IBM WebSphere Payment Manager

Cassette Developer Cookbook

Lance D Bader
P.O. Box 12195
4025 South Miami Boulevard
Research Triangle Park, NC 27709
ldbader@us.ibm.com
Lance Bader/Raleigh/IBM@IBMUS
919-254-4461
T/L 444-4461

Revised on: 09/10/99 4:24 PM
Document Identifier: CookBook.doc
Send Comments to:

Table of Contents

Stage 1: Choose a Cassette Name

Although simple, this stage has wide ranging effects.

- The merchant software passes the cassette name as a parameter on
  - ReceivePayment
  - AcceptPayment
  - BatchOpen
  - CassetteControl
  - CreateAccount
  - CreateMerchantCassetteObject
  - CreatePaySystem
  - CreateSystemCassetteObject
  - DeleteAccount
  - DeleteMerchantCassetteObject
  - DeletePaySystem
  - DeleteSystemCassetteObject
  - ModifyAccount
  - ModifyCassette
  - ModifyMerchantCassetteObject
  - ModifyPaySystem
  - ModifySystemCassetteObject
  - QueryAccounts
  - QueryBatches
  - QueryCassette
  - QueryCredits
  - QueryOrders
  - QueryPayments
  - QueryPaySystems
  - ReceivePayment

  Each of these verbs has a parameter named PaymentType or a parameter named CassetteName whose value is a character string that contains the name of your cassette. The Payment Manager framework will use this string to find your cassette class when routing API requests.

- The framework uses the name to construct the fully qualified name of your cassette class when the class is dynamically loaded during the initialization.

- The framework uses the name to construct the fully qualified name of your cassette query class when the class is dynamically loaded during initialization.

- The framework uses the name to find the properties file used to construct your cassette ResourceBundle.

- The framework uses the name to find the file containing the XML document used to display and modify cassette dependent information on the user interface panels. This document contains Payment Server Presentation Language (PSPL) tags that are interpreted by the Payment Manager user interface.

- The cassette uses the name to specify the resource bundle to be used when logging messages from the resource bundle or when retrieving data from the resource bundle.

- The cassette uses the name to identify the source of trace messages.

Important: It is important to remember that DB2 identifiers (table names, view names, column names, and so on) are limited to 18 characters. If you choose a cassette name that is longer than 8 characters, you will need to manually abbreviate some of the table names that are automatically generated for you by the cassette exercise.

This naming convention is based on Java rules to avoid name conflicts. These rules avoid naming conflicts by utilizing the Internet domain name assigned to your company. The Java system is case sensitive and, by convention, package names are always lower case and class names are mixed case with the first character capitalized.

When applying the cassette naming conventions, there are two cases:

1. the cassette is developed by IBM
2. the cassette is **not** developed by IBM.

If the Cassette Is Developed By IBM

Assume that you are developing a cassette for a payment protocol called FastPay. Then,

- Your company name is IBM.
- Your cassette name is FastPay (mixed case). Merchants will use this case sensitive value to select your cassette.
- Your cassette's .JAR file will be named FastPayCassette.jar.
- Your Java package name is com.ibm.etill.fastpaycassette (lower case).
- Your development directory structure will be D:\!PMCassette\com\ibm\etill\fastpaycassette where D:\!PMCassette is, by convention, the root directory
- The name of your properties file is FastPay.properties (mixed case.
- The name of your file containing the XML document used to display and modify cassette dependent information on the user interface panels is FastPay.en.pspl.
- The name of your cassette class is FastPayCassette (mixed case) or ,as a fully qualified name, com.ibm.etill.fastpaycassette.FastPayCassette.
- The name of your cassette query class is FastPayCassetteQuery (mixed case) or, as a fully qualified name, com.ibm.etill.fastpaycassette.FastPayCassetteQuery.

If the Cassette Is Developed By A Company Other than IBM

Assume that Your Company name is A Better CashRegister and your World Wide Web site for your company is www.abc.com and assume that you are developing a cassette for a payment protocol called FastPay. Then,

- Your company name is ABC.
- Your cassette name is FastPay (mixed case). Merchants will use this case sensitive value to select your cassette.
- Your cassette's .JAR file will be named FastPayCassette.jar.
- Your Java package name is com.abc.ibmetill.fastpaycassette (lower case).

- Your development directory structure will be D:\!PMCassette\com\abc\ibmetill\fastpaycassette where D:\!PMCassette is, by convention the root directory
- The name of your properties file is FastPay.properties (mixed case)
- The name of your file containing the XML document used to display and modify cassette dependent information on the user interface panels is FastPay.en.pspl.
- The name of your cassette class is FastPayCassette (mixed case) or, as a fully qualified name, com.abc.ibmetill.fastpaycassette.FastPayCassette.
- The name of your cassette query class is FastPayCassetteQuery (mixed case) or, as a fully qualified name, com.abc.ibmetill.fastpaycassette.FastPayCassetteQuery.

Stage 2: Obtain and Install Development Materials

In this chapter, you will obtain and install all the required components for your cassette development project.

A Java Development Kit for Version 1.1.8

By definition, a cassette for the IBM WebSphere Payment Manager is developed in Java. Even if you use a different language for developing the payment protocols used by your cassette, the cassette itself must be written in Java in order to plug into the Payment Manager framework. It is important to note that the IBM WebSphere Payment Manager does not support Java version 1.2. Other Java 1.1 versions better than Version 1.1.7 are supported, but the instructions in this cookbook assume that you are using Version 1.1.8.

Follow the directions in **Appendix A** to obtain and properly install a Java development kit for cassette development.

IBM WebSphere Payment Manager

In order to develop and test a cassette, you will need to obtain and install the IBM WebSphere Payment Manager. It provides both the class files for your development plus the framework necessary to exercise your cassette. Furthermore, it includes the database manager, the web server, and the servlet engine required to create a Payment Manager installation.

