. The default is N.

Some Technicians may require complex configuration. In cases like this, it may be preferable to define an XML configuration script specific to the Technician, rather than to rely on the generic property mechanism. Therefore, Technicians may have an additional property defined that specifies additional information such as log formats or parameter validators. If a separate configuration script is defined, the `startup()` method must read and interpret the configuration script. The following sample from `config\conduit\CollapsedConduitFF.xml` shows an additional script defined in the configuration of the Tax Technician.

Code Sample 5-2 CollapsedConduitFF.xml: Tax Technician Configuration

```
<TECHNICIAN name="TaxTechnician" class = "TaxTechnician"
       package = "com.extendyourstore.domain.manager.tax"
       export = "Y" >
 <PROPERTY
    propname="taxSpecScript"
    propvalue="classpath://config/tax/TaxTechnicianRates.xml"
 />
</TECHNICIAN>
```

Valet Class

The valet is the intermediary between the Manager and Technician. Valets act as commands and transport information back and forth between the Manager and Technician. Valets must implement ValetIfc, which contains a single method.

Table 5-5 ValetIfc Method

Method	Description
Serializable execute(Object)	Executes the valet-specific processing on the object

The execute method is called by the Technician after the valet arrives at its destination as a result of the Manager’s `sendValet()` or `sendValetWithRetry()` methods, as in the following example from `src\com\extendyourstore\foundation\manager\parameter\ParameterManager.java`.

Code Sample 5-3 ParameterManager.java: Valet Passed By Manager

```
MailboxAddress techAddress = getParameterTechnicianAddress();
retVal = sendValetWithRetry(valet, techAddress);
```

Sample Code

The examples below illustrate the primary changes that need to be made to create a Manager/Technician pair. Note that interfaces also need to be created for the new Manager, Technician, and Valet classes.

Configuration

The conduit script needs to define the location of the Manager and Technician. This code would be found in a conduit script such as `config\conduit\ClientConduit.xml`. These code samples would typically be in different files on separate machines. It would include snippets like the following.

Code Sample 5-4 Sample Manager and Technician Configuration

```
<MANAGER name="MyNewManager"
         class="MyNewManager"
         package="com.extendyourstore.foundation.manager.mynew">
</MANAGER>

<TECHNICIAN name="MyNewTechnician"
         class="MyNewTechnician"
         package="com.extendyourstore.foundation.manager.mynew"
         export="Y" >
        <PROPERTY propname="techField" propvalue="importantVal"/>
        <PROPERTY propname="configScript"
                  propvalue="classpath://com/extendyourstore/pos/config/myconfigscript.xml"/>
    </TECHNICIAN>
```

Tour Code

Tour code might include a snippet like the following, which might be located in `src\com\extendyourstore\pos\services`.

Code Sample 5-5 Sample Manager in Tour Code

```
        try
        {
            MyNewManagerIfc myManager = (MyNewManagerIfc)bus.getManager("MyNewManager");
            myManager.doSomeClientWork("From site code ");
        catch (Exception e)
        {
            logger.info(bus.getServiceName(), e.toString());
        }
```

Manager

This is a minimal Manager class to illustrate how to create a new Manager. A new Manager interface also needs to be created for this class. Note that this class references the sample MyNewTechnician class shown in a later code sample.

Code Sample 5-6 Sample Manager Class

```
package com.extendyourstore.foundation.manager.mynew;

import com.extendyourstore.foundation.manager.log.LogMessageConstants;
import com.extendyourstore.foundation.tour.manager.Manager;
import com.extendyourstore.foundation.tour.manager.ValetIfc;

public class MyNewManager extends Manager implements MyNewManagerIfc
{
    //---------------------------------------------------------------------------
    /**
       Constructs MyNewManager object, establishes the manager's address, and
       identifies the associated technician.
    */
    //---------------------------------------------------------------------------

    public MyNewManager()
    {
        getAddressDispatcherOptional();
        setTechnicianName("MyNewTechnician");
    }

    //---------------------------------------------------------------------------
    /**
        This method processes the input argument (via its technician).
        @param  input a String to illustrate argument passing.
        @return  a transformed String
    **/
    //---------------------------------------------------------------------------

    public String doSomeClientWork(String input)
    {
        String result = null;
        ValetIfc valet = new MyNewValet(input);
        try
        {
            result = (String)sendValetWithRetry(valet);
        }
        catch (Exception e) // usually ValetException or CommException
        {
            logger.error(LogMessageConstants.SCOPE_SYSTEM,
                         "MyNewManager.doSomeClientWork, " +
                         "could not sendValetWithRetry: Exception = {0}", e);
        }
        logger.debug(LogMessageConstants.SCOPE_SYSTEM,
                     "MyNewManager.doSomeClientWork, returns {0}", result);
        return result;
    }
}
```

Valet

The following code defines a valet to send input to MyNewTechnician.

Code Sample 5-7 Sample Valet Class

```
package com.extendyourstore.foundation.manager.mynew;

import com.extendyourstore.foundation.tour.manager.ValetIfc;
import java.io.Serializable;

public class MyNewValet implements ValetIfc
{
    /** An input used by the Technician.  **/
    protected String input = null;
    //---------------------------------------------------------------------------
    /**
        The constructor stores the input for later use by the Technician.
        @param input the input to be stored.
    **/
    //---------------------------------------------------------------------------

    public MyNewValet(String input)
    {
        this.input = input;
    }

    //---------------------------------------------------------------------------
    /**
        This method causes the MyNewTechnician to "doSomething" with the input
            and returns the results.
        @param  techIn the technician that will do the work
        @return the results of "MyNewTechnician.doSomething"
    **/
    //---------------------------------------------------------------------------

    public Serializable execute(Object techIn) throws Exception
    {
        if (!(techIn instanceof MyNewTechnician))
        {
            throw new Exception("MyNewTechnician must passed into execute.");
        }
        MyNewTechnician tech = (MyNewTechnician)techIn;
        String result = tech.doSomething(input);
        return result;
    }
}
```

Technician

The following code provides an example of a minimal Technician class. A new Technician interface also needs to be created for this class.

Code Sample 5-8 Sample Technician Class

```
package com.extendyourstore.foundation.manager.mynew;

import com.extendyourstore.foundation.manager.log.LogMessageConstants;
import com.extendyourstore.foundation.tour.manager.Technician;
import com.extendyourstore.foundation.tour.manager.ValetIfc;

public class MyNewTechnician extends Technician implements MyNewTechnicianIfc
{
    /** A value obtained from the config script.  **/
    protected String techField = null;

    public void setTechField(String value)
    {
        techField = value;
    }

    public void setConfigScript(String value)
    {
        // Complicated configuration could go here
    }

    //--------------------------------------------------------------------------
    /**
        This method processes the input argument (via its Technician).
        @param  input a String to illustrate argument passing.
        @return  a transformed String
    **/
    //--------------------------------------------------------------------------

    public String doSomething(String input)
    {
        String result = null;
        result = "MyNewTechnician processed " + input + " using " + techField;
        logger.debug(LogMessageConstants.SCOPE_SYSTEM,
                     "MyNewTechnician.doSomething, returns {0}", result);
        return result;
    }
}
```

Manager/Technician Reference

The following sections describe a Manager/Technician pair, important methods on the Manager, and an example of using the Manager in the application code.

Parameter Manager/Technician

The Parameter Manager is the interface that allows parameters to be used for customization and runtime configuration of applications. The following code from `config\conduit\ClientConduit.xml` specifies the location and properties of the Parameter Manager and Technician. Note that the Parameter Manager is a

Session Manager because it is defined with a PROPERTY element inside the APPLICATION tag. This means it can only be accessed via a tour bus.

Code Sample 5-9 ClientConduit.xml: Code to Configure Parameter Manager

```
        <APPLICATION name="APPLICATION"
                class="TierTechnician"
                package="com.extendyourstore.foundation.manager.tier"
                startservice="classpath://com/extendyourstore/pos/services/main/main.xml">
              <PROPERTY propname="managerData"
propvalue="name=ParameterManager,managerpropname=className,managerpropvalue=com.extendyourstore.foundat
ion.manager.parameter.ParameterManager"/>
              <PROPERTY propname="managerData"
                 propvalue="name=ParameterManager,managerpropname=useDefaults,managerpropvalue=Y"/>
        ...
        </APPLICATION>
```

Code Sample 5-10 ClientConduit.xml: Code to Configure Parameter Technician

```
        <TECHNICIAN name="ParameterTechnician" class = "ParameterTechnician"
              package = "com.extendyourstore.foundation.manager.parameter"
              export = "Y" >
              <PROPERTY propname="paramScript"
                     propvalue="classpath://config/manager/PosParameterTechnician.xml"/>
        </TECHNICIAN>
```

The Parameter Manager classes contain methods to retrieve parameter values. The following table lists the important ParameterManagerIfc methods, implemented in `src\com\extendyourstore\foundation\manager\parameter\ParameterManager.java`. The Customization chapter describes details about where and how parameters are defined. A list of parameters can be found in the Parameter Names and Values Addendum.

Table 5-6 Important ParameterManagerIfc Methods

Method	Description
Serializable[] getParameterValues(String paramName)	Returns the values of the specified parameter
String[] getStringValues(String parameterName)	Returns as an array of Strings the values of the specified parameter
String getStringValue(String parameterName)	Returns as a String the value of the specified parameter
Integer getIntegerValue(String parameterName)	Returns as an Integer the value of the specified parameter
Double getDoubleValue(String parameterName)	Returns as a Double the value of the specified parameter

The following code sample from `src\com\extendyourstore\pos\services\browser\BrowserControlSite.java` illustrates the use of the Parameter Manager to retrieve parameter values.

Code Sample 5-11 BrowserControlSite.java: Tour Code Using ParameterManagerIfc

```
        ParameterManagerIfc pm = (ParameterManagerIfc)bus.getManager(ParameterManagerIfc.TYPE);
        Serializable homeUrl[] = pm.getParameterValues("BrowserHomeUrl");
        String cookieString = pm.getStringValue("CookiesEnabled");
```

UI Manager/Technician

The UI Manager/Technician is used to abstract the UI implementation. User events captured by the screen are sent to the UI Manager. The UI Manager communicates with a UI Technician, which updates the screen for a client running the UI. The UI Technician provides access to the application UI Subsystem. There is one UITechnician per application.

The model is an object that is used to transport information between the screen and the UI Manager via the UI Technician. Models and screens have a one-to-one relationship. The UI Manager allows you to set the model for a screen and retrieve a model for a screen; it knows which screen to show and which model is associated with the screen. The model has data members that map to the entry fields on the given screen. It can also contain data that dictates screen behavior, such as the field that has the starting focus or the state of a specific field.

The following code samples from `config\conduit\ClientConduit.xml` specify the UI Manager and Technician properties. Like the Parameter Manager, the UI Manager can only be accessed via a tour bus.

Code Sample 5-12 ClientConduit.xml: Code to Configure UI Manager

```
       <APPLICATION name="APPLICATION"
                class="TierTechnician"
                package="com.extendyourstore.foundation.manager.tier"
                startservice="classpath://com/extendyourstore/pos/services/main/main.xml">
               <PROPERTY propname="managerData"
propvalue="name=UIManager,managerpropname=className,managerpropvalue=com.extendyourstore.pos.ui.POSUIMa
nager"/>
       ...configuration of other Managers...
       </APPLICATION>
```

Code Sample 5-13 ClientConduit.xml: Code to Configure UI Technician

```
<TECHNICIAN
        name="UITechnician"
        class="UITechnician"
        package="com.extendyourstore.foundation.manager.gui" export="Y">

        <CLASS
            name="UISubsystem"
            package="com.extendyourstore.pos.ui"
            class="POSJFCUISubsystem">

            <CLASSPROPERTY
                    propname="configFilename"
                    propvalue="classpath://com/extendyourstore/pos/config/defaults/defaultuicfg.xml"
                    proptype="STRING"/>
       ...
</TECHNICIAN>
```

The UI is configured in XML scripts. Each tour has its own uicfg file in which screen specifications are defined. The screen constants that bind to screen specification names are defined in `src\com\extendyourstore\pos\ui\POSUIManagerIfc.java`. The UI Framework chapter discusses screen configuration in more detail.

POSUIManager is the UI Manager for the Point-of-Sale application. One is started for each tour that is created. The following table lists important POSUIManagerIfc methods, implemented in `src\com\extendyourstore\pos\ui\POSUIManager.java`.

Table 5-7 Important POSUIManagerIfc Methods

Method	Description
`void showScreen(String screenId, UIModelIfc beanModel)`	Displays the specified screen using the specified model
`UIModelIfc getModel(String screenId)`	Gets the model from the specified screen
`String getInput()`	Gets the contents of the most recent Response area as a string
`void setModel(String screenId, UIModelIfc beanModel)`	Sets the model for the specified screen

These methods are used in tour code to display a screen, as in the following code from `src\com\extendyourstore\pos\services\GetCheckInfoSite.java`.

Code Sample 5-14 GetCheckInfoSite.java: Tour Code Using POSUIManagerIfc

```
POSUIManagerIfc ui = (POSUIManagerIfc) bus.getManager(UIManagerIfc.TYPE);
CheckEntryBeanModel model = new CheckEntryBeanModel();
model.setCountryIndex(countryIndex);
...set additional attributes...
ui.showScreen(POSUIManagerIfc.CHECK_ENTRY, model);
```

Journal Manager/Technician

The Journal Manager provides location abstraction for journal facilities by implementing the JournalManagerIfc interface. By communicating with a JournalTechnicianIfc, the Journal Manager removes your need to know the location of resources. The Journal Technician is responsible for providing journal facilities to other tiers. The Journal Manager must be started on each tier that uses it. There must be a LocalJournalTechnician running on the local tier or an exported JournalTechnician running on a remote tier, or both. Transactions should be written to E-journal only when completed.

The following code samples from `config\conduit\CollapsedConduitFF.xml` specify the Journal Manager and Technician properties. Note that this Manager is a Session Manager; it is defined outside of the APPLICATION element in which the UI Manager and Parameter Manager were defined. This allows the Journal Manager to be accessed outside of the bus, meaning it is more accessible and flexible.

Code Sample 5-15 CollapsedConduitFF.xml: Code to Configure Journal Manager

```
<MANAGER name="JournalManager"
     class="JournalManager"
     package="com.extendyourstore.foundation.manager.journal"
     export="N">
</MANAGER>
```

Code Sample 5-16 CollapsedConduitFF.xml: Code to Configure Journal Technician

```
<TECHNICIAN name="LocalJournalTechnician"
            class="JournalTechnician"
            package="com.extendyourstore.foundation.manager.journal"
            export="Y">
</TECHNICIAN>
```

The Journal Manager must be started on each tier that uses it. The Journal Manager sends journal entries in the following order: (1) Console if consolePrintable is set, (2) LocalJournalTechnician if defined, (3) JournalTechnician if defined. The following table lists important JournalManagerIfc methods, implemented in `src\com\extendyourstore\foundation\manager\journal\JournalManager.java`.

Table 5-8 Important JournalManagerIfc Methods

Method	Description
`void journal(String user, String transaction, String text)`	Adds a new entry to the journal
`void setConsolePrintable(String printable)`	Sets whether journal entries are sent to the console
`void index(String transaction, String key)`	Adds a new entry to the index to provide search capabilities to the transaction
`void setRegisterID(String registerID)`	Sets a register ID associated with the journal entry

These methods are used in tour code to configure the E-journal. This code is from `src\com\extendyourstore\pos\services\GetCheckInfoSite.java`.

Code Sample 5-17 GetCheckInfoSite.java: Tour Code Using JournalManagerIfc

```
        JournalManagerIfc journal =
                (JournalManagerIfc) Gateway.getDispatcher().getManager(JournalManagerIfc.TYPE);
        journal.journal(trans.getCashier().getLoginID(),
                                  trans.getTransactionID(),
                                  purchaseOrder.toJournalString());
```

CHAPTER 6

RETAIL DOMAIN

This chapter contains an overview of the 360Commerce business objects, including steps to create, extend, and use them. The Retail Domain is the set of classes that represent the business objects used by Point-of-Sale, which are contained in the Commerce Services layer of the architecture. Typical domain classes are Customer, Transaction, and Tender.

Overview

The Retail Domain is a set of business logic components that implement retail-oriented business functionality in Point-of-Sale. The Retail Domain is the part of the Commerce Services layer of the 360Commerce architecture that is retail-specific. The Retail Domain provides a common vocabulary that enables the expression of retail functionality as processes that can be executed by the 360Platform engine.

The Retail Domain is a set of retail-oriented objects that have a set of attributes. They do not implement work flow or a user interface. The Tour scripts executed by 360Platform provide the work flow, and the UI subsystem provides the user interface. The Retail Domain objects simply define the attributes and logic for application data.

A significant advantage of Retail Domain objects is that they can be easily used as-is or can be extended to include attributes and logic that are specific to a retailer's business requirements. The Domain objects could be used as a basis for many different types of retail applications. The objects serve as containers for the transient data used by the applications. Domain objects do not persist themselves, but they are persisted via the 360Store Data Manager interface.

Retail Domain is packaged as `domain360.jar`, `domain360res.jar`, `and domain360config.jar`, which are installed with the Point-of-Sale application. The Data Managers and Technicians, along with the related Data Transactions and Data Operations classes that they require, are also packaged within the Retail Domain jars.

All Retail Domain classes extend EYSDomainIfc. This interface ensures the following interfaces are implemented:

- Serializable—This communicates Java's ability to "flatten" an object to a data stream and, conversely, reconstruct the object from a data stream, when using RMI.
- Cloneable—This communicates that it is legal to make a field-for-field copy of instances of this class.

The EYSDomainIfc interface also requires that the following methods be implemented:

- `equals()`—This method accepts an object as a parameter. If the object passed has data attributes equal to this object, the method returns true, otherwise it returns false.
- `clone()`—This method creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object.
- `toString()`—This method returns a String version of the object contents for debugging and logging purposes.

New Domain Object

When an existing Retail Domain object contains attributes and methods that are a subset of those required, a new Retail Domain object can extend the existing object. For example, if a new Domain object is necessary for the Tender service, the AbstractTenderLineItem class can be extended. This class implements `TenderLineItemIfc`, which extends the generic EYSDomainIfc interface. If no similar Domain object exists in the application, create a new Domain object. The usual coding standards apply; reference the Development Standards document.

1. Create a new interface extending EYSDomainIfc.

   All Retail Domain objects extend EYSDomainIfc, but existing Services have an interface available for Domain objects related to that Service. For example, TenderLineItemIfc, which extends EYSDomainIfc, is the interface implemented by each Retail Domain object interface in the Tender service. The following code sample shows the header of TenderPurchaseOrderIfc, found in `src\com\extendyourstore\domain\tender\TenderPurchaseOrderIfc.java`.

   Code Sample 6-1 TenderPurchaseOrderIfc.java: Class Header
   ```
   public interface TenderPurchaseOrderIfc extends TenderLineItemIfc
   {
           public static final String revisionNumber = "$Revision:   1.0  $";
           // begin TenderPurchaseOrderIfc
   }
   ```

2. Create a new Java class that implements the interface created in the previous step. The class of a brand new object that does not fit an existing pattern should extend AbstractRoutable, which defines a “luggage tag” for EYS domain classes; otherwise, the class should extend the existing class that represents a similar type of object.

   The following code sample shows the header for the TenderPurchaseOrder Domain object from `src\com\extendyourstore\domain\tender\TenderPurchaseOrder.java`.

   Code Sample 6-2 TenderPurchaseOrder.java: Class Header
   ```
   public class TenderPurchaseOrder extends AbstractTenderLineItem implements TenderPurchaseOrderIfc
   {
           public static final String revisionNumber = "$Revision:   1.0  $";
           //begin TenderPurchaseOrder
   }
   ```

   In the implementation of the class, make sure to do the following:

   - Define attributes for the class

Check the superclass to see if an attribute has already been defined. For example, the AbstractTenderLineItem class defines the amountTender attribute, so amountTender should not be redefined in a new Tender Domain object.

If the new domain object has numerous constants, you might consider defining ObjectNameConstantsIfc.java

- Define get and set methods for the attributes as necessary
- Implement methods required by EYSDomainIfc: `equals()`, `clone()`, `toString()`, and `getRevisionNumber()`. Reference the superclass as appropriate. `toString()` should indicate the class name and revision number.

3. To return a new instance of the Domain object, add a method to `src\com\extendyourstore\domain\factory\DomainObjectFactoryIfc.java` called `get`*`ObjectName`*`Instance()`.

   Domain objects should **always** be instantiated by the factory. The following code sample shows the method interface to return an instance of the TenderPurchaseOrder object.

   Code Sample 6-3 DomainObjectFactoryIfc.java: Method For Instantiating TenderPurchaseOrder

```
public TenderPurchaseOrderIfc getTenderPurchaseOrderInstance();
```

4. To return a new instance of the Domain object, implement the method `src\com\extendyourstore\domain\factory\DomainObjectFactory.java` called `get`*`ObjectName`*`Instance()`.

   The following code sample shows the method definition to return an instance of the TenderPurchaseOrder object.

   Code Sample 6-4 DomainObjectFactory.java: Method For Instantiating TenderPurchaseOrder

```
public TenderPurchaseOrderIfc getTenderPurchaseOrderInstance()
{
        // begin getTenderPurchaseOrderInstance()
        return(new TenderPurchaseOrder());
}
```

Domain Object in Tour Code

Once a Retail Domain class is identified for use, the Java code needs to be written to instantiate the object and call the object's methods. This code is typically located in site, road and aisle classes of application tours. There are two very important things to keep in mind when using Domain objects in Tour code:

- **Retail Domain objects *cannot* be instantiated directly. They must be generated by the factory.**
- **All interaction with Domain objects take place through the object's interface, even interaction between objects.**

The steps to use the object involve the following.

1. Get an instance of the DomainObjectFactory and request the instance of the object from the factory.

   The factory class is instantiated once for the application and returns instances of Retail Domain objects. Since different implementations use different classes to implement the objects, the factory keeps track of which class implements the requested object.

The following line of code from `src\com\extendyourstore\services\tender\GetCheckInfoSite.java` gets an instance of a Check object.

Code Sample 6-5 GetCheckInfoSite.java: Instantiating Check from DomainObjectFactory

```
check = DomainGateway.getFactory().getTenderCheckInstance();
```

2. Call methods on the object.

   Now that an instance of the object exists, methods of the class can be called. The following lines of code from `GetCheckInfoSite.java` sets attributes on the Check object.

Code Sample 6-6 GetCheckInfoSite.java: Setting Attributes of Check

```
check.setTenderLimits(cargo.getTenderLimits());
check.setAmountTender(amount);
```

Domain Object Reference

The Domain Objects discussed below include a description of the purpose of the object, classes and interfaces involved in its construction, a class diagram, and examples in Tour code.

CodeListMap

To implement Point-of-Sale metadata such as reasons for return, shipping methods, and departments, the CodeList objects are used. This data is referred to as “reason codes” from the UI. Codes are read in from the database at application startup. They are available from the Utility Manager. The following files are involved in the formation of CodeLists. All are found in `src\com\extendyourstore\domain\utility`.

Table 6-1 CodeListMap Object Classes and Interfaces

Class or Interface	Description	Important Methods

CodeEntry	This class handles the functions associated with an entry in a list of codes.	`void setText(String)`
		`void setCode(int)`
		`void setEnabled(boolean)`
		`String getCodeString()`

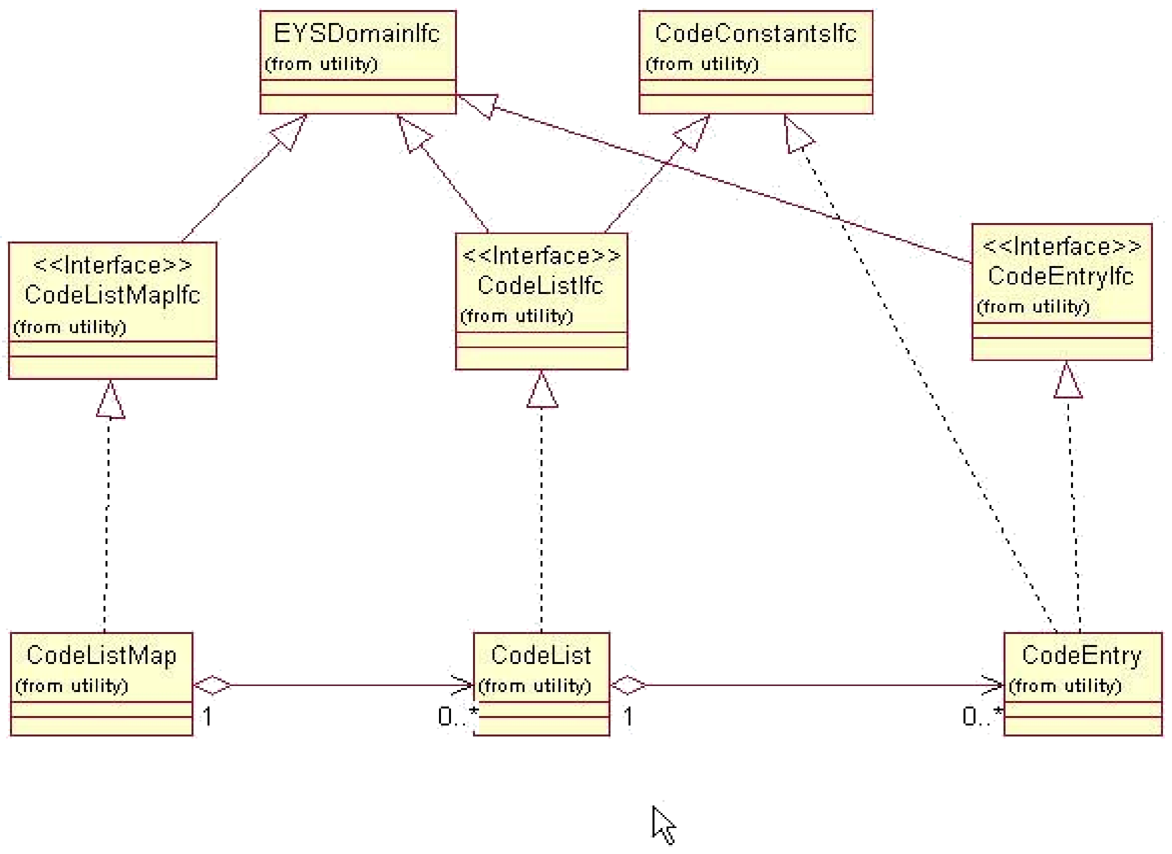
CodeList	This class is used for handling lists of codes which map to strings, such as reason codes.	`CodeEntryIfc[] getEntries()`
		`void setEntries(CodeEntryIfc[])`
		`void addEntry(CodeEntryIfc)`
		`CodeEntryIfc findListEntry(String)`

CodeListMap	This class is used for the collection of code lists used in applications.	`CodeListIfc[] getLists()`
		`CodeListIfc getCodeListInstance(String)`
		`CodeListIfc add(CodeListIfc)`

CodeConstantsIfc	This class defines constants used for the implementation of CodeList and CodeEntry. It includes the constants for the lists currently defined, such as TimekeepingManagerEditReasonCodes and TillPayOutReasonCodes.	This class does not contain methods.

The following class diagram illustrates the relationship between these classes.

Figure 6-1 CodeListMap Class Diagrams



To use the CodeListMap, the Utility Manager provides two methods:

- `CodeListMapIfc getCodeListMap()`
- `void setCodeListMap(CodeListMapIfc)`

Tour code that requires a code entry would retrieve it as in the following code from `src\com\extendyourstore\pos\services\common\ItemInfoEnteredAisle.java`.

Code Sample 6-7 ItemInfoEnteredAisle.java: CodeListIfc in Tour Code

```
        CodeListIfc list  = utility.getCodeListMap().get(CodeConstantsIfc.CODE_LIST_UNIT_OF_MEASURE);
        CodeEntryIfc uomCodeEntry = list.findListEntry(uomString);
        String uomCode = uomCodeEntry.getCodeString();
```

Currency

All currency representation and behavior is abstracted, so any currency can be implemented. Currency is a Domain Object that handles the behaviors and attributes of money used as a medium of exchange. It is important to use Currency objects and methods to compare and manipulate numbers instead of primitive types. Currency is implemented by the following classes. They can be found in `src\com\extendyourstore\domain\currency`.

Table 6-2 Currency Object Classes and Interfaces

Class or Interface	Description	Important Methods

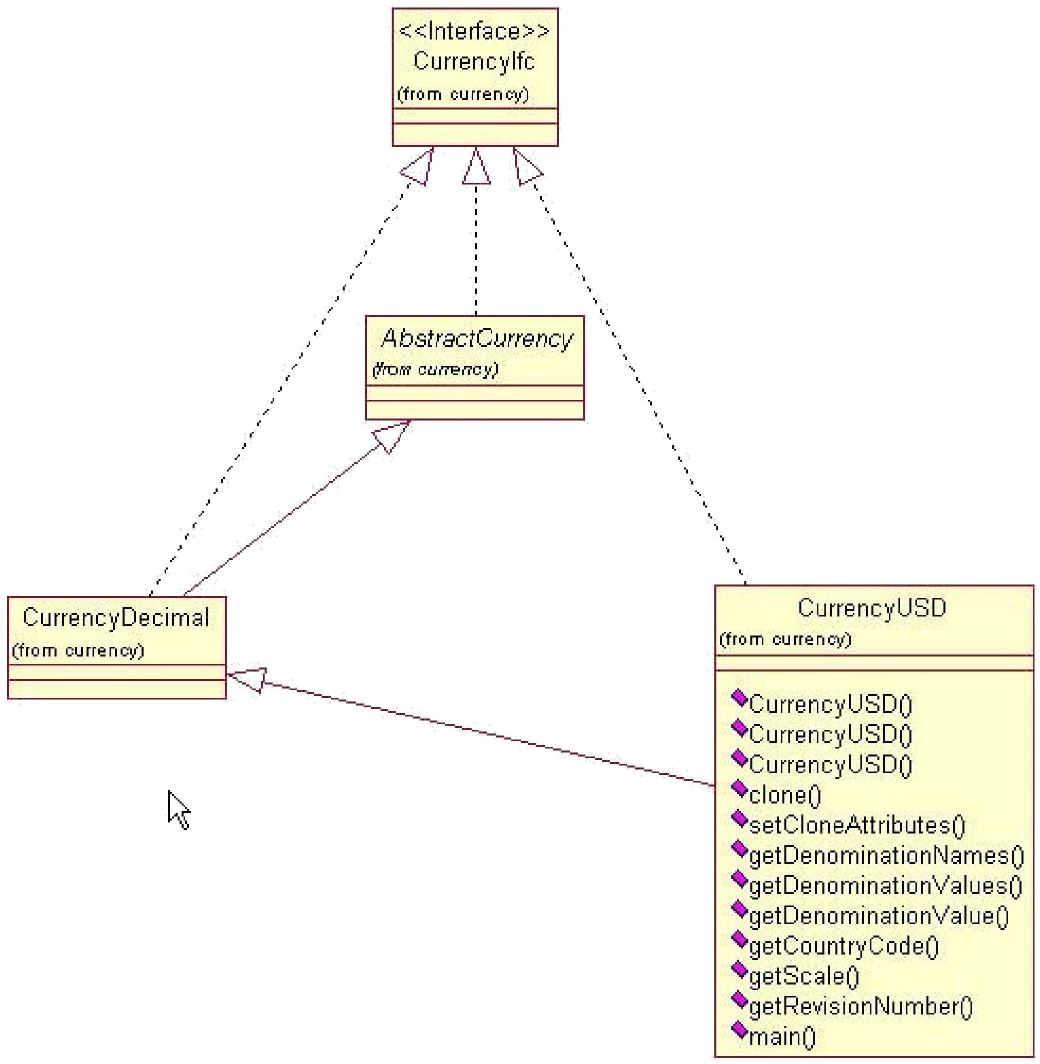
CurrencyIfc	This interface defines a common interface for currency objects.	`CurrencyIfc add(CurrencyIfc)`
		`CurrencyIfc negate()`
		`String getCountryCode()`

AbstractCurrency	This abstract class contains the behaviors and attributes common to all currency.	`BigDecimal getBaseConversionRate()`
		`void setNationality(String)`
		`String getNationality()`

CurrencyDecimal	This class contains the behaviors and attributes common to all decimal-based currency.	`CurrencyIfc add(CurrencyIfc)`
		`CurrencyIfc negate()`
		`String getCountryCode()`

All Currency types extend AbstractCurrency and implement CurrencyIfc. For example, if creating a class to support Canadian currency, the class should extend CurrencyDecimal and implement CurrencyIfc.

Figure 6-2 Currency Class Diagram



The following code is an example of the Currency object used in `src\com\extendryourstore\pos\services\tender\PurchaseOrderAmountEnteredAisle.java`.

Code Sample 6-8 PurchaseOrderAmountEnteredAisle.java: CurrencyIfc in Tour Code

```
        CurrencyIfc balanceDue = totals.getBalanceDue();
        CurrencyIfc amount = DomainGateway.getBaseCurrencyInstance(poAmount);
         if (!(amount.compareTo(balanceDue) == CurrencyIfc.EQUALS)) {
        ...display invalid PO Amount message...
        }
```

Transaction

A Transaction is a record of business activity that involves a financial and/or merchandise unit exchange or the granting of access to conduct business with an external device. There are various types of Transactions found in `src\com\extendyourstore\domain\transaction` such as LayawayTransaction, StoreOpenCloseTransaction, and BankDepositTransaction. SaleReturnTransaction is a commonly used

Domain Object that extends AbstractTenderableTransaction. The classes involved in the implementation of a SaleReturnTransaction and its behaviors are described in the following table.

Table 6-3 Transaction Object Classes and Interfaces

Class or Interface	Description	Important Methods

SaleReturnTransaction	This class is a sale or return transaction.	`void addTender(TenderLineItemIfc)`
		`CustomerIfc getCustomer()`
		`TransactionTotalsIfc getTenderTransactionTotals()`

AbstractTenderableTransaction	This class contains the behavior associated with a transaction that involves the tendering of money.	`void addLineItem(SaleReturnLineItemIfc)`
		`void linkCustomer(CustomerIfc)`
		`void addLineItem(AbstractTransactionLineItemIfc)`

Transaction	This class represents a record of business activity that involves a financial and/or merchandise unit exchange or the granting of access to conduct business at a specific device, at a specific point in time for a specific employee.	`CustomerInfoIfc getCustomerInfo()`
		`String getTillID()`
		`void setCashier(EmployeeIfc)`

TenderableTransactionIfc	This is the interface for all transactions that involve the tendering of money.	`void addTender(TenderLineItemIfc)`
		`TenderLineItemIfc[] getTenderLineItems()`
		`void setTransactionTotals(TransactionTotalsIfc)`

SaleReturnTransactionIfc	This is the interface for all sale/ return transactions.	`void addTender(TenderLineItemIfc)`
		`CustomerIfc getCustomer()`
		`TransactionTotalsIfc getTenderTransactionTotals()`

RetailTransactionIfc	This is the interface for all retail transactions.	`EmployeeIfc getSalesAssociate()`
		`AbstractTransactionLineItemIfc[] getLineItems()`
		`String getOrderID()`

UpdatableInventoryIfc	This interface defines a transaction for which inventory can be updated.	`InventoryTransactionIfc getInventoryUpdate()`

The following code sample from `src\com\extendyourstore\domain\arts\JdbcSaveTenderLineItems.java` shows how SaleReturnTransaction is used in Tour code.

Code Sample 6-9 JdbcSaveTenderLineItems.java: SaleReturnTransactionIfc in Tour Code

```
public void saveTenderLineItems(JdbcDataConnection dataConnection,
                                     TenderableTransactionIfc transaction) throws DataException
{
        if (transaction instanceof SaleReturnTransactionIfc)
        {
               SaleReturnTransactionIfc srt = (SaleReturnTransactionIfc) transaction;
               int numDiscounts = 0;
               if (srt.getTransactionDiscounts() != null)
               {
                       numDiscounts = srt.getTransactionDiscounts().length;
               }
               lineItemSequenceNumber = srt.getLineItems().length + 1 + numDiscounts;
        }
...code to handle different transaction types...
}
```

CHAPTER 7

CUSTOMIZATION

This chapter covers additional customization options. Frequently, it is necessary to customize Point-of-Sale to integrate with existing systems and environments.

Parameters

Parameters are used to control flow, set minimums and maximums for data, and allow flexibility without recompiling code. A user can modify parameter values from the UI without changing code. Parameter values can be modified by Point-of-Sale, and the changes can be distributed by other 360Commerce applications. For example, the maximum cash refund allowed and the credit card types accepted are parameters that can be defined by Point-of-Sale. To configure parameters, you need to understand the parameter hierarchy, define the group that the parameter belongs to, and define the parameter and its properties.

Parameter Hierarchy

Parameters are defined in XML files that are organized in a hierarchy. Different XML files represent different levels in a retail setting at which parameters may be defined. Understanding the parameter hierarchy helps you define parameters at the appropriate level. The following table lists the parameter directories, XML filenames, and file descriptions.

Table 7-1 Parameter Directories, Files, and Descriptions

Directory	Parameter-Related XML File	Description
application	application.xml	default parameter information provided by the base product
corporate	corporate.xml	company information
store	store.xml	local store information
register	workstation.xml	register-level information
userrole	operator.xml	user-level information

Higher-level parameters by default are overridden by lower-level parameter settings. For example, store-level configuration parameters override application-level parameters. The FINAL element in a parameter definition signifies whether the parameter can be overridden. Below is an excerpt from `config\manager\PosParameterTechnican.xml`, showing the order of precedence from highest level to lowest level.

Code Sample 7-1 Default Parameter Settings

```
<SELECTOR name="defaultParameters">
        <SOURCE categoryname="application" alternativename="application">
        <SOURCE categoryname="corporate" alternativename="corporate">
       <SOURCE categoryname="store" alternativename="store">
       <SOURCE categoryname="service" alternativename="NO_OP">
       <SOURCE categoryname="uidata" alternativename="NO_OP">
       <SOURCE categoryname="register" alternativename="workstation" >
        <SOURCE categoryname="userrole" alternativename="operator" >
    <SELECTOR
```

The `categoryname` specifies the directory name and the `alternativename` specifies the name of the XML file. All parameter subdirectories reside in `config\parameter`.

Parameter Group

Each parameter belongs to a group, which is a collection of related parameters. The groups are used when modifying parameters within the UI. The user selects the group first, then has the option to modify the related parameters that belong to that group. Examples of groups are Browser, Customer, Discount, and Employee.

Adding a parameter requires adding it to the proper group. The following excerpt from application.xml shows the Tender group and a parameter definition inside the group. The “hidden” attribute indicates whether or not the group is displayed in the UI.

Code Sample 7-2 Definition of Tender Group

```
<GROUP name="Tender"
            hidden="N">
       <PARAMETER name="MaximumCashChange"
          ...
       <PARAMETER>
...
<GROUP>
```

Parameter Properties

Each parameter file contains parameter definitions organized by group. The following shows an example of two parameter definitions from `config/parameters/application/application.xml`.

Table 7-2 Parameter Definitions From application.xml

```
<PARAMETER name="CashAccepted"
         type="LIST"
         default="USD"
         final="N"
         hidden="N">
         <VALIDATOR class="EnumeratedListValidator"
            package="com.extendyourstore.foundation.manager.parameter">
            <!-- Use ISO 3 letter currency code -->
            <PROPERTY propname="member" propvalue="None" />
            <PROPERTY propname="member" propvalue="USD" />
            <PROPERTY propname="member" propvalue="CAD" />
         </VALIDATOR>
         <VALUE value="USD"/>
         <VALUE value="CAD"/>

<PARAMETER name="StoreCreditRefundByAmount"
         type="CURRENCY"
         final="N"
         hidden="N">
         <VALIDATOR class="FloatRangeValidator"
            package="com.extendyourstore.foundation.manager.parameter">
            <PROPERTY propname="minimum" propvalue="0.00" />
            <PROPERTY propname="maximum" propvalue="99999.99" />
         </VALIDATOR>
         <VALUE value="25.00"/>
      </PARAMETER>
```

The FINAL attribute indicates whether the property definition is final, meaning it cannot be overridden by lower-level parameter file settings. The VALUE element is the current setting of the parameter. If multiple values are set, that means the value of the parameter is a list of values. The three types of VALIDATOR classes are listed in the following table.

Table 7-1 Validator Types

Validator	Description
EnumeratedListValidator	Determines whether a value supplied is one of an allowable set of values
FloatRangeValidator	Ensures that the value lies within the specified minimum and maximum float range
IntegerRangeValidator	Ensures that the value of a parameter lies within the specified minimum and maximum integer range

Devices

POS devices are configured with the `posdevices.xml` file, device-specific property files, and other JavaPOS configuration files. The device vendor typically provides a JavaPOS configuration file to support the JavaPOS standards. If necessary, you can create your own configuration file to meet your device requirements. Interaction of the Point-of-Sale application with devices is managed by the Device Manager and Device Technician.

Set Up the Device

To configure a device to work with Point-of-Sale, first consult the user manual for that device for specific setup requirements. Set up the device drivers and configuration file so the device is available to applications.

Test the Device

Use the POStest application available internally or at http:www.javapos.com to determine if a device adheres to existing JavaPOS standards. POStest is a GUI-based utility for exercising POS devices using

JavaPOS. Currently it supports the following devices: POSPrinter, MICR, MSR, Scanner, Cash Drawer, Line Display, Signature Capture, and PIN Pad. Perform the following steps to use POStest. See http:www.javapos.com for more details.

1. Configure the classpath for JavaPOS. This means that the classpath should include the location of `POStest, jpos.jar, jcl.jar` and the JavaPOS services for the devices.
2. To build POStest, compile the classes in <location of POStest>\upos\com\jpos\POStest.
3. To run POStest, enter the following at a command line:
   `java com.jpos.POStest.POStest`

Sometimes, the hardware vendor provides test utilities that come with the JavaPOS implementation. You should test with these tools as well.

Create a Session and ActionGroup

In Point-of-Sale code, devices require a Session and an ActionGroup. If you need to interact with a new JavaPOS device, you must create a new Session and ActionGroup.

Sessions capture input for the application. In UI scripts, device connections are defined that allow the application code to receive input from a device by connecting the Session with the screen specification. The Session listens to JavaPOS controls on the device.