To obtain your development copy of the IBM WebSphere Payment Manager, first register to be a member of IBM's Partnerworld for Developers Program.

1. Start your favorite browser and go to http://www.developer.ibm.com/member/register/register.html .

2. Using one of the following methods, register to become a member **of IBM's Partnerworld for Developers** program. Be sure to select the **Internet specialty** when registering.

   - If you are an IBM employee, make sure you are connected to IBM's intranet, press the **Register an IBM Employee** link, and follow the directions.
   - If your company has not registered, press the **Commercial Membership**button and follow the directions.
   - If your company has already registered, obtain your company's member ID from the person in your company responsible for keeping it, press the **Individual Membership**button, and follow the directions.

3. Record your member ID and password.

Once you are a member of IBM's Partnerworld for Developers program, order your software from the IBM Software Mall. Generally, the first copy of any IBM software product sold in this mall can be purchased for the cost of the media plus a modest charge for shipping and handling. Additional copies are typically 50% of the list price.

1. Start your favorite browser and go to http://www.developer.ibm.com/welcome/softmall.html .

2. On the right hand side of the panel, there is a navigation pane. Look for a link to the **Software Mall** under the heading **Login Required**. Follow that link.

3. Enter the membership ID and password you recorded when you registered. Press **Enter**.

4. On the resulting panel, find the link to the Internet products. Follow that link.

5. On the resulting panel, find the products you wish to order and add them to the cart. For developing your cassette, you will only need the IBM WebSphere Payment Manager for Windows/NT. To find this product, you may need to use the **All** tab and use the **Next** button at the very bottom of the panel. **Note:** This cookbook assumes that you will be developing your cassette on a Windows NT workstation. Be sure to order the Payment Manager for Windows NT. There is no need to order any of the Payment Manager cassettes.

6. Press **View Contents** to check your order. Correct the order if necessary and then press **Checkout** and follow the directions to complete your order.

Once you have received your copy of the IBM WebSphere Payment Manager for Windows/NT, follow the directions **Appendix B** to install the product on your workstation.

IBM DB2 Universal Database (UDB)

The IBM DB2 Universal Database is required by the IBM WebSphere Payment Manager and some of the tools provided by this cookbook. The database is used to store configuration information and transaction data. Although there are other database managers supported by the Payment Manager, the instructions in this cookbook assume that you are using IBM DB2.

Conveniently, the IBM WebSphere Payment Manager includes DB2. For cassette development, there is no need to obtain this product separately. This database manager will be installed properly if you follow the installation directions in **Appendix B**.

IBM HTTP Server

The IBM HTTP Server is required to run the Payment Manager user interface as well as the simulated merchant environment provided by this cookbook. Both static web pages and Java servlets are used to configure and test your cassette during development. Although other web servers could be used, the instructions in this cookbook assume that you are using the IBM HTTP Server.

Conveniently, the IBM WebSphere Payment Manager includes the HTTP Server. For cassette development, there is no need to obtain this product separately. This web server will be installed properly if you follow the installation directions in **Appendix B**.

IBM WebSphere Application Server

The IBM WebSphere Application Server is required by the IBM WebSphere Payment Manager and some of the tools provided by this cookbook. The Payment Manager uses Java servlets to service merchant commands and to build dynamic web pages for the user interface. Furthermore, this cookbook includes servlets that help you test your cassette during development.

Conveniently, the IBM WebSphere Payment Manager includes the Application Server. For cassette development, there is no need to obtain this product separately. This Application Server will be installed properly if you follow the installation directions in **Appendix B** .

IBM WebSphere Payment Manager Service Packs

At this instant, it is very important to use Version 2.1.4 or better while developing your cassette. Version 2.1.4 contains important defect fixes. In addition, it contains important enhancements, including

- The new AliasValidationItem class which makes it easy for a cassette to support the protocol data keywords used by the IBM Payment Server Version 1.2 and so provide a smooth migration for merchants who have business systems written to support this early version of the WebSphere Payment Manager
- New protocol data keywords for Version 1.2 compatibility as well as new keywords for billing address and shipping address
- New secondary return codes for conditions that should be reported the same way by all cassettes.

As time goes on, I expect defects to be removed and small enhancements (for example, new protocol data keywords and new return codes) to be added. To benefit from these changes, you will need to obtain and install a service pack.

To obtain a service pack, start your favorite browser and navigate to **http://www.ibm.com/software/webservers/paymgr/support/sbull.html** . Follow the directions to find, download, and install the desired service pack.

Integrated Development Environment

Strictly speaking, an Integrated Development Environment (IDE) is not required to develop a cassette. However, your fundamental task is to develop the Java classes used to implement your Payment Manager cassette and your productivity can be enhanced with a good Java development environment.

This author cannot endorse any of the popular products that claim to provide a productive Java development environment. In my opinion, the best environment employs Visual SlickEdit ™ by MicroEdge, Inc (http://www.slickedit.com) and the Java Development Kit Version 1.1.8. Visual SlickEdit project files and tag files, created when I developed the example cassette, are included in this cookbook.

If you choose to use some other IDE, be sure it will allow you to use Java Version 1.1.8. While not the latest Java version, it is the version compatible with the IBM WebSphere Payment Manager. If you accidentally use some feature or function that is not in Java Version 1.1.8, it will not be available when the merchant installs your cassette in the field.

The IBM WebSphere Payment Manager Cassette Developer Toolkit

Although this cookbook contains the information needed by most cassette developers, The IBM WebSphere Payment Manager Cassette Developer Toolkit is still a vital resource. It contains more detailed information about the Payment Manager framework, Javadoc for the Payment Manager framework classes, and a sample cassette that is different from the LdbCard sample included in this cookbook. Whenever your cassette has requirements contrary to the simplifying assumptions used to develop the cookbook, the Cassette Developer Toolkit will provide information you need.

1. Start your favorite web browser and navigate to http://www.ibm.com/software/webservers/commerce/payment/download.html .