ActionGroups provide the commands that can be used to control the device. ActionGroups are instantiated by Tour code. When a method on an ActionGroup is called in Tour code, the DeviceTechnician talks to JavaPOS controls on the device.

To create or modify a Session and ActionGroup, perform the following steps.

1. Configure the Session and ActionGroup in `config\pos\posdevices.xml`.

   To do this, enter the name of the Session and ActionGroup in `posdevices.xml`. You must specify the name of the object, its class and its package. In addition, you can set some attributes available in the corresponding class in `posdevices.xml`. This file creates a hash table of ActionGroups and Sessions, which are part of the DeviceTechnician. Below is a definition of an ActionGroup and Session from posdevices.xml.

   Code Sample 7-1 ActionGroup Configuration

   ```
   <ACTIONGROUP name="LineDisplayActionGroupIfc"
               class="LineDisplayActionGroup"
               package="com.extendyourstore.pos.device"/>
   ```

   Code Sample 7-2 Session Configuration

   ```
   <SESSION name="ScannerSession"
               devicename = "defaultScanner"
               class="ScannerSession"
               package="com.extendyourstore.foundation.manager.device"
               defaultmode = "MODE_RELEASED"
               />
   ```

2. Define a Session class to get input that extends InputDeviceSession or DeviceSession.

Each type of device has a Session class defined in `src\com\extendyourstore\foundation\manager\device`. A device session like CashDrawerSession would extend DeviceSession, whereas an input device session like a ScannerSession would extend InputDeviceSession.

Sessions are not instantiated in Tour code but are accessed by UI scripts in device connections.

3. Define an ActionGroupIfc interface that extends DeviceActionGroupIfc.

   This class should also be located in `src\com\extendyourstore\pos\device`. The following line of code shows the header of the CashDrawerActionGroupIfc class.

   ```
   public interface CashDrawerActionGroupIfc extends DeviceActionGroupIfc
   ```

4. Create the ActionGroup class. This class should be located in `src\com\extendyourstore\pos\device`, and its purpose is to define specific device operations available to Point-of-Sale. The following line of code shows the header of the CashDrawerActionGroup class.

   ```
   public interface CashDrawerActionGroup extends CashDrawerActionGroupIfc
   ```

5. If one does not already exist, create a device connection in the UI Subsystem file. Device connections in the UI Subsystem files allow the application to receive input data from the Session.

   The DeviceSession class is referenced in the device connections for the relevant screen specifications. For example, the following code is an excerpt from `src\com\extendyourstore\pos\services\tender\tenderuicfg.xml`.

   Code Sample 7-3 Example of Device Connection

   ```
   <DEVICECONNECTION
               deviceSessionName="ScannerSession"
               targetBeanSpecName="PromptAndResponsePanelSpec"
               listenerPackage="java.beans"
               listenerInterfaceName="PropertyChangeListener"
               adapterPackage="com.extendyourstore.foundation.manager.gui"
               adapterClassName="InputDataAdapter"
               adapterParameter="setScannerData"
               activateMode="MODE_SINGLESCAN">
   ```

6. Access the device manager and input from the Session in the application code.

   Using the bean model, data from the Session can be accessed with methods in the device's ActionGroupIfc. Other devices such as the printer are accessed through a device manager as in the following code from `src\com\extendyourstore\pos\services\tender\CompleteTenderSite.java`.

   Code Sample 7-4 ActionGroup in Tour code

   ```
   POSDeviceActions pda = new POSDeviceActions((SessionBusIfc) bus);
   pda.clearText();
   pda.displayTextAt(1,0,displayLine2);
   ```

Simulate the Device

It is often practical to simulate devices for development purposes until the hardware is available or the software is testable. Switching to a simulated device is easily accomplished by editing `config\pos\posdevices.xml`. In fact, when you install Point-of-Sale and choose the option to run in Simulated mode, `posdevices.xml` is modified accordingly. By default, unselected devices are set up as simulated. The following code sample shows the configuration of SimulatedPrinterSession.

Code Sample 7-5 Simulated Device Configuration

```
<SESSION name="SimulatedPrinterSession"
            devicename = "defaultPrinter"
            class="SimulatedPrinterSession"
            package="com.extendyourstore.foundation.manager.device"
            defaultmode = "MODE_RELEASED"
            />
```

Help Files

The 360Store Point-of-Sale application includes help files to provide information to assist the end-user. When the user chooses Help or F1 from the global navigation panel, a help browser appears in Point-of-Sale to describe the current screen. An index is provided on the left so the user may choose additional topics to view. The help is implemented as JavaHelp and includes these components:

- One HTML help file for each screen. The product help files are Microsoft Word files saved as HTML. They can be edited with Word, an HTML editor or a text editor.
- A Table Of Contents file that defines the index that displays on the left.
- A properties file that associates overlay screen names with the corresponding HTML filenames.

Refer to http://www.java.sun.com for more information on JavaHelp.

Note: If the base product help files are modified, upgrades for help files will not be available, and you will not be able to take advantage of updates provided with future maintenance releases of the application.

Modifying Help Files

1. Locate the name of the help file associated with the overlay screen name that needs to be modified. The help file names are defined in `helpscreens.properties` located in `config\ui\help`.

   Code Sample 7-6 JavaHelp—helpscreens.properties
   ```
   REFUND_OPTIONS                        refundoptionshelp.htm
   ```
2. Locate the help file in the `locales\en_US\config\ui\help` directory. Open the file in Microsoft Word or an HTML editor and edit the content. If you are using Word to edit, be sure to save the file as HTML when the edits are complete.
3. Make identical modifications to the help file for each of the supported languages. For example, the base product also has help files in `locales\es_PR\config\ui\help` and `locales\fr_CA\config\ui\help`.
4. If the index location or text descriptions needs to be modified, change toc.xml located in `locales\en_US\config\ui\help`. The order of the items in the index is also defined by this file.

   Code Sample 7-7 JavaHelp—toc.xml
   ```
     <tocitem target="REFUND_OPTIONS"    text="Refund Options" />
   ```

CHAPTER 8

STORE DATABASE

This chapter describes the database used with Point-of-Sale and how to interface with it, including:

- Updating tables
- Rebuilding the database
- Creating new tables
- Updating flat file configurations

The chapter includes an example of writing code to store new data in the database using the Tender function.

ARTS Compliance

The 360Commerce Point-of-Sale system uses an Association of Retail Technology Standards (ARTS)-compliant database to store transactions and settings. The ARTS standard (see http://www.nrf-arts.org/) is a key element in maintaining compatibility with other hardware and software systems.

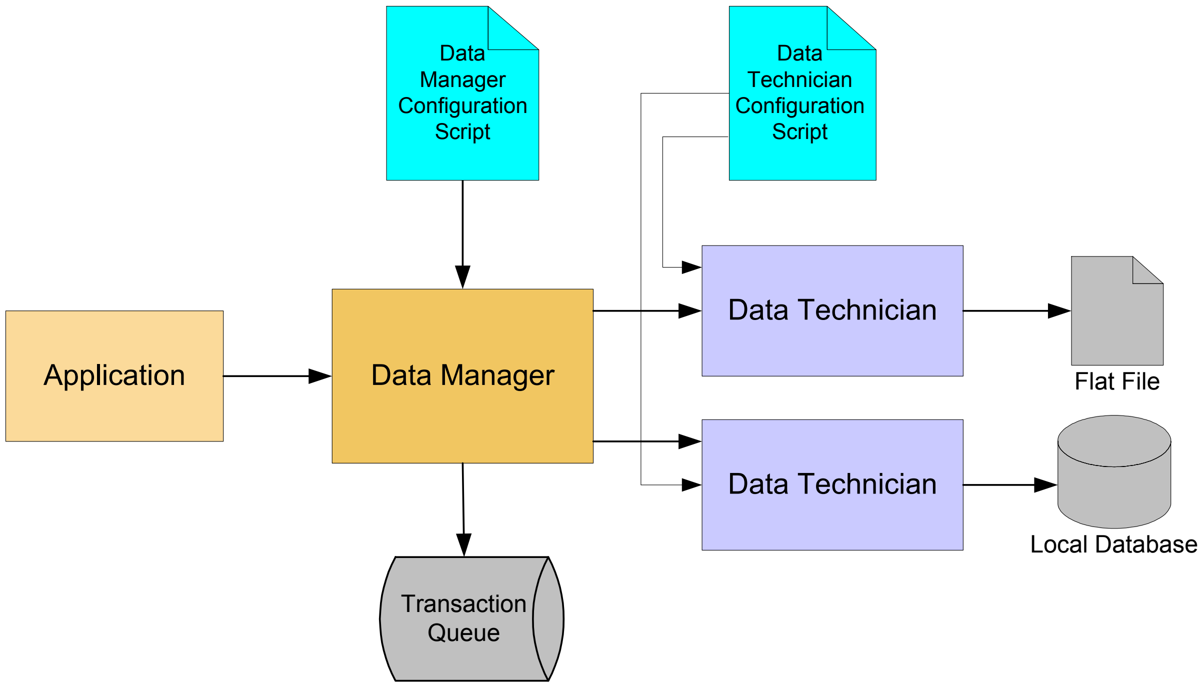
Although the Point-of-Sale system complies with the ARTS guidelines, it does not implement the entire standard, and contains some tables which are not specified by ARTS. For example, ARTS tables for store equipment and recipe are not included, while tables for tender types and reporting have been added.

The `ARTSDatabaseIfc.java` file defines the mapping of ARTS names to constants in application code.

Understanding Data Managers and Technicians

The following diagram shows how Data Managers and Data Technicians handle communication with the database in the Point-of-Sale application.

Figure 8-1 Data Managers and Data Technicians



The Point-of-Sale system uses the following components to write to the database:

- The Data Manager's primary responsibilities are to provide an API to the application code and to contact the Data Technician and pass it data store requests. Typically, there are multiple Data Manager instances (one per register).
- The Data Manager Configuration Script is an XML file that specifies the properties of the Data Manager.
- The Data Technician handles the database connection. Configure the Data Technician with an XML script. The Data Transaction class is the valet from the manager to the technician. The Data Transaction class has the add, find, and update methods to the database. Typically, there is one Data Technician that communicates with the local database and one that communicates with flat files.

  Note: Most managers create valets when they need talk to technicians. Data Manager works a little differently: the Data Transaction class calls the Data Manager and passes itself as a valet. The valet finds the data operation class, then the valet knows which technician it is associated with and calls its execute method.

- The Data Technician configuration script is an XML file that specifies the properties of the Data Technician.
- The Transaction Queue collects data transactions and guarantees delivery.
- Flat Files are local register files that are used when the register is offline.
- The Local Database is the store database.

How Data Transactions Work

This section gives an overview of how 360Platform, Data Manager, and Data Technician components work together to store data in the database.

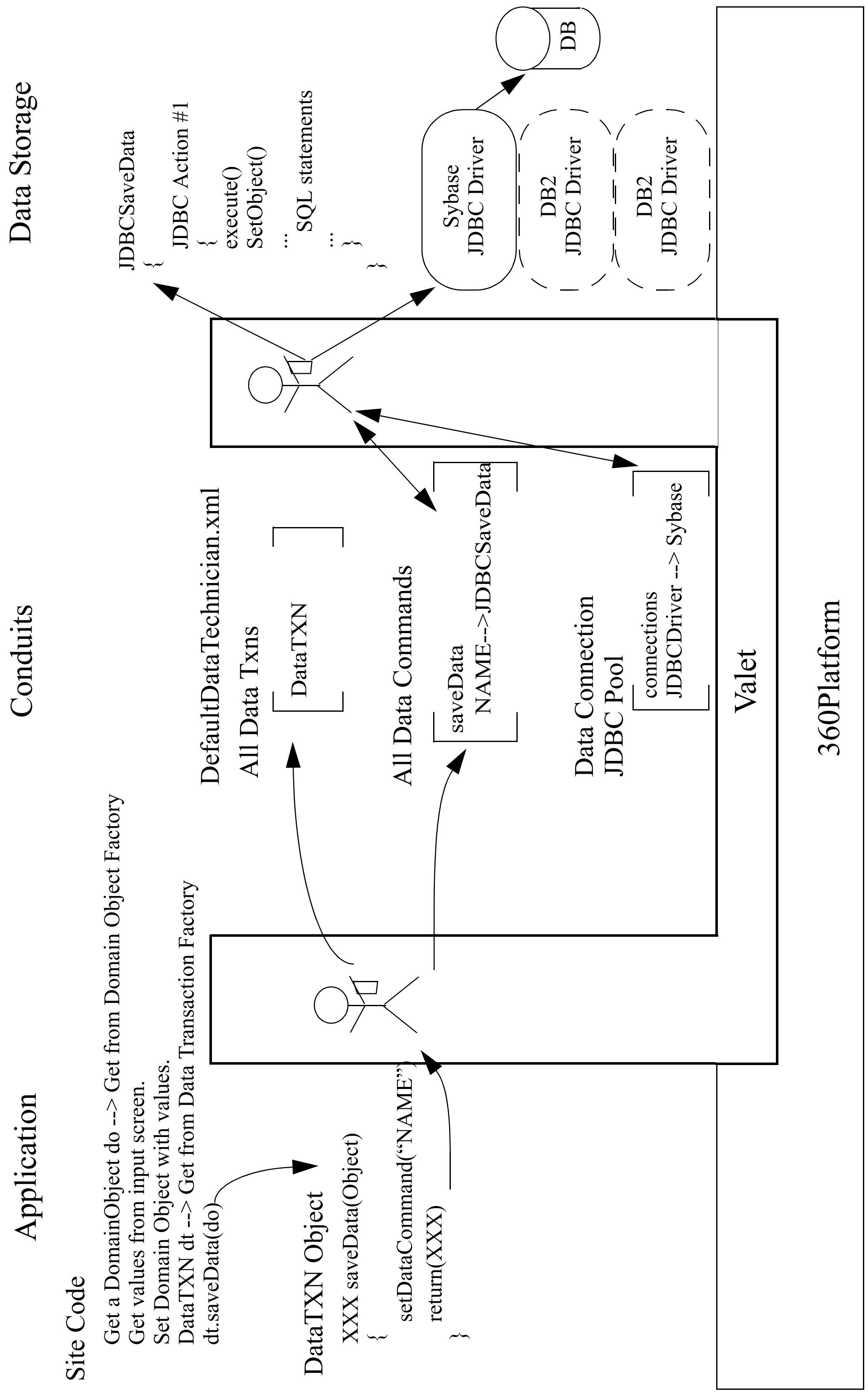
Note: The notation TXN refers to a data transaction, which can be any guaranteed transmission of data, not necessarily a sales transaction in the retail sense.

360Platform is responsible for configuring the system so that the Data Manager, Data Technician, configuration scripts, and conduit scripts work together to provide the mechanism to update, store, and retrieve data from a database.

1. The client conduit script defines the name and package for the Data Manager and Data Manager configuration script, `POSDataManager.xml`.
2. The server conduit script defines the name and package for the Data Technician and Data Technician configuration script, `DefaultDataTechnician.xml`.
3. At runtime, the tour code requests a data transaction object from the Data Transaction Factory.
4. The Data Transaction Factory verifies that the transaction is defined in `POSDataManager.xml` and the transaction object is returned to the tour code.
5. The tour code calls a method on the transaction object that creates a vector of data actions. A data action corresponds to a set of SQL commands that are executed as a unit. (Data actions are reused by different transactions.)
6. The method in the transaction object gets a handle to the Data Manager and calls `execute()`, sending itself as a parameter. This instructs the Data Manager to send the Transaction object (a valet) across the network to the Data Technician.

   Note: Most Manager/Technician pairs work differently. The standard pattern is for the tour code to get a handle to the Manager, then call a method on the manager that will create the valet object and send it to the technician. For the Data Manager/Technician pair, the transaction object (the valet class), gets the handle to the Data Manager. The tour code is only responsible for getting a transaction object from the factory and calling the appropriate method.
7. On the server side, the Data Technician configuration script, DefaultDataTechnician.xml, lists all available transactions. It also defines an operation class for each data action. Each data action is then processed by the appropriate data operation class.

Figure 8-2 Updating the Database: Simplified Runtime View



Creating or Updating Database Tables

Use this procedure when creating a new database table or updating an existing one. Refer to the ARTS standards when designing tables.

Note: When you add or change a table, you need to rebuild the database for your local copy of Point-of-Sale before you can test your changes. The Point-of-Sale system includes scripts for building the database; the main script, `dbbuild.bat`, runs multiple subordinate scripts to create all the necessary tables and populate them with initial data. The script automatically includes all files in the `sql` directory, so the build scripts do not have to be modified in order to build your files. However, you may have to edit a build script in order to test foreign key constraints; see step 6.

1. Edit the appropriate database script, or write a new one.

   Database scripts can be found in the source directory `commerceservices\trunk\db\sql`. In a Point-of-Sale installation, see `C:\360store\360common\db\sql`.

   Start a new file (or edit the appropriate existing file) in the `db/sql` source directory file to store SQL commands for creating the new table. Code Sample 8-1 shows the SQL commands for creating the table that stores the credit card data.

   Code Sample 8-1 CreateTableCreditDebitCardTenderLineItem.sql

```
DROP TABLE TR_LTM_CRDB_CRD_TN;

CREATE TABLE TR_LTM_CRDB_CRD_TN
(
    ID_STR_RT           char(5) NOT NULL,
    ID_WS               char(3) NOT NULL,
    DC_DY_BSN           char(10) NOT NULL,
    AI_TRN              integer NOT NULL,
    AI_LN_ITM           smallint NOT NULL,
    TY_TND              varchar(20),
    ID_ISSR_TND_MD      varchar(20),
    TY_CRD              VARCHAR(40),
    ...additional column lines omitted here...
);

ALTER TABLE TR_LTM_CRDB_CRD_TN ADD PRIMARY KEY (ID_STR_RT, ID_WS, DC_DY_BSN, AI_TRN,
        AI_LN_ITM);

COMMENT ON TABLE TR_LTM_CRDB_CRD_TN IS 'Credit/Debit Card Tender Line Item';

COMMENT ON COLUMN TR_LTM_CRDB_CRD_TN.ID_STR_RT IS           'Retail Store ID';
COMMENT ON COLUMN TR_LTM_CRDB_CRD_TN.ID_WS IS               'Workstation ID';
COMMENT ON COLUMN TR_LTM_CRDB_CRD_TN.DC_DY_BSN IS           'Business Day Date';
COMMENT ON COLUMN TR_LTM_CRDB_CRD_TN.AI_TRN IS              'Transaction Sequence
Number';
COMMENT ON COLUMN TR_LTM_CRDB_CRD_TN.AI_LN_ITM IS           'Retail Transaction Line
Item
                                                                    Sequence Number';
COMMENT ON COLUMN TR_LTM_CRDB_CRD_TN.ID_ISSR_TND_MD IS      'Tender Media Issuer ID';
COMMENT ON COLUMN TR_LTM_CRDB_CRD_TN.TY_TND IS TenderTypeCode';
COMMENT ON COLUMN TR_LTM_CRDB_CRD_TN.TY_CRD IS              'Card Type';
...additional comment lines omitted...
```

2. Create or edit the insert files (also in the `db/sql` source directory) for inserting initial data into the new database table.

   This step is used only to insert data into the database table for purposes of initially logging on, testing, and so on. Code Sample 8-2 contains three inserts from the `db/sql/InsertTableTenderLineItem.sql` file.

   Code Sample 8-2 InsertTableTenderLineItem.sql

```
INSERT INTO TR_LTM_TND
(ID_STR_RT, ID_WS, AI_TRN, AI_LN_ITM, DC_DY_BSN, TY_TND, MO_ITM_LN_TND,
 TS_CRT_RCRD, TS_MDF_RCRD)
VALUES ('04241', '149', 1000, 2, '1999-09-23', 'CASH', 54.11,
        TIMESTAMP('1999-09-05 12:53:06.536'), TIMESTAMP('1999-09-05 12:53:06.536'));

INSERT INTO TR_LTM_TND
(ID_STR_RT, ID_WS, AI_TRN, AI_LN_ITM, DC_DY_BSN, TY_TND, MO_ITM_LN_TND,
 TS_CRT_RCRD, TS_MDF_RCRD)
VALUES ('04241', '149', 1000, 2, '1999-09-30', 'CASH', 4.32,
        TIMESTAMP('1999-09-05 12:53:06.536'), TIMESTAMP('1999-09-05 12:53:06.536'));

INSERT INTO TR_LTM_TND
(ID_STR_RT, ID_WS, AI_TRN, AI_LN_ITM, DC_DY_BSN, TY_TND, MO_ITM_LN_TND,
 TS_CRT_RCRD, TS_MDF_RCRD)
VALUES ('04241', '129', 1, 2, '1999-09-05', 'CASH', 54.11,
        TIMESTAMP('1999-09-05 12:53:06.536'), TIMESTAMP('1999-09-05 12:53:06.536'));
```

3. Make updates to foreign keys in `CreateForeignKeys.sql`.

4. If you are creating a new table, add a string constant to the `src/com/_360commerce/domain/arts/ARTSDatabaseIfc.java` file. Use a string constant with a meaningful name to store the official ARTS name of the database table.

   Code Sample 8-3 shows two examples of meaningful String constants found in ARTSDatabaseIfc.java.