2. On the resulting panel, find the link to the **Cassette Developer Toolkits** on the right side. Follow the link.

3. On the resulting panel, find the link to the US English version of the toolkit for **PM 2.1**. Follow the link.

4. Unfortunately, the member ID and password you used for the IBM Partnerworld for Developers Program **cannot** be used at this site. If you have not registered previously, use the link to **1. I have not registered before**, follow the directions to register, record your user ID and password, and then return to this step. Otherwise, enter your user ID and password and press **Continue**.

5. On the resulting panel, enter all the required information using **ps1cdt** (all lower case) as the download key, review the license agreement, and press **I accept license** if you agree.

6. On the resulting panel, follow the **cassetteKit2.1.zip**link and save the file on your workstation disk.

7. Use your favorite utility to extract the files in the cassetteKit2.1.zip file. Although you can place these files anywhere, subsequent steps assume that the are extracted to **D:\Kit**.

The PayGen Test Command Generator

The PayGen test command generator is a very powerful test tool. Using your favorite text editor, you can build test scripts that contain any of the WebSphere Payment Manager application program interface (API) commands and verify the data values returned. The cookbook tool kit contains sample test scripts that can easily be modified to test your cassette.

Although the WebSphere Payment Manager user interface can be used to send commands to your cassette during development, there are a number of things it will not do. For example, it will not omit required parameters and verify that the correct return codes are returned. It will not run automatically nor guarantee that a test scenario is repeated exactly the same way every time. PayGen test scripts will.

1. Start your favorite web browser and navigate to http://www.ibm.com/software/webservers/commerce/payment/download.html .

2. On the resulting panel, find the link to the **Cassette Developer Toolkits** on the right side. Follow the link.

3. On the resulting panel, find the link to the US English version of the toolkit for **PM 2.1**. Follow the link.

4. Enter the same user ID and password you used in the last step and press **Continue**.

5. On the resulting panel, enter all the required information using **ps1cdt** (all lower case) as the download key, review the license agreement, and press **I accept license** if you agree.

6. On the resulting panel, follow the **PayGen.exe** link and save the file on your workstation disk.

7. Run PayGen.exe to extract the files. Although you can place these files anywhere, subsequent steps assume that they are extracted to **D:\!PMCassette**.

8. For convenience, you may want to make a hardcopy of the file **D:\PMCassette\PayGen\Readme.txt**. This file contains all the PayGen documentation.

Stage 3: Design Your Cassette

This chapter describes the framework responsibilities, the objects you will be creating, and the tasks included in subsequent chapters of this cookbook.

Understanding the Framework

The IBM WebSphere Payment Manager multi-payment framework provides the environment for your cassette. It is responsible for

- **The Data Model:** The framework defines the payment objects and the administration objects that your cassette must implement. In particular, this cookbook demonstrates how to implement the following objects.

  - A Cassette represents a component that provides cassette dependent process for framework administration and payment requests. It plugs into the framework and services requests sent to the framework and routed to the cassette by the framework.

  - A Cassette Query represents a component that provides an external view of the cassette dependent information. It plugs into the framework's query facility and services query requests sent to the framework. Using criteria supplied on the request, it extracts cassette dependent information and combines it with framework information in a reply document built with eXtended Markup Language (XML) tags.

  - An Order represents a shopper's transaction with the merchant system. It contains information that defines the order (the description and the amount, for example) as well as the information provided by the shopper to pay for the order (the credit card brand and account number, for example).

  - A Payment represents a payment processor interaction that collects money from the shopper. Since the merchant can split an order into separate shipments, there may be more than one payment for each order.

  - A Credit represents a payment processor interaction that returns money to the shopper. Since the shopper may return several items at different times, there may be more than one credit for each order.

  - A Batch represents a collection of payments and credits deposited by the merchant system. When the batch is settled, an interaction with a payment processor will trigger the transfer of funds into the merchant's account.

  - An Account represents the merchant's account at a payment processor.

- **The Application Program Interface for Payments:** The framework defines the application program interface (API) between the merchant system and your cassette. In particular, this cookbook demonstrates how to implement the following payment commands.

  - ReceivePayment is used to create a new order when your cassette must receive its payment information from an electronic wallet.

  - AcceptPayment is used to create a new order when the shopper has already provided the payment information required by your cassette.

- CloseOrder is used to mark the order as complete and to prune the order, including all its payments and credits, from the database.

- CancelOrder is used to remove an order that was created incorrectly so that the order number can be reused by the merchant system (presumably when the shopper reissues the order from the same virtual shopping cart).

- Approve is used to create a new payment.

- ApproveReversal is used to modify an Approval amount when an amount is entered incorrectly or when the merchant system modifies the contents of a shipment (for example, when inventory requires that a single shipment be split into multiple shipments).

- Deposit is used to capture a payment into the batch.

- DepositReversal is used to void a Deposit.

- Credit is used to create a new credit.

- CreditReversal is used to void a Credit.

- BatchOpen is used to create a new batch.

- BatchClose is used to settle a batch and trigger the transfer of funds.

- BatchPurge is used to reverse all the payments and credits currently in an open batch, presumably to recover from conditions that prevent a batch from settling.

- DeleteBatch is used to prune the batch from the database.

- **The Application Program Interface for Administration:** The framework defines that application program interface (API) for the administration of the Payment Manager. Although there are many administration commands, it is typically not necessary for the cassette to participate in all of them. This cookbook demonstrates how to implement the following administration commands.

  - CreateAccount is used to create an account object that contains the cassette dependent information necessary to process payment commands for a given merchant.

  - ModifyAccount is used to modify the merchant's information in an account.

  - DeleteAccount is used to delete an account.

- **The Application Program Interface for Query Operations:** The framework defines the application program interface (API) for query operations and the XML documents returned. This cookbook demonstrates how to implement the classes necessary to return cassette dependent data for a query operation.

- **Access Control:** The framework automatically limits access to the functions provided by a cassette. It ensures that only users authorized by a merchant can view or manipulate the merchant's data. Furthermore, the merchant can restrict each of these users to a particular role:

- A clerk who can create orders, payments, and batches, deposit payments, and query merchant data
- A supervisor who can do everything a clerk can do plus refund money and void payments and refunds.
- A merchant administrator who can do everything a supervisor can do plus administrate the merchant's configuration.

- **Memory management:** The framework manages a memory cache for the order, payment, credit, and batch objects.

- **Synchronization:** The framework automatically ensures data integrity and thread safety for most of the objects used by the cassette while servicing a request. Although the cassette must still provide its own protection for an object (typically contained in the cassette's account object) that can be accessed by multiple threads operating on otherwise independent requests , the framework uses read and write locks to prevent multiple threads from modifying or accessing the same payment objects.

- **Parameter Validation and Conversion:** Using a parameter validation table provided by the cassette, the framework will automatically validate cassette dependent parameter values and convert them from the byte array supplied by the data stream into the fundamental Java types expected by the cassette.

- **Thread Management:** The framework manages a pool of threads. For example, when a request is received from the merchant system, a thread is removed from the pool and used to process the request. When the request is completed, the thread is returned to the pool so it can be reused.

- **Database Access:** The framework provides facilities for JDBC connections.

- **Scheduling Background and Timed Tasks:** Although not used by the sample included in this cookbook, the framework provides a timer facility and a pool of threads that a cassette can use to schedule future work.

- **Protocol Message Management:** Although not used by the sample included in this cookbook, the framework provides facilities to send and receive protocol messages using TCP/IP connections.

- **Event Notification:** For business systems that need asynchronous notification when a payment event has occurred, the framework provides a notification service without explicit actions by the cassette. The cassette merely manages the state of each payment object and the framework automatically notifies the systems listening for the change.

- **Error Logging:** Using a cassette dependent properties file that can easily be translated for foreign languages, the framework provides facilities that allow the cassette to generate informational and error messages that are recorded in an error log. By capturing critical data when an error is detected, the cassette and reduce the effort required to find the cause of a defect and provide service to customers that encounter it.

- **Tracing Internal Events:** The framework provides a granular trace facility for recording internal events, including events in a cassette. During development, the trace facility provides the information necessary to prove that the cassette is operating correctly and to aid perform problem determination when it isn't. Note that it is a bad practice to require trace information to perform problem determination for a problem discovered at an installation in production. The error log should be used instead because no customer wants to turn on trace and recreate an error. However, the trace facility can be used as a last resort.

For more information on the facilities provided by the framework, see the Cassette Kit Programmer's Guide in the IBM WebSphere Payment Manager Cassette Developer Toolkit.

Understanding the Assumptions

This cookbook makes some assumptions about the cassette you will be developing. If any of these assumptions are not valid for your cassette, you will need to make appropriate adjustments to the instructions provided here.

An Existing Payment Appliance

This cookbook assumes that you have previously developed a library, an application or an Internet gateway that acts as a payment appliance for some payment processor. This payment appliance implements all the protocols necessary to interact with the payment processor and has an API that can be used by your cassette.

The LdbCard example included in this cookbook use Java Remote Method Invocation (RMI) to communicate with a payment appliance, but your cassette can exploit other methods just as easily. For example, it can exchange data with your payment appliance using

- File I/O
- Pipes
- XML, HTTP, or any other format carried by a TCP/IP socket
- Database tables
- Java Native Interface (JNI) calls to a dynamic link library
- Java Remote Method Invocation (RMI).

If this assumption is not valid for your cassette, you have additional responsibilities to determine the payment protocols required by the payment processor you wish to support and to implement the software necessary to perform those protocols. Start with this cookbook, but implement the necessary protocols in your cassette account class plus any additional objects referenced by the cassette account object.

Idempotent Payment Appliance Transactions

This cookbook assumes that all transactions performed with the payment appliance are idempotent. In other words, if the identical transaction is performed multiple times, the payment appliance will take no action other than return the same response that was returned for the first time.

Consider the case where the cassette sends a message to the payment appliance but no reply is received within a reasonable time. The cassette cannot distinguish between the case where its message was lost on its way to the payment processor and the case where the reply was lost on the return trip.

If the transaction is idempotent, the cassette simply resends the original request and retries if necessary (within some reasonable limit). Even if all recovery attempts fail, processing is simple. The cassette aborts the operation so that it appears that nothing has happened. If the merchant attempts to perform the same operation again, the cassette services the request exactly as if the first operation never occurred.

If the transaction is not idempotent, the cassette needs additional logic to recover. Perhaps it must send a query to find if the original request was completed before attempting to recover. Perhaps it must attempt to void the original request before attempting to recover. Plus, you must consider how to continue recovery if this step fails and, if all recovery attempts fail, how to remember that recovery protocols are required when the merchant attempts to perform the same operation again.

If your payment appliance transactions are not idempotent, start with this cookbook. When you have completed the cookbook exercise, return to your account class and add the logic necessary to recover in those cases where it cannot tell if a request was processed.

Automatic Synchronous Retry by the Payment Appliance

This cookbook assumes that the payment appliance will automatically retry operations that fail due to data communication failures (also known as network failures). Furthermore, it assumes that control will not be returned to the merchant application until the operation succeeds or the retry limit is exhausted.

If your payment appliance does not automatically retry operations that fail due to data communication failures, start with this cookbook. When you have completed the cookbook exercise, return to your account class and add the logic necessary to recover.

If the frequency or duration of your recovery options withhold control from the merchant for an unsatisfactorily long period, start with this cookbook. When you have completed the cookbook exercise, return to your account class and add additional logic. The logic should schedule retry operations using the framework's timer thread or a thread from the service thread pool and then return control to the merchant using a primary return code value of PRC_OPERATION_PENDING. The Cassette Kit Programmer's Guide in the IBM WebSphere Payment Manager Cassette Developer Toolkit contains more information about using the framework's timer thread or a thread from the service thread pool.