   Code Sample 8-3 String Constant in ARTSDatabaseIfc.java

```
public static final String TABLE_TENDER_LINE_ITEM = "tr_ltm_tnd";
public static final String TABLE_CREDIT_DEBIT_CARD_TENDER_LINE_ITEM =
"tr_ltm_crdb_crd_tn";
```

5. Update the flat file configuration XML files, if needed.

   If you are creating a new table, consult functional specifications to determine whether the table needs to be represented in the flat files.

   For existing tables, you can inspect the file `pos/config/manager/FFTableDefs.xml` to determine whether the table is represented in the flat files.

   See "Updating Flat File Configurations" on page 8-14 for information on updating the configuration files.

6. Check foreign key constraints.

   For performance reasons, the database build scripts do not turn on foreign key constraints until late. If you make inserts which break foreign key constraints, you will not be notified. To check this, test all inserts with foreign key constraints in place, by editing the appropriate database build script. In the following example, the locations of the `CreateK.sql` and `InsertD.sql` scripts have been swapped:

   Code Sample 8-4 mysql_builddb.bat: Changes to Implement Foreign Key Checking

```
COPY /B %_360COMMON_MYSQL_PATH%\mysql_prologue.sql + %_360COMMON_LOGS_PATH%\CreateS.sql
+ %_360COMMON_LOGS_PATH%\CreateK.sql + %_360COMMON_LOGS_PATH%\InsertD.sql +
%_360COMMON_DB2_PATH%\mysql_epilogue.sql %_360COMMON_LOGS_PATH%\FinalSQL.sq
```

7. Run `c:\360store\pos\bin\dbbuild.bat` to rebuild the database.

   The `dbbuild.bat` script performs the following operations:

   - Executes `CreateTable*.sql` scripts
   - Performs inserts and adds keys
   - Creates flat files in `C:\360store\pos\bin\*.dat`

8. After you verify that the table builds successfully and the code referencing the table works, check your updates into source control.

Example of Saving Data: Storing Tender Information

This section describes how to save data to the database, using credit card tender information as an example.

When completing a retail transaction, a customer can offer multiple forms of payment for a purchase. Each form of payment is a different tender, and the system stores each one as a tender line item. For example, the customer may pay for a $100 purchase with a $50 gift card payment, a $20 store credit payment, and a $30 credit card payment. There are three forms of payment and three tender line items, each potentially requiring different types of data. The following subsections describe how to store the credit card tender data.

Research Table Requirements and Standards

To plan your database code, refer to functional requirements documents to determine what data must be stored. For example, the Credit Functional Requirements specify that the credit card number and expiration date be stored.

Next, review the ARTS database standards for tables and columns. Determine whether you need to create a new table. If you need to create a table defined by ARTS but not currently used in the Store database, follow the ARTS standard. For instructions on creating a new table, see “Creating or Updating Database Tables” on page 8-5.

For the credit card tender transaction, there are two tables that need to be addressed: the tender line item table and the credit/debit card transaction table.

Table 8-1 Database Tables Used in Credit Card Tender Option

ARTS Table Name	Description
tr_ltm_tnd	Tender line item
tr_ltm_crdb_crd_tn	Credit/debit card transactions

Saving Data from Site Code

To save data to the database from a site:

1. Create and populate the domain object to be saved.
2. Save the data to the cargo’s transaction.

   For the credit card tender option, the TenderCargo contains a retail transaction object that keeps track of all the data for each tender line item, as well as other pertinent data. TenderCargo is the cargo for the Tender Tour.

   In Code Sample 8-5, credit is a domain object that stores the credit card data such as number, expiration date, type of card, and so on. Credit was already stored in the cargo as a pending line item in `GetCreditInfoSite.java`. In the following code, credit is retrieved from the cargo and added to the cargo’s retail transaction as a tender line item.

   Code Sample 8-5 ValidCreditInfoEnteredRoad.java: Transaction Object

```
public void traverse(BusIfc bus)
{
  // Get the pending line item
  TenderCargo cargo = (TenderCargo) bus.getCargo();
  TenderChargeIfc credit = (TenderChargeIfc) cargo.getPendingLineItem();
  TenderableTransactionIfc trans = cargo.getTransaction();
  ...
  // Add the credit line item to the transaction
  trans.addTender(credit);
  ...
}
```

3. Call a method to save the transaction object.

   After the credit object is added to the Tender Cargo transaction, the collected data is saved to the database. In Code Sample 8-6, the `com/extendyourstore/pos/services/common/SaveRetailTransactionAisle.java` file uses the Utility Manager to call the saveTransaction() method.

   Code Sample 8-6 SaveRetailTransactionAisle.java: Save Transaction

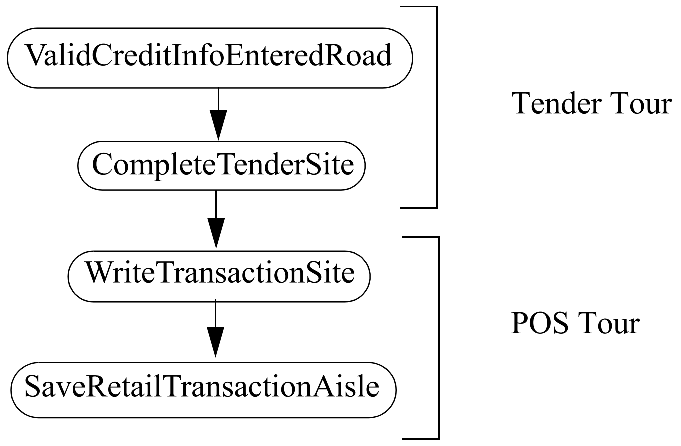
```
public void traverse(BusIfc bus)
{
  ...
  UtilityManagerIfc utility =(UtilityManagerIfc)
bus.getManager(UtilityManagerIfc.TYPE);
  ...
  utility.saveTransaction(trans, totals, till, register);
  ...
}
```

Locate Data Operation

The Data Manager and Data Technician work together to provide access to the database from the application. The developer rarely modifies these. Typically, the site code and the JDBC code are updated. To identify which JDBC class should be used, trace through the site code until the DataAction sets the operation name.

As an example, Figure 8-3 shows the tour workflow that occurs when a tender is complete and the data is ready to be saved.

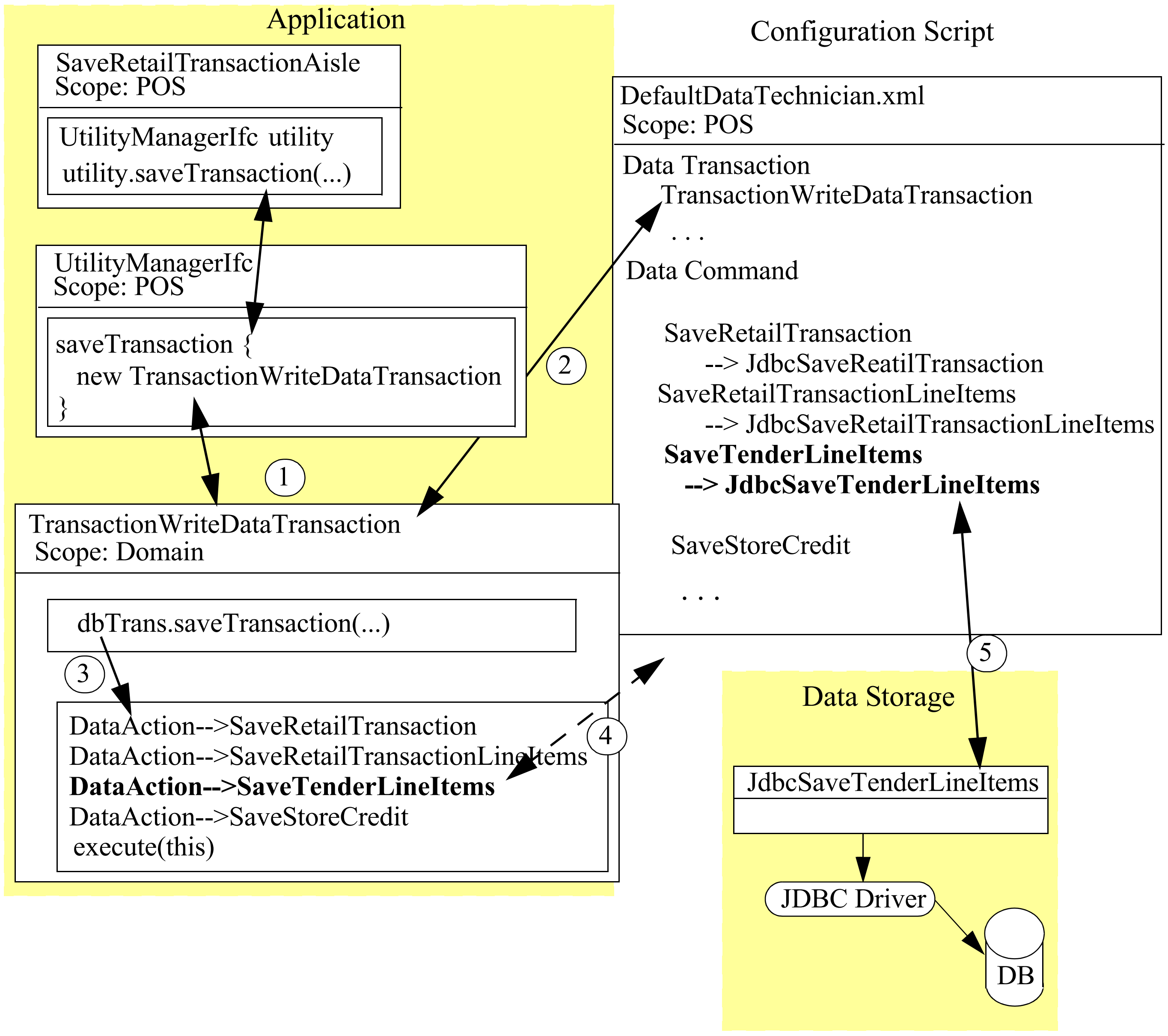
Figure 8-3 Tender Tour to POS Tour Workflow



After the Tender Tour has completed, the program returns to the POS Tour via the WriteTransactionSite to the SaveRetailTransactionAisle. The SaveRetailTransactionAisle initiates the save process.

The conceptual diagram in Figure 8-4 illustrates the basic communication path from the SaveRetailTransactionAisle to the database. For more detail, refer to the source code.

Figure 8-4 Diagram: Saving a Transaction


Application
SaveRetailTransactionAisle
Scope: POS
UtilityManagerIfc utility
utility.saveTransaction(...)
UtilityManagerIfc
Scope: POS
saveTransaction {
new TransactionWriteDataTransaction
}
1
2
TransactionWriteDataTransaction
Scope: Domain
dbTrans.saveTransaction(...)
3
DataAction-->SaveRetailTransaction
DataAction-->SaveRetailTransactionLineItems
DataAction-->SaveTenderLineItems
DataAction-->SaveStoreCredit
execute(this)
4
Configuration Script
DefaultDataTechnician.xml
Scope: POS
Data Transaction
TransactionWriteDataTransaction
. . .
Data Command
SaveRetailTransaction
--> JdbcSaveReatilTransaction
SaveRetailTransactionLineItems
--> JdbcSaveRetailTransactionLineItems
SaveTenderLineItems
--> JdbcSaveTenderLineItems
SaveStoreCredit
. . .
5
Data Storage
JdbcSaveTenderLineItems
JDBC Driver
DB

The following descriptions explain the labels in the figure. When creating the credit card tender option, only the site and road classes for the Tender Tour and the JdbcSaveTenderLineItems class were changed.

1. SaveRetailTransactionAisle uses the Utility Manager to call the `saveTransaction()` method as shown in Code Sample 8-6. The `utility.saveTransaction()` method uses the data transaction class TransactionWriteDataTransaction to save the retail transaction.

   The following code samples show details for Figure 8-4.

   Code Sample 8-7 UtilityManager.java: Save Data Transaction

   ```
   TransactionWriteDataTransaction dbTrans = new
   TransactionWriteDataTransaction(tranName);
   dbTrans.saveTransaction(trans, totals, till, register);
   ```

   Code Sample 8-8 TransactionWriteDataTransaction.java: Save Transaction

```
public void saveTransaction(TransactionIfc transaction,
                                FinancialTotalsIfc   totals,
                                TillIfc              till,
                                RegisterIfc          register)
                                throws DataException
{
  ...
  int transactionType = transaction.getTransactionType();
  ...
  switch(transactionType)
  {                            // begin add actions based on type
    case TransactionIfc.TYPE_SALE:
    case TransactionIfc.TYPE_RETURN:
    addSaveSaleReturnTransactionActions((SaleReturnTransactionIfc)
transaction,totals,till,
      register);
    break;
    ...
}
```

2. The `com/extendyourstore/domain/arts/DefaultDataTechnician.xml` file is the configuration file for the Data Technician and is used to configure the links between the application and the JDBC class that performs the work. All Data Transaction classes must be defined in this file, including TransactionWriteDataTransaction.

Code Sample 8-9 DefaultDataTechnician.xml: Define Data Transaction Class

```
<DATATECHNICIAN
  package="com.extendyourstore.domain.arts">
  ...
  <TRANSACTION name="TransactionWriteDataTransaction" command="jdbccommand"/>
  ...
```

3. The TransactionWriteDataTransaction class instantiates the DataAction object and sets the data operation name to SaveTenderLineItems. Other data actions occurred before these tender data actions. Data Actions are added in the specific order in which they should occur.

Code Sample 8-10 TransactionWriteDataTransaction: DataAction

```
protected void addSaveSaleReturnTransactionActions(SaleReturnTransactionIfc
transaction,
                                                       FinancialTotalsIfc totals,
                                                       TillIfc till,
                                                       RegisterIfc register)
{
  artsTransaction = new ARTSTransaction(transaction);

  // Add a DataAction to save the SaleReturnTransactionIfc
  DataAction dataAction = new DataAction();
  dataAction.setDataOperationName("SaveRetailTransaction");
  dataAction.setDataObject(artsTransaction);
  actionVector.addElement(dataAction);

  // Add a DataAction to save all the line items in the Transaction
  dataAction = new DataAction();
  dataAction.setDataOperationName("SaveRetailTransactionLineItems");
  dataAction.setDataObject(artsTransaction);
  actionVector.addElement(dataAction);

  // Add a DataAction to save all the tender line items in the Transaction
  DataActionIfc da = new SaveTenderLineItemsAction(this, artsTransaction);
  actionVector.addElement(da);

  //Add a DataAction to save store credit in the Transaction
  dataAction = createDataAction(artsTransaction, "SaveStoreCredit");
  actionVector.addElement(dataAction);
  ...
  dataAction =
createDataAction(transaction.getInventoryUpdate(),"UpdateInventoryBalances");
  actionVector.addElement(dataAction);
  ...
}
```

Code Sample 8-11 SaveTenderLineItemsAction: Set Data Operation Name

```
protected static final String OPERATION_NAME = "SaveTenderLineItems";
```

4. The DefaultDataTechnician uses the data command to list several data operation names. The data operation name SaveTenderLineItems points to the name of the JDBC class, which is JdbcSaveTenderLineItems.

   Code Sample 8-12 DefaultDataTechnician.xml: Define Data Operation Class

```
<DATATECHNICIAN
  package="com.extendyourstore.domain.arts">
  ...
  <TRANSACTION name="TransactionWriteDataTransaction" command="jdbccommand"/>
  ...
  <COMMAND name="jdbccommand"
           class="DataCommand"
           package="com.extendyourstore.foundation.manager.data"

  <COMMENT>
    This command contains all operations supported on a JDBC
    database connection.
  </COMMENT>
  <POOLREF pool="jdbcpool"/>
  ...
  <OPERATION class="JdbcSaveTenderLineItems"
    package="com.extendyourstore.domain.arts"
    name="SaveTenderLineItems">
    <COMMENT>
      This operation saves all tender line items associated with the transaction.
    </COMMENT>
  </OPERATION>
        ...
</DATATECHNICIAN>
```

5. The JdbcSaveTenderLineItems class is used to write the credit card data to the database table. See the next section, “Modify Data Operation.”

Modify Data Operation

Use this procedure to modify the data operation class to access the database.

1. Add a save method to the data operation class.

   The `com/extendyourstore/domain/arts/JdbcSaveTenderLineItems.java` file creates the JDBC code that saves the tender line items to the database via the saveTenderLineItem() method, shown in Code Sample 8-13. This code checks the type of a line item. If the tender line item is an instance of the TenderChargeIfc, then it calls the `insertCreditDebitCardTenderLineItem()` method.

Code Sample 8-13 JdbcSaveTenderLineItems: Saving Tender Line Item

```
public void saveTenderLineItem(JdbcDataConnection dataConnection,
                                  TenderableTransactionIfc transaction,
                                  int lineItemSequenceNumber,
                                   TenderLineItemIfc lineItem) throws DataException
{

  if (lineItem instanceof TenderCashIfc)
  {
    insertTenderLineItem(dataConnection,
                         transaction,
                         lineItemSequenceNumber,
                         lineItem);
  }
  else if (lineItem instanceof TenderGiftCardIfc)
  {
    insertGiftCardTenderLineItem(dataConnection,
                         transaction,
                         lineItemSequenceNumber,
                         (TenderGiftCardIfc) lineItem);
  }
  else if (lineItem instanceof TenderChargeIfc)
  {
    /*
     * Charge tender updates the Credit/Debit Card Tender Line Item,
     * Tender Line Item, and Retail Transaction Line Item tables.
    */
    insertCreditDebitCardTenderLineItem(dataConnection,
                                        transaction,
                                        lineItemSequenceNumber,
                                        (TenderChargeIfc)lineItem);
  }
  ...
}
```

2. Write an implementation for methods written for the data operation class.

   Code Sample 8-14 lists the source code for the `insertCreditDebitCardTenderLineItem()`, called in Code Sample 8-13. First, the tender line item must be saved to the tender table using the `insertTenderLineItem()` method. This code already existed for the other tender options.

   Second, the credit data must be saved to the new database table using SQL factory methods.

Code Sample 8-14 JdbcSaveTenderLineItems.java: SQL Factory Methods

```
public class JdbcSaveTenderLineItems extends JdbcSaveRetailTransactionLineItems
                                     implements ARTSDatabaseIfc
{
  public void insertCreditDebitCardTenderLineItem(JdbcDataConnection dataConnection,
                                                  TenderableTransactionIfc transaction,
                                                  int lineItemSequenceNumber,
                                                  TenderChargeIfc lineItem)
  throws DataException
  {
    /*
     * Update the Tender Line Item table first.
     */
    insertTenderLineItem(dataConnection,
                         transaction,
                         lineItemSequenceNumber,
                         lineItem);

      SQLInsertStatement sql = new SQLInsertStatement();

    // Table
    sql.setTable(TABLE_CREDIT_DEBIT_CARD_TENDER_LINE_ITEM);
    // Fields
```

```
      sql.addColumn(FIELD_RETAIL_STORE_ID, getStoreID(transaction));
      sql.addColumn(FIELD_WORKSTATION_ID, getWorkstationID(transaction));
      sql.addColumn(FIELD_BUSINESS_DAY_DATE, getBusinessDayString(transaction));
      sql.addColumn(FIELD_TENDER_AUTHORIZATION_DEBIT_CREDIT_CARD_ACCOUNT_NUMBER,
        getCardNumber(lineItem));
      sql.addColumn(FIELD_TENDER_AUTHORIZATION_CARD_NUMBER_SWIPED_OR_KEYED_CODE,
        getEntryMethod(lineItem));
      sql.addColumn(FIELD_TENDER_AUTHORIZATION_DEBIT_CREDIT_CARD_EXPIRATION_DATE,
        getExpirationDate(lineItem));
    }
    ...
  }
```

Test Code

To test the new code:

1. Run Point-of-Sale.
2. Select the path to the screen.
3. Enter the data.
4. Complete the retail transaction.

Verify Data

To verify that the correct data exists in the database table, use a database access program to view the table that should contain the new information. Verify that the data in the database table matches the data entered. Code Sample 8-15 shows a sample SQL statement you can use to retrieve the data.

Code Sample 8-15 Sample SQL Statement

```
select * from tr_ltm_crdb_crd_tn
```

Updating Flat File Configurations

A Point-of-Sale flat file is a simple database system in which each table is contained in one file. The Point-of-Sale system uses flat files created by the Store Server to provide access to minimal data when the network or server is down. With the help of the flat files, the Point-of-Sale system can continue to process transactions without access to the network.

The information provided in flat files includes:

- Item data, such as price, tax group, SKU
- Tax rules for the local store
- User logon and role information
- Reason codes

When a register is opened at the start of a new business day, the system updates the flat files on the register. The files can also be updated periodically during the business day if an optional parameter is set.

The 360Platform FlatFileEngine provides access to flat file tables. The FlatFileEngine integrates with the Data Technician using the DataConnectionIfc and DataOperationIfc interfaces. A wrapper class, FlatFileDataConnection, implements the DataConnectionIfc interface. The application developer must provide the classes implementing the DataOperationIfc interface for the application-specific operations. Two configuration scripts are required, the Data Technician configuration script and the FlatFileEngine configuration script.

Data Technician Script

The Data Technician script specifies the data connection class and the data operation mappings. Two sections of the XML script are highlighted, the first containing the `OPERATION` tags and the second containing the `CONNECTION` and `CONNECTIONPROPERTY` tags.