No Wallet

This cookbook assumes that your cassette does not use a browser plug-in to obtain payment information from the shopper. In other words, it does not support a wallet and will not support the framework's ReceivePayment command.

If this assumption is not valid for your cassette, start with this cookbook. When you have completed the cookbook exercise, look at the KitCash cassette included in the IBM WebSphere Payment Manager Cassette Developer Toolkit. It demonstrates how to implement the ReceivePayment command to generate a wake up message for a browser plug in, how to implement a browse plug in, and how to use a framework ComPoint to listen and exchange protocol messages with the plug in.

One Account per Merchant

This cookbook assumes that your cassette supports only one account per Payment Manager merchant ID. If more than one account is required at an installation, a different Payment Manager merchant ID must be used for each account.

If this assumption is not valid for your cassette, start by using this cookbook. When you have completed the cookbook exercise, extend your implementation to support multiple accounts for each merchant. Be sure to determine how an account is assigned to a particular order. You will need to answer these design questions.

- Why does the merchant want to use multiple accounts?
- What, in addition to the merchant ID, will be used to select the correct account?
- Will it be the currency?
- Will it be the credit card brand?
- Will it be a special value passed from the merchant using a cassette specific protocol data keyword, like $ACCOUNTNUMBER for example?
- Will you need to implement an algorithm to select a default account when a specific criterion is not provided?

Once you have this design complete, you need to enhance the retrieveAccount() static method in your cassette account class as well as all the places where the method is called.

One Currency per Account

This cookbook assumes that a given account will support one and only one currency. If the merchant requires more than one currency, another account must be defined.

If this assumption is not valid for your cassette, start by using this cookbook. When you have completed this cookbook exercise, extend your implementation to support multiple currencies per account. Most often, this will require that you implement multiple batches for each account, one batch per currency.

A Tangible Batch

This cookbook assumes that payments and credits are accumulated into a tangible batch and that money is not transferred to the merchant's account until the batch is closed. In some systems this is known as batch settlement. Furthermore, it assumes that the settlement is triggered by a protocol that is initiated at the merchant site.

If your cassette uses a batch that can only be closed by the payment network and there is a settlement protocol exchange that informs the merchant when the batch is closed, start by using this cookbook. When you have completed the cookbook exercise, modify the batch implementation so it always rejects a BatchClose API request but still performs the batch close functions when triggered by the settlement protocol exchange.

If your cassette has no batch concept (all money is transferred in real time) or if batches are closed automatically by the payment network without any protocol exchange with the merchant system, it is recommended that you implement a cosmetic batch. A cosmetic batch is a batch that does not participate in settlement protocols, but still allows the merchant to collect payments and credits. The batch allows the merchant to organize payments and credits that are related by time. Also, batch reports can be used for cash flow analysis.

If you want to implement a cosmetic batch, start by using this cookbook. Then, while enhancing the method that performs settlement protocols, modify the method to simply return a boolean value of true.

One Open Batch per Account

This cookbook assumes that a given account will have only add payments and credits to one open batch at a time. Although there may be more than one open batch, all accept the most recently opened batch will not be used to hold new transactions. Once the cassette assigns an order to an account, no further information is required to determine which batch will contain the order's payments and credits.

If this assumption is not valid for your cassette, start by using this cookbook. When you have completed this cookbook exercise, extend your implementation to support multiple batches per account. Be sure to determine how a batch is chosen, given a merchant ID and an account. Then modify the selectBatch() static method in your cassette account class as well as all the places where the method is called.

Required Authorization Protocols

This cookbook assumes that authorization protocols must be exchanged when a payment is created. If the payment is authorized, another protocol exchange is used to place the payment into a batch. For most of the merchant systems in the world, authorization is very important. The authorization is performed while the shopper is still available. If the shopper specified a payment parameter value incorrectly, the shopper

can be informed immediately and allowed to correct the mistake before the order is accepted. Later, when the goods are ready to ship, a subsequent protocol flow is used to capture the payment so the funds can be transferred.

If there is some way to validate a payment's parameter values before committing to the exchange of funds, it is recommend that you perform this validation as part of the payment authorization protocols. Remember that the shopper is on-line during the authorization phase but is typically unavailable during the capture phase. If something is wrong with the payment information provided, it is much easier to notify the shopper during the authorization phase.

The authorization protocols for your cassette may not be required or even obvious. For example, debit cards typically don't require authorization before the payment is captured. However, there are debit card protocols that allow you to simulate an authorization protocol by specifying an amount of zero or by setting a special flag.

If there is no way to implement authorization protocols for your cassette, start by using this cookbook. Then, while enhancing the method that implement the authorization protocols, modify the method to simply return a boolean value of true.

Single Tier Configuration

This cookbook assumes that all the configuration information required by the cassette is held by the cassette's account object (more formerly known as the cassette extension to the framework's AccountAdmin object).

Remember, there is a cassette account for each merchant and, by definition, it contains all the information necessary to identify the merchant and connect to the payment appliance. For LdbCard, there is a fixed amount of configuration information for each merchant. Furthermore, if there is configuration information that may be common to all merchants (for example, the RMI server URL may be a candidate for a common parameter), the information is simply duplicated in every account.

This assumption is highly recommended. The simplicity and convenience of having a single configuration point typically outweighs the limitation caused by fixing the amount of configuration information and the burden of re-entering duplicate information. However, there clearly are cases where this assumption will not be valid.

Here are design questions to help you determine if this assumption is valid for your cassette.

1. Does your cassette have a fixed number of configuration parameters that are common for all merchants and their accounts? If so, consider creating a cassette extension to the framework's CassetteAdmin object.

2. Does your cassette have a group of related parameters, common for all merchants and their accounts, where there may variable number of groups in the configuration,? If so, consider creating a system cassette object where each object represents one group of parameters in the collection. This system cassette object would configure the framework's CassetteAdmin object.

3. Does your cassette have a fixed number of configuration parameters that are different for all merchants but common over all the accounts for a given merchant? If so, consider creating a cassette extension to the framework's PaySystemAdmin object.

4. Does your cassette have a group of related parameters, typically different for each merchant but common over all the accounts for a given merchant, where there may be a variable number of groups in the configuration? If so, consider creating a merchant cassette object that configures the

frameworks PaySystemAdmin object.

5. Does your cassette have a fixed number of configuration parameters for each merchant account? If so, consider creating a cassette extension to the framework's AccountAdmin object.

6. Does your cassette have a group of related parameters where there may be a variable number of groups for each account in the configuration? If so, consider creating a merchant cassette object that configures the framework's AccountAdmin object.

While designing LdbCard, only question 5 was answered with an overwhelming "YES!". Even for a parameter that could have a more global scope (the RMI server URL, for example), careful consideration made it apparent that the parameter was better suited to the cassette extension to the framework's AccountAdmin object. Using the RMI server URL as an example, placing the parameter on the account allows the cassette to use a simple method for configuration: all parameters are provided with the account is created and there is no need to manage relationships with other configuration objects. In addition, it supports the flexibility of using multiple RMI servers in order to distribute the load or the administration domain.

However, if you determine that this assumption is not valid for your cassette, start by using this cookbook. During this stage of development, simply define constants for the fixed parameters and collections of constants for the parameter groups where the configuration may contain a variable number of groups. When you have completed the cookbook exercise, enhance your cassette to support the additional configuration objects. You can use your cassette's account object as a template. Be sure to

1. Define or select a protocol data keyword for each cassette dependent configuration parameter

2. Enhance your cassette's clsValidationTable to include a validation item for each additional protocol data keyword

3. Enhance your cassette's readCassetteConfig() method to read the configuration tables and construct the additional configuration objects during initialization

4. Enhance your cassette's verifyCassetteConfig() method to assure the relationships between the various configuration objects are correct

5. Enhance your cassette's initializeCassette() method to perform any initialization that the new configuration objects may require

6. Enhance your cassette's validateAdminRequest() and processAdminRequest() methods to process the administration requests that create, modify, and delete the additional configuration objects

7. Enhance your cassette's PSPL file so the Payment Manager user interface can provide the proper panels to display, create, modify, and delete the additional configuration objects

8. Enhance your cassette's query object so it can respond to query operations that include information from the additional configuration objects.

The Cassette Kit Programmer's Guide in the IBM WebSphere Payment Manager Cassette Developer Toolkit contains more information about the framework object model and how to implement cassette dependent administration objects.

Implicitly Open Batch with Automatic Close and Manual Close

This cookbook assumes that no merchant interaction is required to open a batch. A batch is automatically opened as needed.

If this assumption is not valid for your cassette, start by using this cookbook. When you have completed this cookbook exercise, extend your implementation to support BatchOpen requests and, if necessary, adjust the logic that opens a batch automatically.

This cookbook assumes that the merchant can optionally configure an account so a batch will be closed automatically. There are two schools of thought on this feature and this cookbook demonstrates both. Be aware that while a batch is closing, no other command that uses the same account can be processed. Any command that references the account will be suspended until the batch processing completes.

1. To limit the amount of time commands are suspended, the merchant may want to configure a maximum batch size. That way, the batch never reaches a size where the delay will become intolerable. When the batch reaches the configured size, the batch will be closed automatically.

2. Often, it is convenient if the batch is closed at the same time every day, typically early in the morning when the system is mostly idle. This not only reduces the chance of suspending a command for an intolerable period, it also allows the merchant to use the batch totals as a daily cash flow value. For this feature, the merchant may want to specify two values: the time of day (specified as minutes after midnight) when the batch is to be automatically closed and the minimum batch total. The merchant typically pays a fee every time a batch is closed. The minimum batch total prevents the batch from being closed when it is not worth the fee.

If this assumption is not valid for your cassette, start by using this cookbook and then disable the code that allows the merchant to configure these values for your account. The code that implements this feature will never perform an automatic close and will still be available if you need to add this feature in the future.

In addition, this cookbook assumes that the merchant can issue a request that triggers batch settlement. When the merchant issues a BatchClose request, the batch is settled.

If this assumption is not valid for your cassette, start by using this cookbook. When you have completed this cookbook exercise, extend your implementation to support batch settlement correctly. Be sure to add the logic necessary to change the state of all payments and credits contained in the batch. When a batch is closed, the state of all its payments and credits should be set to CLOSED as well.

Mapping the Payment Manager API to Your Payment Appliance API

You must determine how to map the merchant Payment Manager requests into actions performed by your payment appliance. Start by reviewing the LdbCard mapping.

The payment appliance used by the LdbCard cassette included in this cookbook, does not support approve reversals. In order to change a payment amount, the LdbCard cassette must tell the payment appliance to void the original payment and then authorize a new payment with the correct amount.

Furthermore, the payment appliance used by the LdbCard cassette does not have a perfect map for deposit reversals. When the payment appliance voids a payment, the payment authorization becomes void as well. But, semantically speaking, the Payment Manager DepositReversal request must set the captured amount to zero while maintaining the original approval. Therefore, the LdbCard cassette must tell the payment appliance to void the original payment and then authorize a new payment for the original authorized amount.

Here is the mapping used by the LdbCard cassette.