- The first highlighted section specifies the mapping of the data actions to data operations. For the FlatFileEngine, the FlatFilePLUOperation and FlatFileEmployeeLookupOperation are classes that implement the DataOperationIfc interface.
- The second highlighted section declares the use of the FlatFileConnection class for the data connection and specifies the configSource property for the connection. Specification of the configSource provides the location of the FlatFileEngine configuration script and is required for the FlatFileEngine to operate.

Code Sample 8-16 PosLFFDataTechnician.xml: Sample Data Technician Script for Flat Files

```
<!DOCTYPE DATATECHNICIAN SYSTEM "classpath://com/extendyourstore/foundation/toru/dtd/
datascript.dtd">

<DATATECHNICIAN
    package="com.extendyourstore.domain.arts">

    <TRANSACTION name="PLU" command="flatfilecommand"/>
    <TRANSACTION name="employee" command="flatfilecommand"/>

    <COMMAND name="flatfilecommand"
             class="DataCommand"
             package="com.extendyourstore.foundation.manager.data" >
        <COMMENT>
            This command contains all operations supported
            on a flat file database connection.
        </COMMENT>
        <POOLREF pool="flatfilepool"/>

        <OPERATION class="FlatFilePLUOperation" package="flatfileops"
            name="PLULookup">
            <COMMENT>
                This operation retrieves a priced item from a
                flat file database, given a string lookup key.
            </COMMENT>
        </OPERATION>
        ... operation omitted here...
    </COMMAND>
    <POOL name="flatfilepool"
            class="DataConnectionPool"
            package="com.extendyourstore.foundation.manager.data" >
          <COMMENT>
            This pool defines a FlatFile connection to the gift registry database.
          </COMMENT>
    <POOLPROPERTY propname="numConnections"
                  propvalue="1" proptype="INTEGER"/>
        <CONNECTION class="FlatFileDataConnection"
                   package="com.extendyourstore.foundation.manager.data.flatfile">
            <CONNECTIONPROPERTY propname="configSource"
                        propvalue="classpath://datafiles/TableDefs.xml" />
        </CONNECTION>
    </POOL>
</DATATECHNICIAN>
```

Flat File Engine Configuration Script

The FlatFileEngine configuration script is required for 360Commerce applications to access the FlatFileEngine. This script specifies the files where the information is stored, the physical schema of the file, and the supported indexes on the files. Code Sample 8-17 is a sample FlatFileEngine configuration script that specifies two tables with associated fields and indexes.

The XML blocks beginning with the tag FWTABLE declare two fixed-width tables with table names Item and Employees. The example configuration in FWFIELDS provides the definitions of the fields within the tables. The field definitions use one base indexing for the starting positions.

After the fields are declared, the following script defines two indexes for the table (indexes are optional). The index names are ItemID_Index and ItemName_Index. The files to store the index information are specified along with the index. Within the individual index specifications, the fields used to generate the index are specified by field name. During configuration, the FlatFileEngine validates the index files and rebuilds the index files if necessary.

Code Sample 8-17 FFTableDefs.xml: Sample FlatFileEngine Configuration File

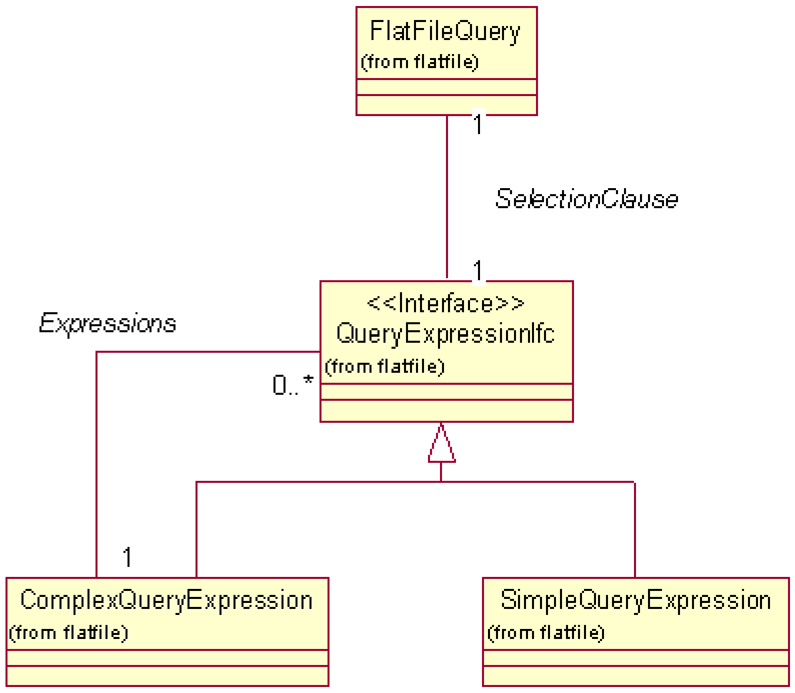
```
<?xml version='1.0' ?>
<!DOCTYPE FFENGINE SYSTEM "classpath://com/extendyourstore/foundation/tour/dtd/flatfile.dtd">


<!—Configuration Script for FlatFileEngine   -->

<FFENGINE>
    <FWTABLE>
        <TABLE  tablename="Items"
                datasource="datafiles/Items.txt"
        />

        <FWFIELDS>
            <FWFIELD fieldname="ItemID"
                     startpos="1"   width="10" />
               <FWFIELD fieldname="Name"
                     startpos="11"  width="80" />
             <FWFIELD fieldname="SupplierID"
                     startpos="91"  width="10" />
               <FWFIELD fieldname="CategoryID"
                     startpos="101" width="10" />
             ...additional fields omitted...
        </FWFIELDS>
```

```
         <INDEXES>
             <INDEX indexname="ItemID_Index"
                    indexfile="datafiles/item_id.idx" >
                 <INDEXFIELD fieldname="ItemID"/>
             </INDEX>
             <INDEX indexname="ItemName_Index"
                    indexfile="datafiles/item_name.idx" >
                 <INDEXFIELD fieldname="Name"/>
             </INDEX>
         </INDEXES>
        </FWTABLE>

        <FWTABLE>
            <TABLE tablename="Employees" datasource="datafiles/Employees.txt"/>
                <FWFIELDS>
                  <FWFIELD fieldname="EmployeeID"
                        startpos="1"  width="10" />
                 <FWFIELD fieldname="LastName"
                        startpos="11" width="20" />
                 <FWFIELD fieldname="FirstName"
                        startpos="31" width="10" />
                 <FWFIELD fieldname="Title"
                        startpos="61"  width="10" />
        ...additional fields omitted...
                </FWFIELDS>

                <INDEXES>
                    <INDEX indexname="Employee_Name"
                        indexfile="datafiles/emp_name.idx" >
                        <INDEXFIELD fieldname="LastName"/>
                        <INDEXFIELD fieldname="FirstName"/>
                    </INDEX>
                    <INDEX indexname="Employee_HireDate"
                        indexfile="datafiles/emp_hire.idx">
                        <INDEXFIELD fieldname="HireDate"/>
                    </INDEX>
                </INDEXES>
        </FWTABLE>
</FFENGINE>
```

Implementing FlatFileDataOperations

To create a FlatFileDataOperation, you create a class that extends the FlatFileDataOperation class and implements the execute method. You must create a FlatFileQuery to communicate with the FlatFileEngine via the `FlatFileDataConnection.execute()` method. The following diagram shows the class relationships.

Figure 8-5 FlatFileQuery Classes



The types of FlatFileQueries are: insert, update, delete, retrieve, clear table, and rebuild indexes. The query type and the target table are specified in the constructor for the FlatFileQuery. Some of the query types (update, delete, and retrieve) require the creation of a selection clause to identify the set of records on which the operation is to be performed. The sample code shown below creates a retrieve query, the most common of the queries that you implement. Differences for other query types are shown following the sample code (see "Other Query Types" on page 8-20).

The sample code shown below is an implementation for an item retrieve operation:

1. The first lines of the method simply cast the connection and get the relevant selection criteria from the dataTransaction object.
2. The major work of the method occurs within the try-catch block. Refer to the comments within the sample code. The input to the FlatFileEngine is a FlatFileQuery. The FlatFileQuery(Instance) is created in the statements immediately following the *try*. First, a new FlatFileQuery instance is created, and then the target data table is specified. Lastly, the selection clause is set to create a new SimpleQueryExpression using the target data fields and the item number supplied in the Data Transaction.
3. Calling the `connection.execute()` method with the FlatFileQuery as a parameter returns a FlatFileResultSet or throws a FlatFileException. If an exception is thrown, it is translated to a DataException by the parent class. If a result set is returned, the set is iterated record by record and the field values within the records are translated to appropriate domain objects.

Code Sample 8-18 Item Retrieve Sample Code

```
public void execute(DataTransactionIfc dataTransaction,
                    DataConnectionIfc dataConnection,
                    DataActionIfc action)
                    throws DataException
{
    FlatFileDataConnection connection =
             (FlatFileDataConnection)dataConnection;

    String prodId = (String)action.getDataObject();
    PLUItems[] pluItems = null;

    try
    {
           // Create a new query of type retrieve for table Items
        FlatFileQuery query =
              new FlatFileQuery(FlatFileQuery.QUERY_RETRIEVE,
                                "Items");
        query.setSelectionClause(
                    new SimpleQueryExpression("ItemID",
                    QueryExpressionIfc.EQ, itemId));

        connection.execute(query);

        FlatFileResultSet rs =
                (FlatFileResultSet)connection.getResult();

        int recCount = rs.getRecordCount();

        if (recCount == 0)
        {
            throw new DataException(DataException.NO_DATA,
                "No PLU was found proccessing the result "
                + set in FlatFilePLUOperation.");
        }

        items = new PLUItem[recCount];

        FlatFileRecord record = rs.getFirstRecord();
        for (int i = 0; i < recCount; i++)
        {
            /*
             * Grab the fields selected from the database
             */
            // Sting fldValue = record.getFieldValue("FIELDNAME");

             // TRANSFER ATTRIBUTES HERE

            record = rs.getNextRecord();
        }
    }

    catch (FlatFileException eff)
    {
        throw translateToDataException(eff);
    }

    dataTransaction.setResult((Serializable)items);
}
```

Other Query Types

The following table provides additional information for creating the query types supported by the FlatFileEngine:

Table 8-2 FlatFileEngine Query Types

Update	The update query allows the application to update field values within a table. The table name is specified and a selection clause is created to identify the record(s) to apply the updated field values. The field values are placed in a hash table, keyed by field name that contains the new field values. The `FlatFileQuery.setValues()` method is called and passes the values hashtable as a parameter. The query is passed as a parameter to the execute method of the collection. The number of records updated is returned via the `getUpdated()` method.
Insert	The insert query inserts a new record into the flat file table. The table name and the values are specified in the query. Values are transmitted using a hashtable keyed by field name. Not all fields require values. A confirmation of the insertion is accessed using the `getInserted()` method.
Delete	The delete query marks the records matching the selection clause for deletion. The table name and a selection clause must be specified in this query. After executing the query, the number of records deleted is available using the `getDeleted()` method. Deleting records invalidates the indexes. To rebuild the indexes, a rebuild query must be executed.
Clear Table	A clear table query removes all the records in a specified table. Only the table name is required. Completion status is available from the `FlatFileResultSet.getCleared()` method.
Rebuild	The rebuild query type removes records marked for deletion from the table and rebuilds the associated indexes. Only the table name is required.

Complex Query Expressions

Complex Query Expressions allow the creation of selection clauses with multiple criteria. To select an employee based on last name and first name, create a ComplexQueryExpression. The logical operation joining the associated expressions is set using the constants AND and OR from the QueryExpressionIfc class as the parameter in the `setJoinCondition()` method. Two SimpleQueryExpression objects are created, one for the last name criteria and one for the first name criteria. These two SimpleQueryExpressions are added to the expressions vector in the ComplexQueryEpression. The selection clause association of the FlatFileQuery is set to the ComplexQueryExpression. The ComplexQueryExpression can contain both Simple and Complex expressions, and supports nested conditions.

EXTENSION GUIDELINES

Customers who purchase Point-of-Sale extend the product to meet their particular needs. These guidelines speed implementation and simplify the upgrade path for future work.

Developers on customer projects should also refer to the Development Standards. The Development Standards address how to code product features to make them less error-prone and more easily maintained. They are especially important if code from the customer implementation may be rolled back into the base product.

Conventions

This section describes conventions used throughout this chapter.

Terms

The following definitions are used throughout the document:

- Product source tree—A directory tree that contains the 360Commerce product code. The contents of this tree do not change, with the exception of product patches. In production code, these files are accessed as external `.jar` files.
- Customer source tree—A directory tree separate from the product code that contains customer-specific files. Some of these files are new files for customer-specific features; others are extensions or replacements of files from the product source tree. The customer tree should not contain packages from the product tree.
- Customer abbreviation—A short name that represents the customer. For example, a company named My Bike Store might use MBS as their customer abbreviation. The MBS example is used throughout this chapter; replace MBS with the customer abbreviation for your own project when writing code. The customer abbreviation is added to filenames to clarify that the file is part of the customized code, and is used as part of the package name in the customer source tree.

Filename Conventions

Filenames in the customer source tree usually include the customer abbreviation. Name files according to the following rules:

- If a class in the customer source tree extends or replaces a class in the product source tree, use the customer abbreviation followed by the original filename as the new filename(i.e., `SaleReceipt.java` becomes `MBSSaleReceipt.java`).
- New Java classes should also begin with the customer abbreviation.
- Script or properties file names that are hard-coded in Foundation classes must use the same filename in the customer source tree as was used in the product source tree (for example, `posfoundation.properties`).

Modules

The Point-of-Sale system is divided into a number of different modules, and each module corresponds to a project in an integrated development environment (IDE). When setting up a development environment for modifying code, building Point-of-Sale, and testing changes, you must configure your system to make MBSpos dependent on all the other modules.

To set up your development environment:

1. Check out each of the required customer modules as shown in Table 9-1.
2. Reference each of the standard modules as external `.jar` files.
3. Add the required modules to your CLASSPATH environment variable in the order shown in Table 9-1, with all of the customer modules preceding the set of standard modules.

Table 9-1 Required Modules in Dependency Order

Customer Modules	Standard Modules
MBS pos (root, src, locales_US and other language directories)[1]	pos (root, src, locales_US and other language directories)
MBS domain (root and src)	domain (root and src)
MBS commerce services	360common
MBS common	commerce services
MBS 3rd-party	foundation
	3rd party

1. Directory names in parentheses must be specified individually in the classpath.

Directory Paths

Paths given in this chapter are relative, starting either with the module or with the source code, as follows:

- Paths beginning with a module name start from the module location. `pos\config` refers to the `config` directory within the pos module, wherever that module is located on your system.
- Paths beginning with `com` refer to source code. Source code paths are nested within modules, in `\src` directories. Multiple `\src\com` file hierarchies are built together into one file structure during compilation. For example, a reference to `com\_360commerce\pos\services\tender` can be found in the `pos` module's `src` directory. If your pos module is in `c:\workspace\360store`, then the full path is:
`C:\workspace\360store\pos\src\com\_360commerce\pos\services\tender`

pos Package

This section addresses extension of files in the pos package.

Note: The pos module may be nested within a `360store` directory in the source code control system.

Tour

You extend tours mainly by editing proprietary XML scripts developed by 360Commerce. This section describes how to customize tours, beginning with replacing the Tour Map, and continuing with customization of individual tours or parts of tours.

Tour Map

The product code references tours at transfer stations by logical names, so that you can change a single tour without having to update references to that tour in various tour scripts. Tour maps tell the system the specific tour files to use for each logical name.

The tour map also enables overlays of tour classes. If a tour script does not need to be customized, but some of the Java classes do, the tour map can specify individual classes to customize. Note that any class files must still use their own unique names (such as `MBScashSelectedAisle.java` for a new Aisle used in place of `CashSelectedAisle.java`).

Typically, the base product Tour Map file, `tourmap.xml`, does not change. Instead, you create a custom Tour Map for your project, and an additional one for each supported locale beyond your default locale. Each of these Tour Map files contains only the differences it adds to the base Tour Map.

Follow these steps to add new Tour Map files:

1. Create one custom Tour Map file for each supported country in the `pos\config` directory of the customer source tree. Initially, these Tour Map files may be empty; as you customize tour components, you can add tags. The following sample shows the initial state of the file:

   Code Sample 9-1 MBStourmap_CA.xml: Sample initial tourmap file for Canadian locale

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <tourmap
      country="CA"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="com/extendyourstore/foundation/tour/dtd/tourmap.xsd">

   ...Tour tags can be added here...

   </tourmap>
   ```

2. Copy the `pos\config\posfoundation.properties` file to the customer source tree. Modify the `tourmap.files` property in this file, adding the names of the new Tour Map files. Do not rename the `posfoundation.properties` file, since this filename is referenced by Foundation classes. It is important to keep the customized tour map files after the product tour map file in the list, since the files listed *later* override earlier files.

   Code Sample 9-2 posfoundation.properties: Adding new Tour Maps

   ```
   # comma delimited list of tourmap files to overlay
   tourmap.files=tourmap.xml, MBStourmap.xml, MBStourmap_CA.xml
   ```

3. Refer to the procedures that follow to modify tour scripts and Java components of a tour.

Tour Scripts

If you need to change the workflow of a tour, you must replace the tour script; you cannot extend a tour script. To replace a tour script, follow these steps:

1. Create a new XML tour script in the customer source tree.
2. Modify the tour map in the customer source tree to specify the correct package and filename for the new tour script. The logical tour name must stay the same.

   Code Sample 9-3 tourmap_CA.xml: Replacing one tour script

   ```
   <tour name="tender">
         <file>classpath://com/mbs/pos/services/tender/tender.xml</file>
   </tour>
   ```

3. Copy and modify sites, roads, aisles, shuttles and signals.

Site

Extending siteactions in the traditional object-oriented sense is not recommended; letters mailed in the original arrive method would conflict with the arrive method in the extended class. Since siteactions represent relatively small units of code, they should be replaced instead of extended. Follow these steps:

1. Create a new siteaction class in the customer source tree, such as `MBScashSelectedSite.java`.
2. If you are overlaying a siteaction class, but not modifying the tour script, then all letters that were mailed from the product version of the siteaction class should also be mailed from the new version. Do not mail new letters that are not handled by the product code, unless the tour script and related Java classes are also modified.
3. Edit the appropriate Tour Map for the locale, using the `replacewith` property in the `<SITEACTION>` tag to define the new package and filename for the siteaction class.

   Code Sample 9-4 tourmap_CA.xml: Replacing a siteaction

   ```
   <tour name="tender">
      <file>classpath://com/mbs/pos/services/tender</file>
      <SITE
         name="cashSelected"
         useaction="com.extendyourstore.pos.services.tender.cashSelectedSite"/>
      <SITEACTION
         class="cashSelectedSite"
         replacewith="com.mbs.pos.services.tender.MBScashSelectedSite"/>

   </tour>
   ```

Lane—Road or Aisle

As with siteactions, extending laneactions in the traditional object-oriented sense is not recommended, as letters from the original and extended classes could conflict. Replace laneactions instead of extending them, using the following steps:

1. Create a new laneaction class in the customer source tree, such as `MBSOpenCashDrawerAisle.java`.
2. If you are overlaying a siteaction class, but not modifying the tour script, then all letters that were mailed from the product version of the laneaction class should also be mailed from the new version. Do not mail new letters that are not handled by the product code, unless the tour script and related Java classes are also modified.

3. Edit the appropriate Tour Map for the locale, using the `replacewith` property in the `<LANEACTION>` tag to define the new package and filename for the laneaction.

Code Sample 9-5 tourmap_CA.xml: Replacing a laneaction

```
<tour name="tender”>
       <file>classpath://com/mbs/pos/services/tender</file>
       <SITE
          name="RefundDueUI"
          useaction="com.mbs.pos.services.tender.refundDueUISite">"/>
       <LANEACTION
          class="OpenCashDrawerAisle"
          replacewith="com.mbs.pos.services.tender.MBSOpenCashDrawerAisle"/>

    </tour>
```

Shuttle

Since shuttles do not mail letters, they may be extended or replaced; however extending them is recommended. Follow these steps in either case:

1. Modify the shuttle class.

   Create a new class in the customer source tree. If it extends or replaces the product bean class, add the customer abbreviation to the filename. For example, `TenderAuthorizationLaunchShuttle.java` becomes `MBSTenderAuthorizationLaunchShuttle.java`.

2. Edit the appropriate Tour Map for the locale, using the `replacewith` property in the `<SHUTTLE>` tag to define the new package and filename for the shuttle.

Code Sample 9-6 tourmap_CA.xml: Replacing or Extending a shuttle

```
<tour name="tender”>
       <file>classpath://com/mbs/pos/services/tender</file>
       <SITE
          name="RefundDueUI"
          useaction="com.mbs.pos.services.tender.refundDueUISite">"/>
       <SHUTTLE
          class="TenderAuthorizationLaunchShuttle"
          replacewith="com.mbs.pos.services.tender.MBSTenderAuthorizationLaunchShuttle"/>