Payment Manager Request	Payment Appliance Equivalent
ReceivePaymengt	No equivalent. The payment appliance only tracks payments and credits. It does not understand the relationship between payments/credits and the shopper's order.
AcceptPayment	No equivalent. Same reason as above.
CloseOrder	No equivalent. Same reason as above.
CancelOrder	No equivalent. Same reason as above.
Approve	Authorize (Book).
ApproveReversal	Void followed by Authorize (Book) for the new amount if the new amount is not zero.
Deposit	Capture (Ship).
DepositReversal	Void followed by Authorize (Book) for the amount of the original approval.
Refund	Credit.
RefundReversal	Void.
BatchOpen	No equivalent. The payment appliance automatically opens a batch when needed.
BatchClose	Settle.
BatchPurge	Void followed by Authorize (Book) for the original approval amount for every payment in the batch and void for every credit in the batch.
BatchDelete	No equivalent. The payment appliance automatically prunes a batch, including all its payments and credits, when a closed batch reaches a certain age (configured in the payment appliance).¾
CreateAccount	No equivalent. The payment appliance does not support dynamic configuration. It assumes that its own user interface is used to configure a merchant before the cassette starts to use the merchants account.
ModifyAccount	No equivalent. Same reason as above.
DeleteAccount	No equivalent. Same reason as above.

The Cassette Kit Programmer's Guide in the IBM WebSphere Payment Manager Cassette Developer Toolkit contains more information about mapping the Payment Manager API to your payment protocols. Furthermore, the Programmer's Guide and Reference included in the IBM WebSphere Payment Manager contains descriptions for all the commands in the Payment Manager API. After reviewing this reference material, draw a table with the same dimensions as the table above with the same values in the first column. In each cell of the second column, describe how your payment protocol would be used to implement the command listed in the corresponding cell to its left.

If there is a command that is not supported by your payment protocol, it is important to implement your cassette to process the command and advance the state of the framework and cassette objects as if the command was processed successfully. This technique, sometimes call a cosmetic implementation, keeps the merchant system independent of the cassette being used for a particular order. In other words, the merchant can remove a cassette and replace it with your cassette without making major modifications to the merchant's automated business system.

For example, consider a payment protocol that does not support explicit batch settlement. Perhaps the payment processor automatically settles the batch at a given time of day and there is no payment protocol that allows the cassette to synchronize its batch with the payment processor batch.

In this case, it is still important for the cassette to process a BatchClose command and close all the payment and credits in the batch being closed. Even though there is no action perform by the payment processor, an automated merchant system can still close the batch. The payments and credits in the closed batch will no longer be candidates for reversal commands and the merchant can generate the batch reports used to determine how much business was done over the scope of that batch.

Designing Your Persistent Data

Although the framework maintains most of the data required for processing credits and payments, there is always additional cassette dependent data required as well. When this data must be persistent (in other words, the information must be restored to memory when the Payment Manager restarts or when a cassette object is restored after being pushed out of the memory cache), it must be maintained in cassette dependent database tables.

To design your cassette dependent database tables, examine the interface provided by your payment appliance. Your cassette must be able to manufacture a value for each value required in the interface. Typically, all parameter values fall into one of the following categories:

- Account data
- Order data
- Payment data
- Credit data
- Batch data.

Start by organizing your cassette parameters values into two categories: values supplied and values returned. Then use the following design questions to decide where to place the parameter value.

Design Questions for Supplied Values

- Can the parameter value be computed from other parameters? If so, there is no need to make the value persistent. However, if you want to make the value persistent for the sake of convenience, use the design questions for returned values to determine where to place the parameter value.

- Does the parameter value affect the behavior of your cassette logic? Retry limits, timeout values, maximum batch sizes are examples of this type of parameter. If so, the value should be placed in the account's persistent data.

- Is the parameter value the same for all payment appliance commands? User IDs, passwords, payment appliance URLs, and merchant identifiers are examples of this type of parameter. If so, the value should be placed in the account's persistent data.

- Does the parameter value change for each shopper? Personal account numbers (PAN), account expiration dates, account brands, and billing addresses are all examples of this type of parameter. If so, the value should be placed in the order's persistent data.

- Does the parameter value change for each order? Order descriptions and line item details are examples of this type of parameter. If so, the value should be placed in the order's persistent data.

- Is the parameter value returned on some payment appliance command and then supplied on subsequent commands? Transaction identifiers and approval codes are examples of this type of parameter. If so, use the design questions for returned values to determine where to place the parameter value.

- If you have reached this point in your analysis, you are probably tempted to put the supplied value in the payment, credit, or batch persistent data. Resist. Unless the values are computed from other values or returned by payment appliance commands, placing supplied values in the payment, credit, or batch persistent data is **not recommended**.

It is relatively easy for a merchant system to provide cassette dependent parameter values when an order is created. Using parametric designs and profiling, merchant systems can create new orders and pass a collection of protocol data values that are valid no matter which cassette is being used. Passing cassette dependent information when a payment is created (the Approve command), when a credit is created (the Refund command), or when a batch is created (the BatchOpen command) is much more difficult. Most automated business systems do not have a way to add cassette dependent parameter when these commands are used. Humans using the user interface would be aggravated by the extra effort required to enter cassette dependent information when performing these operations.

However, if the parameter is optional, a reasonable default value is acceptable in most cases, and a different value is only needed in extraordinary circumstances, supplied values can be kept in the payment, credit, or batch persistent data.

- Is the parameter value supplied when a payment is created? An authorization code obtained from the payment processor's operator when a "voice auth" has been used is an example of this type of parameter. If so, the value should be placed in the payment persistent data.

- Is the parameter value supplied when a credit is created? An authorization code obtained from the payment processor's operator when a "voice auth" has been used is an example of this type of parameter. If so, the value should be placed in the credit persistent data.

- Is the parameter value supplied when a batch is created? A new batch identifier assigned by the payment processor's operator when manually reconciling a batch that doesn't balance is an example of this type of parameter. If so, the value should be placed in the batch persistent data.

Design Questions for Returned Values

- Does the parameter value indicate the outcome of creating a new account? Reason codes, return codes and error messages are examples of this type of parameter. If so, the value should be placed in the account's persistent data.