    </tour>
```

3. Modify the calling and nested tour scripts as necessary to adjust to the change.

Signal

Extending signals in the traditional object-oriented sense is not recommended. This is because signals are typically so small that extending an original signal class makes them overly complex.

The REPLACEWITH tag of the TourMap does not work for Signals. The tour script must be customized to refer to the package and filename of the new signal. Follow these steps:

1. Create a new signal class in the customer source tree. For example, create a replacement for `IsAuthRequiredSignal.java` in the Tender service by creating a class file `com\mbs\pos\services\tender\MBSIsAuthRequiredSignal.java`.

2. Customize the appropriate tour script:

Code Sample 9-7 MBStender.xml: Tender tour script with customized signal

```
<SERVICECODE>
... non-signal declarations omitted...
     <SIGNAL class="IsReturnTransactionSignal" />
```

```
      <SIGNAL class="IsSaleTransactionSignal" />
      <SIGNAL class="IsNotVoidTransactionSignal" />
      <SIGNAL class="IsAuthNotRequiredSignal" />
      <SIGNAL class="MBSIsAuthRequiredSignal" package="com.mbs.pos.services.tender" />
      <SIGNAL class="IsRemoveTenderSignal" />
      <SIGNAL class="IsNoRemoveTenderSignal" />
      <SIGNAL  class="IsValidDriverLicenseSignal" />
      <SIGNAL  class="IsInvalidDriverLicenseSignal" />
... more declarations omitted...
</SERVICECODE>
... code omitted...
<ROAD name="AuthorizationRequested"
                     letter="Next"
                     destination="AuthorizationStation"
                     tape="ADVANCE"
                     record="OFF"
                     index="OFF">
                    <LIGHT signal="MBSIsAuthRequiredSignal"/>
```

Cargo

Since cargos do not mail letters, they may be extended or replaced. Cargo classes are typically part of a hierarchy of classes. Follow these steps:

1. Modify the cargo class by doing one of the following:
   - To extend the cargo, create a new class in the customer source tree that extends the cargo in the product source tree. Be sure to extend from the lowest-level subclass. Add the customer abbreviation to the beginning of the filename.
   - To replace the cargo, create a new cargo class in the customer source tree.
2. Edit the appropriate Tour Map for the locale, using the `replacewith` property in the `<CARGO>` tag to define the new package and filename for the cargo.

   Code Sample 9-8 tourmap_CA.xml: Replacing a Cargo
   ```
   <tour name="tender">
          <file>classpath://com/mbs/pos/services/tender</file>
          <SITE
             name="RefundDueUI"
             useaction="com.mbs.pos.services.tender.refundDueUISite">"/>
          <CARGO
             class="TenderCargo"
             replacewith="com.mbs.pos.services.tender.MBSTenderCargo"/>

       </tour>
   ```
3. Modify the tour map and/or tour scripts and shuttles of the calling and nested tours to adapt to the cargo modifications. Be sure to address:
   - Classes in the same tour as the modified cargo
   - All tours for which this tour is a nested tour
   - All tours which are called by this tour

UI Framework

The UIManager and UITechnician classes are provided by Foundation. They are configurable through the Conduit Script and should not be modified directly. This section describes customization to the default UI configuration and individual screens.

Default UI Config

The product file `pos\config\defaults\defaultuicfg.xml` contains the building blocks for the UI (displays, templates and specs) and references to all tour-specific `uicfg.xml` files. If you change any UI script in the customer implementation, the `defaultuicfg.xml` file must be replaced. It also needs to be replaced if the displays, templates, and basic bean specs need to be replaced. Follow these steps to replace the file:

1. Copy the file `defaultuicfg.xml` to the `pos\config\defaults` directory in the customer source tree, and rename it (for example, to `MBSdefaultuicfg.xml).`
2. Modify the displays, templates, default screens, and specs as necessary to represent the customer's user interface.
3. Verify that the conduit script for the client tier has been customized and is located in the customer source tree.
4. Modify the client conduit script to include the new filename and package name for the `MBSdefaultuicfg.xml` file, in the configFilename property value in the UISubsystem section of the UITechnician tag.

Code Sample 9-9 ClientConduit.xml: Conduit script modified to use custom UI configuration file

```
<TECHNICIAN
        name="UITechnician"
        class="UITechnician"
        package="com.extendyourstore.foundation.manager.gui" export="Y">

        <CLASS
            name="UISubsystem"
            package="com.extendyourstore.pos.ui"
            class="POSJFCUISubsystem">

            <CLASSPROPERTY
                    propname="configFilename"
                    propvalue="classpath://com/mbs/pos/config/defaults/MBSdefaultuicfg.xml"
                    proptype="STRING"/>
...additional class properties omitted...
        </CLASS>
</TECHNICIAN>
```

UI Script

A UI script changes if the overlays or unique bean specifications of one or more screens in a tour need to be modified. Follow these steps:

1. Create a new UI script in the customer source tree. For example, copy the `tenderuicfg.xml` file from the product source tree to the customer source tree and rename it `MBStenderuicfg.xml`.
2. Modify the `MBSdefaultuicfg.xml` file in the customer source tree to refer to the new filename and package for the UI script.

Code Sample 9-10 MBSdefaultuicfg.xml: Customized Default UI Configuration File

```
... other include statements omitted...
```

```
<INCLUDE filename="classpath://com/_360commerce/pos/services/sale/saleuicfg.xml"/>
    <INCLUDE filename="classpath://com/mbs/pos/services/tender/MBStenderuicfg.xml"/>
    <INCLUDE filename="classpath://com/_360commerce/pos/services/tender/capturecustomerinfo/
capturecustomerinfouicfg.xml"/>
    <INCLUDE filename="classpath://com/extendyourstore/pos/services/inquiry/
inquiryoptionsuicfg.xml"/>
... other include statements omitted...
```

Bean Model and Bean

The Point-of-Sale product code provides generalized beans that are designed to be reused as-is, such as `GlobalNavigationButtonBean.java` for the global navigation button bar and `DataInputBean.java` for the work area of form layout screens. These classes are not intended to be extended for a specific implementation, though they may be extended if the general behavior or data must change in all cases.

The classes can be used for different screens within the application without changing to Java code by modifying parameter values and calling methods on the bean. Use the generalized beans whenever possible and avoid beans specialized for only one screen. However, bean and bean model classes in the product code that are specific to an individual screen, such as `CheckEntryBean.java` and `CheckEntryBeanModel.java`, may be customized. Follow these steps to modify a bean model:

1. Create a new bean model class.

   Create a new class in the customer source tree, and add the customer abbreviation to the filename.
2. Copy tour files that need to reference the new bean model into the customer source tree. Modify them to create and manipulate data for the new bean model.

Follow these steps to modify the bean:

1. Create a new bean class.

   Create a new class in the customer source tree, and add the customer abbreviation to the filename.
2. Modify the UI config scripts that reference the bean class **in the customer source tree** to refer to the new bean class filename and package.

Code Sample 9-11 MBStenderuicfg.xml: Tender UI Configuration with Customized Bean Reference

```
<UICFG>

    <BEAN
        specName="TenderOptionsButtonSpec"
        configuratorPackage="com.extendyourstore.pos.ui"
        configuratorClassName="POSBeanConfigurator"
        beanPackage="com.mbs.pos.ui.beans"
        beanClassName="MBSNavigationButtonBean">

        <BUTTON
            actionName="Cash"
            enabled="true"
            keyName="F2"
            labelTag="Cash"/>
...other buttons omitted...
     </BEAN>
...other UI objects omitted...
</UICFG>
```

Other

This section covers customization of components other than the tour and the UI framework, including internationalization and localization changes as well as conduit scripts, PLAF, receipts, and reports.

Internationalization

The process of internationalization includes modifications to the code so that a single code base can support multiple languages. The base product supports US English, Canadian French and Puerto Rican Spanish. If additional languages need to be supported, additional internationalization steps need to be completed by the customer.

1. For each non-product-supported language, create a new directory in the `pos\trunk\locales` directory within the customer source tree. Locale names consist of a two-letter lowercase code for the country, an underscore, and a two-letter uppercase code for language. Examples are en_US for United States English and fr_CA for Canadian French. Copy the resource bundles from `pos\trunk\locales\en_US\config\ui\bundles` to the `config\ui\bundles` directory for the given country. Modify the text for that language and country combination.
2. Create help files for each of the supported languages similar to those for English located in `locales\en_US\config\ui\help`. Refer to the Point-of-Sale *Administrator's Guide* for more detail on writing help files.
3. Create images for each of the supported locales similar to those located for English in `locales\en_US\config\ui\images`.
4. Maximum lengths for input fields may need to be increased for languages that generally have longer words (for example, German) or for double- and multi-byte character set support. The maximum lengths are found in UI scripts located in `pos\trunk\src\com\extendyourstore\pos\services` directories, and parameter files, located in `pos\trunk\config\parameter`.
5. Maximum lengths for database fields in internationalized tables may need to be increased for languages that generally have longer words or for double- and multi-byte character set support. This requires modifying the field length in the database and the `CreateTableX.sql` build script. Data operations classes that refer to the fields should be checked for length dependencies and modified if necessary.
6. Metadata stored in the database also needs to be internationalized. Tables that contain text that should be represented for each supported language have a corresponding table to store text for the non-default languages. For example, the ORDER table includes fields for all ORDER information including text fields for the default language. The ORDER_I8 table includes a row for each text field and locale combination for the ORDER table. If a new language is added, rows could be added to the _I8 table for the new language.
7. If double- or multi-type character sets are to be supported, I8 tables must be translated into UTF-8 format. Follow these steps:
   - Install fonts if not installed on the current operating system.
   - Use a translation editor such as NJStar to translate text and save translations in UTF-8 format.
   - Use the `<JAVA_HOME>/bin/native2ascii` executable to process the UTF-8 file and save it as properly named resource bundle file. For example, `posText_zh_CN.properties` is the filename for simplified Chinese as Unicode.

- Modify the `font.properties` files located in the `<JAVA_HOME>/jre/lib` directory. Search for 'font.properties' on the http://ww.java.sun.com Web site for more information on what can be modified in the `font.properties` files.
- Refer to the client side conduit script to determine the look-and-feel property file. The file is defined as a uiPropertyFile classproperty of the UISubSystem tag. The product default is named `tigerplaf.properties`. Modify this file to use only the Helvetica font. This system font allows the double- and multi-byte character sets to be rendered properly.

Localization

Once the application has been internationalized to support all necessary languages, it must be customized to reflect default and alternate locales. These modifications affect default and alternate locales and various formats (for example, date and currency). Follow these steps:

1. If one does not already exist, place a copy of the conduit script for the client tier in the customer source tree, where it can be customized.
2. Modify the customer version of the conduit script to update the `<LOCALE>` element to specify the default language and country. Also modify the tag to specify any alternate language and countries.
3. Copy `application.properties`, located in config, to the customer source tree. Modify the default_locale and supported_locales properties.
4. Copy `domain.properties`, located in `domain\trunk\config`, to the customer source tree in the domain package. (Most references to the domain package are in the following section. This reference is kept here to include all localization efforts in one location.) Modify currency, date, address and other formats.
5. Modify the default and alternate currencies. Edit the `InsertTableCurrency.sql` file located in the `db\sql` directory of the `commerceservices` package. There is one INSERT statement for each type of currency. Set the value of the DE_CNY field to '1' for the default currency only. If a supported currency does not have an INSERT statement in this file, you must add one and also add a corresponding Java class in the currency package. After the `.sql` file is updated, rebuild the database so that the change can take effect.
6. Set the exchange rate. Edit the `InsertTableExchangeRate.sql` file located in the `db\sql` directory of the `commerceservices` package. This file contains exchange rates for each entry in the currency table. After this `.sql` file has been updated, the database needs to be rebuilt for the change to take effect.
7. Set the taxes. The base product supports US and Canadian taxes. To add support for additional country's taxes requires custom code. Modify `InsertTableAddress.sql` to update the lo_ads table similar to the following, for the tax locale to be British Columbia, Canada, perform the following SQL statements.

   Code Sample 9-12 InsertTableAddress.sql: Sample lo_ads table updates
   ```
   update lo_ads set ST_CNCT = 'BC' where id_prty = <party id number>;
   update lo_ads set CO_CNCT = 'CA' where id_prty = <party id number>;
   ```
   Run `dbbuild.bat` to include the new data in the database.
8. Configure devices. They should be updated to reflect the target locale.

Conduit Scripts

The conduit scripts provided with 360Store applications define a typical tier configuration and are usually replaced with customer conduit scripts for a given implementation. Conduit scripts include an XML file and a `.bat` and `.sh` file to execute the XML; both `.bat` and `.sh` versions of the batch file are provided to support Windows and Linux.

Follow these steps to set up customer conduit scripts:

1. Copy the conduit scripts (client, server, and collapsed) to the customer source tree.

   Copy the XML and `.bat` and `.sh` files for each type of conduit script. Rename the scripts using the customer abbreviation (`ClientConduit.xml` becomes `MBSClientConduit.xml`).
2. Edit each XML file to include only the managers and technicians that should be loaded on the given tier.
3. Modify the class and package names for any managers, technicians and configuration scripts that have been customized.

   Code Sample 9-13 MBSClientConduit.xml: Customized with New Data Manager
   ```
   <MANAGER name="DataManager" class="MBSDataManager"
            package="com.mbs.foundation.manager.data">
       <PROPERTY propname="configScript"
                 propvalue="classpath://config/manager/PosDataManager.xml" />
   </MANAGER>
   ```
4. Modify your development environment to pass in the new conduit XML file as a parameter to the TierLoader.
5. Edit the `.bat` and `.sh` files to pass the correct conduit XML files to the Java environment.

PLAF

Point-of-Sale implements a pluggable look-and-feel (PLAF) so that customers may modify the look of the application including screen colors and images. To modify the PLAF, follow these steps:

1. Create a new properties file that is a copy of one of the following files. Place the file in the `com\mbs\pos\config` directory in the customer source tree.
   - `tigerplaf.properties`—yellow-and-purple, text-based LAF
   - `imagePlaf.properties`—blue and gold image-based LAF
2. Update the conduit scripts in the customer source tree to specify the package and filename for the new LAF file in the UI Technician tag.
3. Have new UI beans call `uiFactory.configureUIComponent(this, UI_PREFIX)` in the `initialize()` method to set the look-and-feel.

Receipts

Receipts are composed of two levels:

- A base receipt that manages data and behavior for all receipts
- Specific receipt types such as Layaway and Return

The receipt class names are specified in the tour code and there is no factory for creating receipts. Therefore, modifications to the tour code that accesses the receipts are required.

If the base receipt and specific receipt classes are both going to be extended, typical inheritance is not sufficient since Java does not support multiple inheritance. For example, the `MBSLayawayReceipt.java` class cannot extend both `MBSPrintableDocument.java` and `MBSLayawayReceipt.java`. The recommended approach is to extend both classes, and have `MBSLayawayReceipt.java` extend `LayawayReceipt.java`. `MBSLayawayReceipt.java` then includes an instance of `MBSPrintableDocument.java` and methods can be called on the extended class.

Follow these steps to customize receipts:

1. If modifications are required to the base receipt, create a class in the customer source tree named `MBSPrintableDocumentUtility.java`. This class is a utility class since the receipt classes delegate common functionality to it.
2. For each receipt type that needs to be customized, do one of the following:
   - To modify an existing receipt type, create a Java class in the customer source tree that extends the receipt class in the product code. Add the customer abbreviation to the beginning of the filename.
   - To create a new receipt type, create a Java class in the customer source tree that extends `MBSPrintableDocument.java`.
3. For extended classes, include an instance of the `MBSPrintableDocumentUtility.java class`. Call methods on the utility class when a customized method is required.
4. Modify tours in the customer source tree as necessary to call `new()` for the customized receipt types.
5. Modify parameters for the receipt header and footer as necessary.

Reports

Point-of-Sale has a set of reports that print on the slip printer. These reports are in a proprietary format and do not use a reporting engine. The report class names are specified in the tour code and there is no factory for creating reports. Therefore, modifications to tours that access the reports are required.

To modify existing Point-of-Sale reports, the report Java files can be extended. Follow these steps:

1. For each report, do one of the following:
   - To modify an existing report, create a Java class in the customer source tree that extends the reports class in the product code (found in `pos\trunk\srb\com\extendyourstore\pos\reports`).
   - To create a new report, create a Java class in the customer source tree that extends the abstract RegisterReport class in the product code. Use the customer abbreviation in the filename.
2. Create, modify or override data and methods as necessary to modify the report.
3. Modify the tour code that creates the report object to call `new()` for the new report class.

domain Package

This section addresses customization of files in the domain package. The domain package can be found in the `\360store\domain` directory in your source control system.

Retail Domain

The Retail Domain provides a retail-specific implementation of business objects. These objects are easily extended to meet customer's requirements.

DomainObjectFactory

If any Retail Domain Objects (RDOs) are added or extended, the DomainObject Factory must be extended. This needs to be done only one time for the application. The extended class must include `getXinstance()` methods for all new and extended RDOs, where X is the name of the RDO. Follow these steps:

1. Create a new Java class that extends `DomainObjectFactory.java`. It should be named with the customer abbreviation in the filename `MBSDomainObjectFactory.java` and be located in the customer source tree.
2. Copy the `domain.properties` file to the `domain\config` directory of the customer source tree. Modify the setting for the DomainObjectFactory to refer to the new package and class name created in the previous step.
   ```
   DomainObjectFactory=com.acmebrick.domain.factory.MBSDomainObjectFactory;
   ```
3. Add `getXInstance()` methods as necessary for new Retail Domain Objects.

Retail Domain Object (RDO)

Follow these steps to create or extend an RDO:

1. Complete one of the following steps:
   - To create a new RDO, create a Java class in the customer source tree in the appropriate subdirectory of `domain\src\com\mbs\domain`. Extend an appropriate superclass from the product code. At a minimum, the new class must extend `EYSDomainIfc.java`.
   - To modify an existing RDO, create a Java class in the customer source tree that extends an RDO in the product code.

   Include the customer abbreviation in the filename; for example, you might name your class file `MBSCustomer.java`.
2. Add data attributes and methods required by the customer-specific functionality.
3. Create `setCloneAttributes()`, `equals()` and `toString()` methods to address the new data attributes and then reference the corresponding superclass method.
4. Complete one of the following steps:
   - For a new RDO, add a new `getXInstance()` method to `MBSDomainObjectFactory.java` for the new RDO.
   - For an extended RDO, ovverride the existing `getXInstance()` method in `MBSDomainObjectFactory.java` to return an object of the new class type.
5. Access the new RDO data and methods from tours located in the customer source tree. If product tours need to access the new RDO data and methods, the tours must be modified.
6. If the RDO data is represented on a screen, modify the UI script, bean and bean model classes to reflect the change.

7. If the RDO is saved to the database, modify the data operation to save the new data attributes.

Database

This section details how to extend database behavior through changes to the data operations. The architecture of the Data Technician simplifies this somewhat, because changes to data operations can be implemented without changes to the Point-of-Sale application code.

Data Manager and Technician Scripts

The Data Manager and Data Technician Scripts, `DefaultDataManager.xml` and `DefaultDataTechnician.xml`, are routinely customized when transactions, data actions, and data operations are customized. See the next section for details.

Data Actions and Operations

When a new or modified RDO contains data that need to be saved to the database, a data operation class must be created or extended. A Data Action must be modified if a unit of database work is changed.

1. Create class files.

   Create new class files for each new or modified item in the customer source tree. If an item extends a product class, add the customer abbreviation to the filename.

2. If a customized version of `POSDataManager.xml` does not already exist, copy it to the customer source tree and give it a new name, such as `MBSPOSDataManager.xml`.
3. For customized transactions with new filenames, modify the transaction name.
4. If a customized version of `DefaultDataTechnician.xml` does not already exist, copy it to the customer source tree and give it a new name, such as `MBSDefaultDataTechnician.xml`.
5. Edit the customized `MBSDefaultDataTechnician.xml` file, updating package and class names for data actions and data operations that have been modified.

   Code Sample 9-14 MBSDefaultDataTechnician.xml: Customizing a Data Operation

```
     <OPERATION class="JdbcSaveTenderLineItems"
            package="com.mbs.domain.arts"
            name="MBSSaveTenderLineItems">
            <COMMENT>
                This operation saves all tender line items associated
                with the transaction.
            </COMMENT>
        </OPERATION>
```

6. Modify the conduit scripts to reference the new package and/or filename of the technician script.

   Code Sample 9-15 CollapsedConduitFF.xml: Customizing the Data Technician

```
<TECHNICIAN name="LocalDT" class="DataTechnician"
                package="com.mbs.foundation.manager.data"
                export="Y">
        <PROPERTY
            propname="dataScript"
            propvalue="classpath://config/manager/MBSDefautlDataTechnician.xml"
        />
    </TECHNICIAN>
```

Data Transactions

Data transactions are the valet classes that carry requests from the client to the server. A data transaction factory implements the factory pattern for data transaction classes. The application code asks the factory for a transaction object and the factory determines which Java class is used to create the object. To create or extend a data transaction class, follow these steps:

1. Create new or modified data transactions.

   Create a Java class in the customer source tree and prepend the customer abbreviation to the filename. If you are modifying an existing transaction, have the class extend the transaction class in the product code, and overwrite the methods you are modifying.

2. Copy `POSDataManager.xml` to the customer source tree.
3. For customized transactions with new filenames, modify the transaction name.
4. Copy `DefaultDataTechnician.xml` to the customer source tree.
5. Modify package and class names for data actions and data operations that have been modified.
6. If not already done, modify the conduit scripts to reference the new package and/or filename of the technician script.
7. Extend `DataTransactionKeys.java` as `MBSDataTransactionKeys.java` in the customer source tree to add or modify the static final String for each transaction (the file serves as a list of string constants).

   Code Sample 9-16 MBSDataTransactionKeys.java: Adding Strings

```
public static final String
DATA_MAINTENANCE_TRANSACTION="data.transaction.DATA_MAINTENANCE_TRANSACTION
public static final String PLU_RETURN_TRANSACTION" =data.transaction.PLU_RETURN_TRANSACTION"
```

8. Update `domain.properties` in the customer source tree to add or modify the name/value pairs for each transaction.

   Code Sample 9-17 domain.properties: Sample Modified and New Data Transactions

```
# Registry of DataTransactionIfc implementations
# (try to keep in alphabetical order)
#

data.transaction.ADVANCED_PRICING_DATA_TRANSACTION=com.extendyourstore.domain.arts.AdvancedPricin
gDataTransaction
...code omitted here...
data.transaction.REGISTER_STATUS_TRANSACTION=com.MBS.domain.data.transactions.RegisterStatusTrans
action
data.transaction.REGISTRY_DATA_TRANSACTION=com.extendyourstore.domain.arts.RegistryDataTransactio
n
data.transaction.STORE_LOOKUP_DATA_TRANSACTION=com.MBS.domain.data.transactions.StoreLookupDataTr
ansaction


MBSdata.transaction.DATA_MAINTENANCE_TRANSACTION=com.MBS.domain.data.transactions.DataMaintenance
Transaction
MBSdata.transaction.PLU_RETURN_TRANSACTION=com.MBS.domain.data.transactions.ReturnPluTransaction
```

CHAPTER 10

GENERAL DEVELOPMENT STANDARDS

The following standards have been adopted by 360Commerce product and service development teams. These standards are intended to reduce bugs and increase the quality of the code. The chapter covers basic standards, architectural issues, and common frameworks. These guidelines apply to all 360Commerce applications.

Basics

The guidelines in this section cover common coding issues and standards.

Java Dos and Don'ts

The following dos and don'ts are guidelines for what to avoid when writing Java code.

- DO use polymorphism.
- DO have only one return statement per function or method; make it the last statement.
- DO use constants instead of literal values when possible.
- DO import only the classes necessary instead of using wildcards.
- DO define constants at the top of the class instead of inside a method.
- DO keep methods small, so that they can be viewed on a single screen without scrolling.
- DON'T have an empty catch block. This destroys an exception from further down the line that might include information necessary for debugging.
- DON'T concatenate strings. 360Commerce products tend to be string-intensive and string concatenation is an expensive operation. Use `StringBuffer` instead.
- DON'T use function calls inside looping conditionals (for example, `while (i <=name.len())`). This calls the function with each iteration of the loop and can affect performance.
- DON'T use a static array of strings.
- DON'T use public attributes.
- DON'T use a switch to make a call based on the object type.

Avoiding Common Java Bugs

The following fatal Java bugs are not found at compile time and are not easily found at runtime. These bugs can be avoided by following the recommendations in Table 10-1.

Table 10-1 Common Java Bugs

Bug	Preventative Measure
null pointer exception	Check for null before using an object returned by another method.
boundary checking	Check the validity of values returned by other methods before using them.
array index out of bounds	When using a value as a subscript to access an array element directly, first verify that the value is within the bounds of the array.
incorrect cast	When casting an object, use `instanceof` to ensure that the object is of that type before attempting the cast.

Formatting

Follow these formatting standards to ensure consistency with existing code.

Note: A code block is defined as a number of lines proceeded with an opening brace and ending with a closing brace.

- Indenting/braces—Indent all code blocks with four spaces (not tabs). Put the opening brace on its own line following the control statement and in the same column. Statements within the block are indented. Closing brace is on its own line and in same column as the opening brace. Follow control statements (if, while, etc.) with a code block with braces, even when the code block is only one line long.
- Line wrapping—If line breaks are in a parameter list, line up the beginning of the second line with the first parameter on the first line. Lines should not exceed 120 characters.
- Spacing—Include a space on both sides of binary operators. Do not use a space with unary operators. Do not use spaces around parenthesis. Include a blank line before a code block.
- Deprecation—Whenever you deprecate a method or class from an existing release, mark it as deprecated, noting the release in which it was deprecated, and what methods or classes should be used in place of the deprecated items; these records facilitate later code cleanup.
- Header—The file header should include the PVCS tag for revision and log history.

Code Sample 10-1 Header Sample

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

    Copyright (c) 1998-2003 360Commerce, Inc.    All Rights Reserved.

    $Log$

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
package com._360commerce.samples;

// Import only what is used and organize from lowest layer to highest.
import com.ibm.math.BigDecimal;
import com._360commerce.common.utility.Util;

//--------------------------------------------------------------------------
```

```
/**
     This class is a sample class. Its purpose is to illustrate proper
     formatting.
     @version $Revision$
**/
//------------------------------------------------------------------------------
public class Sample extends AbstractSample
implements SampleIfc
{
    // revision number supplied by configuration management tool
    public static String revisionNumber = "$Revision$";
    // This is a sample data member.
    // Use protected access since someone may need to extend your code.
    // Initializing the data is encouraged.
    protected String sampleData = "";

    //----------------------------------------------------------------------
    /**
        Constructs Sample object.
        Include the name of the parameter and its type in the javadoc.
        @param initialData String used to initialize the Sample.
    **/
    //----------------------------------------------------------------------
    public Sample(String initialData)
    {
        sampleData = initialData;
        // Declare variables outside the loop
        int length = sampleData.length();
        BigDecimal[] numberList = new BigDecimal[length];

        // Precede code blocks with blank line and pertinent comment
        for (int i = 0; i < length; i++)
        {
            // Sample wrapping line.
            numberList[i] = someInheritedMethodWithALongName(Util.I_BIG_DECIMAL_ONE,
                                                             sampleData,
                                                             length - i);
        }
    }
}
```

Javadoc

- Make code comments conform to Javadoc standards.
- Include a comment for every code block.
- Document every method’s parameters and return codes, and include a brief statement as to the method’s purpose.

Naming Conventions

Names should not use abbreviations except when they are widely accepted within the domain (such as the customer abbreviation, which is used extensively to distinguish customized code from product code). Additional naming conventions follow:

Table 10-2 Naming Conventions

Element	Description	Example
Package Names	Package names are entirely lower case and should conform to the documented packaging standards.	`com.extendyourstore.packagename` `com.mbs.packagname`
Class Names	Mixed case, starting with a capital letter. Exception classes end in Exception; interface classes end in Ifc; unit tests append Test to the name of the tested class.	`DatabaseException` `DatabaseExceptionTest` `FoundationScreenIfc`
File Names	File names are the same as the name of the class.	`DatabaseException.java`
Method Names	Method names are mixed case, starting with a lowercase letter. Method names are an action verb, where possible. Boolean-valued methods should read like a question, with the verb first. Accessor functions use the prefixes get or set.	`isEmpty()` `hasChildren()` `getAttempt()` `setName()`
Attribute Names	Attribute names are mixed case, starting with a lowercase letter.	`lineItemCount`
Constants	Constants (static final variables) are named using all uppercase letters and underscores.	`final static int NORMAL_SIZE = 400`
EJBs—entity	Use these conventions for entity beans, where 'Transaction' is a name that describes the entity.	`TransactionBean` `TransactionIfc` `TransactionLocal` `TransactionLocalHome` `TransactionRemote` `TransactionHome`
EJBs—session	Use these conventions for session beans, where 'Transaction' is a name that describes the session.	`TransactionService` `TransactionAdapter` `TransactionManager`

SQL Guidelines

The following general guidelines apply when creating SQL code:

- Keep SQL code out of client/UI modules. Such components should not interact with the database directly.
- Table and column names must be no longer than 18 characters.

- Comply with ARTS specifications for new tables and columns. If you are creating something not currently specified by ARTS, strive to follow the ARTS naming conventions and guidelines.
- Document and describe every object, providing both descriptions and default values so that we can maintain an up-to-date data model.
- Consult your data architect when designing new tables and columns.
- Whenever possible, avoid vendor-specific extensions and strive for SQL-92 compliance with your SQL.
- While Sybase-specific extensions are common in the code base, do not introduce currently unused extensions, because they must be ported to the DataFilters and JdbcHelpers for other databases.
- All SQL commands should be uppercase because the DataFilters currently only handle uppercase.
- If database-specific code is used in the source, move it into the JdbcHelpers.
- All JDBC operations classes must be thread-safe.

To avoid errors:

- Pay close attention when cutting and pasting SQL.
- Always place a carriage return at the end of the file.
- Test your SQL before committing.

The subsections that follow describe guidelines for specific database environments.

DB2

Table 10-3 shows examples of potential problems in DB2 SQL code.

Table 10-3 DB2 SQL Code Problems

Problem	Problem Code	Corrected Code

Don’t use quoted integers or unquoted char and varchar values; these cause DB2 to produce errors.	`CREATE TABLE BLAH`
	`(`
	` FIELD1 INTEGER,`
	` FIELD2 CHAR(4)`
	`);`
	`INSERT INTO BLAH (FIELD1,`
	`FIELD2) VALUES (`**`'5', 1020`**`);`
		`CREATE TABLE BLAH`
		`(`
		` FIELD1 INTEGER,`
		` FIELD2 CHAR(4)`
		`);`
		`INSERT INTO BLAH (FIELD1,`
		`FIELD2) VALUES (5, '1020');`

Don’t try to declare a field default as NULL.	`CREATE TABLE BLAH`
	`(`
	` FIELD1 INTEGER `**`NULL`**`,`
	` FIELD2 CHAR(4) NOT NULL`
	`);`
		`CREATE TABLE BLAH`
		`(`
		` FIELD1 INTEGER,`
		` FIELD2 CHAR(4) NOT NULL`
		`);`

MySQL

MySQL does not support sub-selects.

Oracle

Table 10-4 provides some examples of common syntax problems which cause Oracle to produce errors.

Table 10-4 Oracle SQL Code Problems

Problem	Problem Code	Corrected Code
Blank line in code block causes error.	CREATE TABLE BLAH (FIELD1 INTEGER, FIELD2 VARCHAR(20));	CREATE TABLE BLAH (FIELD1 INTEGER, FIELD2 VARCHAR(20));
When using NOT NULL with a default value, NOT NULL must follow the DEFAULT statement.	CREATE TABLE BLAH (FIELD1 INTEGER **NOT NULL** DEFAULT 0, FIELD2 VARCHAR(20));	CREATE TABLE BLAH (FIELD1 INTEGER DEFAULT 0 **NOT NULL**, FIELD2 VARCHAR(20));
In a CREATE or INSERT, do not place a comma after the last item.	CREATE TABLE BLAH (FIELD1 INTEGER, FIELD2 VARCHAR(20),);	CREATE TABLE BLAH (FIELD1 INTEGER, FIELD2 VARCHAR(20));

PostgreSQL

PostgreSQL does not currently support the command `ALTER TABLE BLAH ADD PRIMARY KEY`. However, it does support the standard `CREATE TABLE` command with a `PRIMARY KEY` specified. For this reason, the PostgresqlDataFilter converts SQL of the form shown in Code Sample 10-2 into the standard form shown in Code Sample 10-3.

Code Sample 10-2 SQL Code Before PostgresqlDataFilter Conversion

```
CREATE TABLE BLAH
(
 COL1 INTEGER NOT NULL,
 COL2 INTEGER NOT NULL,
 COL3 INTEGER,
);

ALTER TABLE ADD PRIMARY KEY (COL1, COL2)
```

Code Sample 10-3 SQL Code After PostgresqlDataFilter Conversion

```
CREATE TABLE BLAH
(
 COL1 INTEGER NOT NULL,
 COL2 INTEGER NOT NULL,
 COL3 INTEGER,
 PRIMARY KEY (COL1, COL2)
);
```

Note: There must be a new line and "(" after the `CREATE TABLE` command for the PostgresqlDataFilter's conversion to work, properly formatting the SQL.

Sybase

Sybase does not throw errors if a table element is too large; it truncates the value. If using a VARCHAR(40), use less than 40 characters.

Unit Testing

For details on how to implement unit testing, see separate guidelines on the topic. Some general notes apply:

- Break large methods into smaller, testable units.
- Although unit testing may be difficult for tour scripts, apply it for Java components within the Point-of-Sale code.
- If you add a new item to the codebase, make sure your unit tests prove that the new item can be extended.
- In unit tests, directly create the data/preconditions necessary for the test (in a `setup()` method) and remove them afterwards (in a `teardown()` method). JUnit expects to use these standard methods in running tests.

Architecture and Design Guidelines

This section provides guidelines for making design decisions which are intended to promote a robust architecture.

AntiPatterns

An AntiPattern is a common solution to a problem which results in negative consequences. The name contrasts with the concept of a pattern, a successful solution to a common problem. The following AntiPatterns introduce bugs and reduce the quality of code.

Table 10-5 Common AntiPatterns

Pattern	Description	Solution
Reinvent the Wheel	Sometimes code is developed in an unnecessarily unique way that leads to errors, prolonged debugging time and more difficult maintenance.	The analysis process for new features provides awareness of existing solutions for similar functionality so that you can determine the best solution. There must be a compelling reason to choose a new design when a proven design exists. During development, a similar pattern should be followed in which existing, proven solutions are implemented before new solutions.
Copy-and-paste Programming, classes	When code needs to be reused, it is sometimes copied and pasted instead of using a better method. For example, when a whole class is copied to a new class when the new class could have extended the original class. Another example is when a method is being overridden and the code from the super class is copied and pasted instead of calling the method in the super class.	Use object-oriented techniques when available instead of copying code.
Copy-and-paste Programming, XML	A new element (such as a Site class or an Overlay XML tag) can be started by copying and pasting a similar existing element. Bugs are created when one or more pieces are not updated for the new element. For example, a new screen might have the screen name or prompt text for the old screen.	If you copy an existing element to create a new element, manually verify each piece of the element to ensure that it is correct for the new element.

Table 10-5 Common AntiPatterns

Pattern	Description	Solution
Project Mismanagement/ Common Understanding	A lack of common understanding between managers, Business Analysts, Quality Assurance and developers can lead to missed functionality, incorrect functionality and a larger-than-necessary number of defects. An example of this is when code does not match Functional Requirements, including details like maximum length of fields and dialog message text.	Read the Functional Requirement before you code. If there is disagreement with content, raise an issue with the Product Manager. Before you consider code for the requirement finished, all issues must be resolved and the code must match the requirements.
Stovepipe	Multiple systems within an enterprise are designed independently. The lack of commonality prevents reuse and inhibits interoperability between systems. For example, a change to till reconcile in Back Office may not consider the impact on Point-of-Sale. Another example is a making change to a field in the 360Store database for a Back Office feature without handling the Point-of-Sale effects.	Coordinate technologies across applications at several levels. Define basic standards in infrastructures for the suite of products. Only mission-specific functions should be created independently of the other applications within the suite.

Designing for Extension

This section defines how to code product features so that they may be easily extended. It is important that developers on customer projects whose code may be rolled back into the base product follow these standards as well as the guidelines in Chapter 9, “Extension Guidelines.”

- Separate external constants such as database table and column names, JMS queue names, port numbers from the rest of the code. Store them in (in order of preference):
  - Configuration files
  - Deployment descriptors
  - “Constant” classes/interfaces
- Make sure the SQL code included in a component does not touch tables not directly owned by that component.
- Consider designing so that any fine grained operation within the larger context of a coarse grain operation can be factored out in a separate “algorithm” class, so that it can be replaced without reworking the entire activity flow of the larger operation.

Common Frameworks

This section provides guidelines which are common to the 360Commerce applications.

Internationalization

The following are some general guidelines for maintaining an internationalized code base which can be localized when needed. Refer to other documents for detailed instructions on these issues.

- All displayed text must be referenced from the appropriate resource bundle and properties file, so that the text can be changed when needed.
- Numbers, currency, and amounts must be displayed using Java internationalization conventions, so that appropriate symbols and number dividers can be used for the current locale.
- Formats and conventions related to dates, times and calendars are locale sensitive. All the date, time and calendar related operations must use DateFormat, SimpleDateFormat and Calendar classes, instead of the Date class. Remove hardcoded dates (mm/dd/yyyy, etc). Use the formats available as part of the DateFormat class.
- Properties in the `application.properties` file specify default and supported locales:
  ```
  default_locale=en_US
  supported_locales=en_US,fr_CA,en_CA
  ```
- Help files for new screens must be created in the appropriate locale directory, and `pos\config\ui\help\helpscreens.properties` must be updated.
- Display database driven locale sensitive data according to the current locale.

Logging

360Commerce's systems use Log4J for logging. When writing log commands, use the following guidelines:

- Use calls to Log4J rather than System.out from the beginning of your development. Unlike System.out, Log4J calls are naturally written to a file, and can be suppressed when desired.
- Log exceptions where you catch them, unless you are going to rethrow them. This is preserves the context of the exceptions and helps reduce duplicate exception reporting.
- Logging uses few CPU cycles, so use debugging statements freely.
- Use the correct logging level:
  - FATAL—crashing exceptions
  - ERROR—nonfatal, unhandled exceptions (there should be few of these)
  - INFO—lifecyle/heartbeat information
  - DEBUG—information for debugging purposes

The following sections provide additional information on guarding code, when to log, and how to write log messages.

Guarding Code

Testing shows that logging takes up very little of a system's CPU resources. However, if a single call to your formatter is abnormally expensive (stack traces, database access, network IO, large data manipulations, etc.), you can use Boolean methods provided in the Logger class for each level to determine whether you have that level (or better) currently enabled; Jakarta calls this a code guard:

Code Sample 10-4 Wrapping Code in a Code Guard

```
    if (log.isDebugEnabled()) {
        log.debug(MassiveSlowStringGenerator().message());
    }
```

An interesting use of code guards, however, is to enable debug-only code, instead of using a DEBUG flag. Using Log4J to maintain this functionality lets you adjust it at runtime by manipulating Log4J configurations.

For instance, you can use code guards to simply switch graphics contexts in your custom swing component:

Code Sample 10-5 Switching Graphics Contexts via a Logging Level Test

```
protected void paintComponent(Graphics g) {

    if (log.isDebugEnabled()) {
        g = new DebugGraphics(g, this);
    }

    g.drawString("foo", 0, 0);
}
```

When to Log

There are three main cases for logging:

- **Exceptions**—Should be logged at an error or fatal level.
- **Heartbeat/Lifecycle**—For monitoring the application; helps to make unseen events clear. Use the info level for these events.
- **Debug**—Code is usually littered with these when you are first trying to get a class to run. If you use System.out, you have to go back later and remove them. With Log4J, you can simply raise the log level. Furthermore, if problems pop up in the field, you can lower the logging level and access them.

Writing Log Messages

When Log4J is being used, any log message might be seen by a user, so the messages should be written with users in mind. Cute, cryptic, or rude messages are inappropriate. The following sections provide additional guidelines for specific types of log messages.

Exception Messages

A log message should have enough information to give the user a good shot at understanding and fixing the problem. Poor logging messages say something opaque like "load failed."

Take this piece of code:

```
try {
    File file = new File(fileName);
    Document doc = builder.parse(file);

    NodeList nl = doc.getElementsByTagName("molecule");
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        // something here
    }

} catch {
   // see below
}
```

and these two ways of logging exceptions:

```
} catch (Exception e){
    log.debug("Could not load XML");
}
```

```
} catch (IOException e){
    log.error("Problem reading file " + fileName, e);
} catch (DOMException e){
    log.error("Error parsing XML in file " + fileName, e);
} catch (SAXException e){
    log.error("Error parsing XML in file " + fileName, e);
}
```

In the first case, you get an error that just tells you something went wrong. In the second case, you're given slightly more context around the error, and you get that key piece of data: the filename.

The log lets you augment the message in the exception itself. Ideally, with the messages, the stack trace, and type of exception, you'll have enough to be able to reproduce the problem at debug time. Given that, the message can be reasonably verbose.

For instance, the `fail()` method in JUnit really just throws an exception, and whatever message you pass to it is in effect logging. It’s useful to construct messages that contain a great deal of information about what you are looking for:

Code Sample 10-6 JUnit

```
if (! list.contains(testObj)) {
    StringBuffer buf = new StringBuffer();
    buf.append("Could not find object " + testObj + " in list.\n");
    buf.append("List contains: ");
    for (int i = 0; i < list.size(); i++) {
        if (i > 0) {
             buf.append(",");
        }
        buf.append(list.get(i));
    }
    fail(buf.toString());
}
```

Heartbeat or Lifecycle Messages

The log message here should succinctly display what portion of the lifecyle is occurring (login, request, loading, etc.) and what apparatus is doing it (is it a particular EJB, are there multiple servers running, etc.)

These message should be fairly terse, since you expect them to be running all the time.

Debug Messages

Debug statements are going to be your first insight into a problem with the running code, so having enough, of the right kind, is important.

These statements are usually either of an intra-method-lifecycle variety:

```
log.debug("Loading file");
File file = new File(fileName);
log.debug("loaded.  Parsing...");
Document doc = builder.parse(file);
log.debug("Creating objects");
for (int i ...
```

or of the variable-inspection variety:

```
log.debug("File name is " + fileName);
log.debug("root is null: " + (root == null));
log.debug("object is at index " + list.indexOf(obj));
```

Exception Handling

The key guidelines for exception handling are:

- Handle the exceptions that you can (FileNotFound, etc.)
- Fail fast if you can’t handle an exception
- Log every exception with Log4J, even when first writing the class, unless you are rethrowing the exception
- Include enough information in the log message to give the user or developer a fighting chance at knowing what went wrong
- Nest the original exception if you rethrow one

Types of Exceptions

The EJB specification divides exceptions into the following categories:

JVM Exceptions—You cannot recover from these; when one is thrown, it’s because the JVM has entered a kernel panic state that the application cannot be expected to recover from. A common example is an Out of Memory error.

System Exceptions—Similar to JVM exceptions, these are generally, though not always, “non-recoverable” exceptions, also described as “unexpected” exceptions. The canonical example here is NullPointerException. The idea is that if a value is null, often you don't know what you should do. If you can simply report back to your calling method that you got a null value, do that. If you cannot gracefully recover, say from an IndexOutOfBoundsException, treat it as a system exception and fail fast.

Application Exceptions—These are the expected exceptions, usually defined by specific application domains. It is useful to think of these in terms of recoverability. A FileNotFoundException is sometimes easy to rectify by simply asking the user for another file name. But something that's application specific, like JDOMException, may still not be recoverable. The application can recognize that the XML it is receiving is malformed, but it may still not be able to do anything about it.

Avoid java.lang.Exception

Avoid throwing the generic Exception; choose a more specific (but standard) exception.

Avoid Custom Exceptions

Custom exceptions are rarely needed. The specific type of exception thrown is rarely important; don't create a custom exception if there is a problem with the formatting of a string (ApplicationFormatttingException) instead of reusing IllegalArgumentException.

The best case for writing a custom exception is if you can provide additional information to the caller which is useful for recovering from the exception or fixing the problem. For example, the JPOSExceptions can report problems with the physical device. An XML exception could have line number information embedded in it, allowing the user to easily detect where the problem is. Or, you could subclass NullPointer with a little debugging magic to tell the user what method of variable is null.

Catching Exceptions

The following sections provide guidelines on catching exceptions.

Keep the Try Block Short

The following example, from a networking testing application, shows a loop that was expected to require approximately 30 seconds to execute (since it calls `sleep(3000)` ten times):

Code Sample 10-7 Network Test

```
    for (int i = 0; i < 10; i++) {
        try {
            System.out.println("Thread " + Thread.currentThread().getName() + " requesting number " +
i);
            URLConnection con = myUrl.openConnection();
            con.getContent();
            Thread.sleep(3000);
        } catch (Exception e) {
            log.error("Error getting connection or content", e);
        }
    }
```

The initial expectation was for this loop to take approximately 30 seconds, since the `sleep(3000)` would be called ten times. Suppose, however, that `con.getContent()` throws an IOException. The loop then skips the `sleep()` call entirely, finishing in 6 seconds. A better way to write this is to move the `sleep()` call outside of the `try` block, ensuring that it is executed:

Code Sample 10-8 Network Test with Shortened Try Block

```
    for (int i = 0; i < 10; i++) {
        try {
            System.out.println("Thread " + Thread.currentThread().getName() + " requesting number " +
i);
            URLConnection con = myUrl.openConnection();
            con.getContent();
        } catch (Exception e) {
            log.error("Error getting connection or content", e);
        }
        Thread.sleep(3000);
    }
```

Avoid Throwing New Exceptions

When you catch an exception, then throw a new one in its place, you replace the context of where it was thrown with the context of where it was caught.

A slightly better way is to throw a wrapped exception:

Code Sample 10-9 Wrapped Exception

```
try {
    Class k1 = Class.forName(firstClass);
    Class k2 = Class.forName(secondClass);
    Object o1 = k1.newInstance();
    Object o2 = k2.newInstance();

} catch (Exception e) {
   throw new MyApplicationException(e);
}
```

However, the onus is still on the user to call `getCause()` to see what the real cause was. This makes most sense in an RMI type environment, where you need to tunnel an exception back to the calling methods.

The better way than throwing a wrapped exception is to simply declare that your method throws the exception, and let the caller figure it out:

Code Sample 10-10 Declaring an Exception

```
public void buildClasses(String firstName, String secondName)
    throws InstantiationException, ... {

    Class k1 = Class.forName(firstClass);
    Class k2 = Class.forName(secondClass);
    Object o1 = k1.newInstance();
    Object o2 = k2.newInstance();
}
```

However, there may be times when you want to deal with some cleanup code and then rethrow an exception:

Code Sample 10-11 Clean Up First, then Rethrow Exception

```
try {
    someOperation();
} catch (Exception e) {
    someCleanUp();
    throw e;
}
```

Catching Specific Exceptions

There are various exceptions for a reason: so you can precisely identify what happened by the type of exception thrown. If you just catch Exception (rather than, say, ClassCastException), you hide information from the user. However, methods should not generally try to catch every type of exception. The rule of thumb is the related to the fail-fast/recover rule: catch as many different exceptions as you are going to handle.

Favor a Switch over Code Duplication

The syntax of `try` and `catch` makes code reuse difficult, especially if you try to catch at a granular level. If you want to execute some code specific to a certain exception, and some code in common, you're left with either duplicating the code in two catch blocks, or using a switch-like procedure. The switch-like procedure, shown below, is preferred because it avoids code duplication:

Code Sample 10-12 Using a Switch to Execute Code Specific to an Exception

```
try{
    // some code here that throws Exceptions...
} catch (Exception e) {
    if (e instanceof LegalException) {
        callPolice((LegalException) e);
    } else if (e instanceof ReactorException) {
        shutdownReactor();
    }
    logException(e);
    mailException(e);
    haltPlant(e);
}
```

This example is preferred, in these relatively rare cases, to using multiple catch blocks:

Code Sample 10-13 Using Multiple Catch Blocks Causes Duplicate Code

```
    try{
        // some code here that throws Exceptions...
    } catch (LegalException e) {
        callPolice(e);
        logException(e);
        mailException(e);
        haltPlant(e);
    } catch (ReactorException e) {
        shutdownReactor();
        logException(e);
        mailException(e);
        haltPlant(e);
    }
```

Exceptions tend to be the backwater of the code; requiring a maintenance developer, even yourself, to remember to update the duplicate sections of separate catch blocks is a recipe for future errors.

CHAPTER 11

POINT-OF-SALE DEVELOPMENT STANDARDS

The following standards specific to the Point-of-Sale architecture have been adopted by 360Commerce product and service development teams. These standards are intended to reduce bugs and increase the quality of the code.

Screen Design and User Interface Guidelines

- Avoid creating new screen beans and screen models for every new screen. Look for ways to reuse existing or generic beans, such as the Data Input Bean, to avoid complicating the code base.
- For detailed user interface standards, see the *UI Guidelines* document, found in the `_resources` directory provided with your documentation.

Tour Framework

This section includes general guidelines as well as subsections on specific tour components.

Tour Architectural Guidelines

Consult these guidelines when making architecture decisions in tour framework designs.

- Services—When designing services, consider their size and reusability. Services that are overlarge create additional work when a portion must be extended.
- Utility Manager—Put methods used by multiple services in this manager so they can be easily extended.
- If the reusable behavior contains flow-dependent behavior, then it is best implemented as a Site and the Site action can be reused within a Service or across Services.
- Large bodies of reusable behavior can be implemented as Managers and Technicians. This pattern is especially useful if the user might offload the processing to a separate CPU.

General Tour Guidelines

- Code that uses bus resources must reside in a Site action, Lane action, Signal or Shuttle.
- Never mail a letter from a Road. *This causes unpredictable results.*
- Never define local data in a Site, Aisle, Road or Signal. Local data is not guaranteed when processing across multiple tiers. Sites and Lanes must be stateless. This is the purpose of Cargo.
- Traffic Signals should not modify Cargo. Signals should only be used to evaluate a condition as true or false. Anything else is a side effect, reducing the maintainability of the system.
- Never implement just one Signal. Always implement Signals when there is more than one Road that responds to the same letter, or when there is an Aisle and a Road that respond to the same letter. See "Signals" on page 11-5.
- Send letters at the end of methods. If the choice of which letter to send depends on conditions which occur during the method, store the method name and mail it at the end of the method.
- Do not mail letters from `depart()` and `undo()` in Sites, `backup()` and `traverse()` in Roads, `roadClear()` in Signals, and `load()` and `unload()` in Shuttles. Letters can be mailed from `traverse()` in Aisles.
- Define Shuttles in the calling Service package. If they are reusable Shuttles, define them in a common package.

Use the following naming conventions for Tour components:

Table 11-1 Tour Naming Conventions

Element	Description	Example
Service	description of the related functionality	Login
Site element	VerbNoun—indicating the action taking place at the Site	EnterID
Site class	The same as the Site name, with Site as a suffix	EnterIDSite.java
Road element	NounVerb—indicating the event that caused the Road to be taken	IDEntered
Road class	The same as the Road name, with Road as a suffix	IDEnteredRoad.java
Aisle element	NounVerb- indicating the event that caused the Aisle to be taken	PasswordEntered
Aisle class	The same as the Aisle name, with Aisle as a suffix	PasswordEnteredAisle.java
Cargo	ServiceNameCargo	LoginCargo.java

Table 11-1 Tour Naming Conventions

Element	Description	Example
Letter	One word action name indicating the event; see list defined in `commonLetterIfc.java`	Success Failure Continue Next Cancel OK Retry Invalid Add Yes No Undo Done
Transfer Station element	NestedServiceNameStation	FindCustomerStation
Shuttle class	NestedServiceNameLaunchShuttle NestedServiceNameReturnShuttle	FindCustomerLaunchShuttle.java FindCustomerReturnShuttle.java
Traffic Signal class	IsCondition.java—indicating the condition being tested	IsAuthRequiredSignal.java

Foundation

- The best reuse in the Foundation engine takes place at the Service level. Sites require extra thought because they can affect flow. Lane actions can be reused without flow implications. Signals and Shuttles are very well suited to reuse especially when interfaces are developed for accessing Cargo.
- If validation and database lookup are coded in Aisles, they may be good candidates for reuse in several Sites as well as in multiple Services.
- All component pieces need to be designed with care for reuse: they must be context insensitive or must do a lot of checking to make sure that the managers they access exist for the bus that is active, the Cargo contains the data they need, etc.
- Trying to maximize reuse can result in confusing code with too many discrete parts. If the reusable unit consists of one or two lines of code, consider whether there is sufficient payoff in reusing the unit of code. If the code contains a complex calculation that is subject to change over time, then isolating this logic in one place may be well worth the effort.

Tours and Services

- There is often a one-to-one mapping between a Use Case and a Service. The Service should provide the best opportunity for reuse. If you design for reuse, it should be focused at the Service level. This is where you get your best return on investment.

- Maintenance is a matter of choosing a style and implementing it consistently within a Service and sometimes within an entire application. When you are comfortable with how TourCam works, maintaining TourCam Services is easy. Maintenance is more difficult in general for TourCam Services, since these Services are more complex. However, the simulation feature in Tour Guide helps with this process.
- Aisles help reduce the total number of Sites in a Service, but they may be harder to see because they are contained within a Site.
- When making choices, give making an application as consistent and easy to maintain as possible the top priority.
- Consider the performance costs of using TourCam or creating additional Sites when designing a Service.
- A Service can often be simplified by reducing the number of individual Sites. You can do this by using Aisles to replace Sites; Sites with one exit Road can be good candidates, and Aisles are good candidates for reuse. However, Aisles are less visible than Roads.

Sites

- Reusing a Site has flow implications. Site classes can be reused whenever the exit conditions are identical. Reusable Sites should be packaged in a common package as opposed to one of the packages that use them. A reusable Site must refer to a reusable Cargo or a common Cargo interface.
- Treat the sending of a letter like a return code: put it at the end of your `arrive()` or `traverse()` method. Sending letters in the middle of the `arrive()` method may cause duplicate letters (with unpredictable results), or no letters (with no results).
- Do not try to store state information in instance variables. Pass in state information through arguments.
- Do not put a lot of functionality in `arrive()`, `traverse()` methods. Decompose them into logical methods that each have one job. For methods not called from outside the package, protect the methods.

Managers and Technicians

- There is a high degree of reuse of Managers and Technicians across the applications. For example, the DataTransactions and DataActions are reusable. By design, it is the DataOperations that change with different database implementations. The UIManager and UITechnician expect a lot of reuse of beans, adapters, and specification objects. In fact, the UISubsystem looks in the UI Script for most of the configuration information that effects changes in screen layout, bean interactions and even bean composition.
- Utility methods can be useful for capturing behavior that is used by many Services, but does not lend itself to Site or Aisle behavior. Put Utility methods in a UtilityManager so they can be easily extended. The Point-of-Sale application contains an example of this called the POSUtilityManager. Service developers can access these methods through the POSUtilityManagerIfc. The UtilityManager and UtilityManagerIfc classes can be extended and the new class is specified through the Conduit Script. For general-purpose behavior that can be called from a Site, Lane, or even from a Signal, use utility methods to capture the common reusable behavior rather than extending a common Site.
- Large bodies of reusable behavior can be implemented as Managers and Technicians. This pattern is especially useful if the user might off-load the processing to a separate CPU.

Roads

It is sometimes useful to define multiple Roads from an origin Site to the same destination if they capture different Road traversal conditions.

Do not trap and change the name of a letter just to reduce the number of Roads in a Service. This is a poor use of system resources and also hides useful information from the reader of the Tour Script. Do not rename letters except as noted in “Renaming Letters” on page 11-6.

For example, the Return Transaction Service has two Roads with the same origin (LookupItem) and the same destination (EnterReturnItemInformation), but the letters that invoke these two Roads are different.

The use of Road actions is dependent on a number of factors: use of TourCam, developer conventions for an application, number of classes generated, and maintainability.

Use Road actions for outcome-specific behavior. If you need to store some data in Cargo on the sending of a specific letter, do the Cargo storage in the `traverse()` method of the Road that is associated with that letter. If the data must be stored in Cargo before leaving a Site, put the logic in the Site’s `depart()` method. Code in a Site or Aisle’s `depart()` method should not check to see what letter was sent before taking an action; use a Road in that case.

Aisles

Aisles are used to implement behavior that occurs within a Site. When there is interaction with an external source (e.g. user, database) use a Site. When you are doing business validation which may keep you in the same screen, use an Aisle.

While it makes sense to create Roads without corresponding Road actions, Aisles are useless without an Aisle action. The important thing about an Aisle is that it is not part of a transition from one Site to another, so the only code that gets executed in an Aisle is the `traverse()` method. The `arrive()` and `depart()` methods are never executed on a Site when an Aisle is processed. The Aisle can initiate an action that causes a transition to another Site, but it cannot transition itself.

Aisle actions can be used to validate data, compute values, provide looping behavior, and do database lookups. Aisle actions are useful for capturing repeatable behavior that can occur while the bus is still in a Site.

For example, suppose you define a Site that gathers data from the user. The data validation is implemented as an Aisle. Because it is an Aisle, the user can repeat the process of entering data, validating, and re-entering until the data is correct, with little system overhead. The Aisle behavior can be triggered over and over without calling the `arrive()` method on the Site (a Road back to the Site calls the `arrive()` method).

Aisles are also useful for looping through a list of items when each item may require error handling. This is done by placing the loop index in the Cargo.

Signals

You cannot use a signal alone; they must be used in groups of two or more. If there is more than one Lane that responds to the same letter, each Lane must implement a Signal. The logic in the Signals must be mutually exclusive; there should be only one valid Road that can be traversed at any time; otherwise, unexpected (and difficult to debug) behavior could occur.

When there are more than two Signals, each of the Signals should evaluate in such a way that only one Signal is green at any given time. But the presence of more than two Signals should raise a red flag. Track down the source of the following issues; determine if the UI or other letter generator needs to be sending more unique letters.

- Why are there so many Signals?
- What are they checking?
- Is the same letter being sent for many different conditions?

Use a Signal only to decide which road to take when you could go to two different places (such as Sites) with the same Letter, based on Cargo information. It should not be used to update cargo. The road you take after making a decision at the Signal should do the updating.

Choosing among Sites, Aisles, and Signals

There are many times when an Aisle can do the same work as a Site. Sometimes a Signal can contain behavior that could be implemented in an Aisle. Sometimes a separate Service does the work that was once a Site if the Site needs to be reused or becomes too complicated. Consult the guidelines for your application development team in order to be consistent with the rest of your team.

If you have the following customer requirement:

> Display a UI screen that gathers search criteria to be used in a database lookup (for example, customer lookup). After the user enters the data, validate the data. Once the data has been validated, do the database lookup.

you have the following design choices:

- Implement as separate Sites and take advantage of TourCam to back up when the data is invalid or database lookup results are not correct.
- Implement as one Site with Aisles that do the validation and lookup.

The database lookup may result in a success or failure letter whether it is coded as a Site or an Aisle. When using an Aisle for database lookup, the failure letter triggers another Aisle that could display an error message but allow the user to re-enter the data and retry the lookup. This can occur without exiting the original Site. When using a Site, the failure condition can trigger a flow change to back up through the lookup Site back to the data entry Site.

If the validation and database lookup are coded in Aisles, they may be good candidates for reuse in several Sites as well as in multiple Services. Reusing the Site is also possible, especially if the TourCam's ability to back up to the last indexed Site is used. But there may be more considerations involving flow when trying to reuse a Site.

Renaming Letters

Use the following guidelines when deciding whether to rename letters:

- Do rename Letters when the application developer does not have power over the Letter that is mailed and there is more than one event associated with a single Letter.

  For example: a single Letter is sent from a button on the UI (such as dialog box OK), but the content of the retrieved data associated with the UI signals a different event notification (such as error message notification).
- Do rename Letters when a common exit Letter from a nested Service is needed.
- Don't rename Letters to reduce the number of Roads in a Service.

Shuttles

If you are creating a sub-tour (i.e. a tour called from other tours via a Station) from scratch, use *only* the following final letters:

- Success
- Failure
- Cancel
- Undo

If you need to provide a reason for a Failure or need to return data to the calling service on a Success, use the Return Shuttle to update the calling service's cargo. Do not use letters to reflect sub-tour results.

Within the Tour Framework, Shuttles are used to transfer data in and out of Services. Shuttles are good candidates for reuse given a common Cargo interface.

Table 11-1 Shuttles

Shuttle Type	**Launch Shuttle**	**Return Shuttle**
Description	Used to send parameter data to a sub-service	Used to return data to the parent service.
Methods	`load()`—can only see the parent Service's Cargo `unload()`—can only see the sub-service's Cargo	`load()`—can only see the sub-service's Cargo `unload()`—can only see the parent service's Cargo

Cargo

All Cargo classes should implement the `CargoIfc` interface.

Log Entry Format

This section describes the format and layout of log entries for the Point-of-Sale application.

Log Entry Description

Log entries adhere to the following format:

```
LLLLL yyyyy-mm-dd hh:mm:ss,ttt bbbbbbb (<classname>):
      [<classname>.<methodname>(<filename>:<linenumber>)]
      <Log entry content>
```

Fixed Length Header

The entry begins with a fixed length record header (38 bytes) that adheres to the following layout:

```
LLLLL yyyyy-mm-dd hh:mm:ss,ttt bbbbbbb
12345678901234567890123456789012345678
```

LLLLL is the log message level and consists one of the substrings in the following table:

Table 11.2 Log Message Level

Log Message Level	Description
ERROR	Highest severity entry; critical
WARN	Application warning; serious
INFO	For information only
DEBUG	For developer use (not displayed by default application configuration

yyyy-`mm-dd` is the date.

`hh:mm:ss,ttt bbbbbbb` is the time stamp of the entry, comprised of the sub-fields described in the following table:

Table 11-3

Field	Description
hh	Time of entry in hours, in 24-hour format
mm	Minutes past the full hour
ss	Seconds past the last full minute
ttt	Milliseconds past the last full second
bbbbbbb	Milliseconds since the application was started. Left justified and blank filled on the right, out to 7 places.

Additional Logging info

The fixed length record header is followed by a blank space followed by the parenthesized, fully qualified class name of the logging entity followed by a colon followed by a carriage return/line feed pair.

```
(<classname>):<cr><lf>
```

The next line in a log entry begins with 6 blank spaces and a square-bracketed sequence containing the following information: `<classname>.<methodname>(<filename>:<linenumber>)`

Parentheses are included in the sequence. This sequence reflects the fully qualified name of the method invoking the logging action and the source line number in the file where the logging call was made.

The next line(s) in a log entry are the log entry content. The content is comprised of freeform text supplied by the calling routine. The content reflected in the freeform text may be multiple lines in length.

The next log entry is delineated with another 38 byte fixed length header beginning in column one of the text log file.

Example Log Entry

```
INFO  2004-09-02 11:12:41,253 23697
(main:com.extendyourstore.foundation.manager.gui.DefaultBeanConfigurator):

[com.extendyourstore.foundation.manager.gui.DefaultBeanConfigurator.applyProperties(DefaultBeanConfigur
ator.java:198)]
      Applying property cachingScheme to Class: DialogBean (Revision 1.9) @12076742
```