- Does the parameter value provide an identifier that is supplied when subsequent orders, payments or credits are created? Account identifiers, merchant identifiers and secure socket layer (SSL) certificates are examples of this type of parameter. If so, the value should be placed in the account's persistent data.

- Does the parameter value indicate the outcome of creating a new order? Fraud scores, export indicators, and return codes are examples of this type of parameter. If so, the value should be placed in the order's persistent data.

- Does the parameter value provide an identifier that is supplied when subsequent payments or credits are created? Order identifiers and reference numbers are examples of this type of parameter. If so, the value should be placed in the order's persistent data.

- Does the parameter value indicate the outcome of an operation on a payment? Reason codes, return codes, and error messages are examples of this type of parameter. If so, the value should be placed in the payment's persistent data.

- Does the parameter value provide an identifier that is supplied when subsequent operations are performed on the payment? Transaction identifiers and approval codes are examples of this type of

parameter. If so, the value should be placed in the payment's persistent data.

- Does the parameter value indicate the outcome of an operation on a credit? Reason codes, return codes, and error messages are examples of this type of parameter. If so, the value should be placed in the credit's persistent data.

- Does the parameter value provide an identifier that is supplied when subsequent operations are performed on the credit? Transaction identifiers and approval codes are examples of this type of parameter. If so, the value should be placed in the credit's persistent data.

- Does the parameter value indicate the outcome of an operation on a batch? Reason codes, return codes, and error messages are examples of this type of parameter. If so, the value should be placed in the batch's persistent data.

- Does the parameter value provide an identifier that is supplied when subsequent operations are performed on the batch? Transaction identifiers and approval codes are examples of this type of parameter. If so, the value should be placed in the batch's persistent data.

Account Data

Typically, account data contains the information required to connect to the payment appliance and identify the merchant within the payment appliance's configuration. It is usually configuration data that is only read by the cassette. The cassette writes account data when an account is created or modified and reads account data during initialization.

The LdbCard cassette keeps account data in the LdbCardAccount database table. Each row in the table represents a unique Merchant/Account combination.

Column Name	Value Description
MerchantNumber	The identifier used by the Payment Manager to identify the merchant.
AccountNumber	The identifier used by the Payment Manager to identify the account
Currency	The ISO currency code for the currency supported by the payment appliance for this account.
UserId	The identifier used to get access to the RMI server that interfaces with the payment appliance.
Password	The password used by the RMI server to authenticate the user Id.
LdbCardId	The identifier used by the payment appliance to identify this merchant.
LdbCardRmiUrl	The universal resource locator used to make a connection to the RMI server.
BatchCloseTime	The time of day, specified as minutes after midnight, when the cassette will automatically close the currently open batch, if any.
MinBatchTotal	The smallest amount of money that the batch must have in order to be a candidate for an automatic close at the specified time of day. If the value has not been reached, the automatic close will be rescheduled for the next day.
MaxBatchSize	The maximum number of transactions allowed in a batch. When this number is reached, a new batch is automatically opened and the current batch is closed.

Most likely, the first three columns will be the same for your cassette. The MerchantNumber and AccountNumber make up the primary key. The currency is used to double-check the validity of the currency specified for a new order. The remaining columns need to be replaced to hold the information that your cassette needs to connect to the payment appliance and identify the merchant.

Note that the merchant must provide cassette dependent account information as protocol data passed on the CreateAccount verb of the Payment Manager framework API. Typically, the Payment Manager administrator configures a merchant and enables the merchant to use your cassette. Then, using a user ID authorized as a merchant administrator, the merchant logs into the Payment Manager user interface and creates the account.

The user interface presents a panel where the merchant enters the cassette dependent information and the issues the CreateAccount verb with the correct protocol data keywords and values. Your cassette provides an XML document containing PSPL tags to tell the user interface how to display the configuration panel, how to pass the parameter keywords and values on the CreateAccount verb, and how to transform the command's return codes into a message for the operator.

Order Data

Typically, order data contains the information required to identify the shopper's account and to authenticate the shopper. The cassette writes order data when a new order is created and reads order data to resurrect an order into the cache.

The LdbCard cassette keeps order data in the LdbCardOrder database table. Each row in the table represents an order that has been accepted by the cassette.

Column Name	Value Description
MerchantNumber	The identifier used by the Payment Manager to identify the merchant.
OrderNumber	The identifier used by the Payment Manager to identify the order.
Amount	A running total of all the approved payments.
AccountNumber	The identifier used by the Payment Manager to identify the cassette account used to process payments and credits for this order.
Pan	The shopper's personal account number (PAN).
Expiry	The expiration date of the shopper's account in YYYYMM format.
Brand	The brand name of the shoppers credit card.
Address	The cardholder's street address (for the address verification system (AVS))
PostalCode	The cardholder's postal code (also for AVS)

Most likely, the first four columns will be the same for your cassette. The MerchantNumber and the OrderNumber make up the primary key. The Amount is used to verify that the merchant does not attempt to use multiple payments to obtain more money than the shopper authorized when the order was created. The account number is used to identify the account object that will be used to process the payments and credits for this order. The remaining columns need to be replaced to hold the information that your cassette needs to identify the payment instrument and identify/authenticate the shopper.

Note that the merchant must provide cassette dependent order information as protocol data passed on the AcceptPayment command. Typically, the merchant presents the shopper with a secure HTML form where this information is entered manually. Although not covered by this example, an alternative is for your cassette to wakeup a wallet plug-in inside the shopper's browser and use it to obtain the required information.

The Cassette Kit Programmer's Guide in the IBM WebSphere Payment Manager Cassette Developer Toolkit contains more information about implementing a cassette that uses a wallet.

Payment Data

Typically, payment data contains the information required to identify the payment in the payment appliance as well as information about the result of a payment operation. The cassette writes payment data when a

new payment is created or when a payment changes state. The cassette reads payment data to resurrect a payment into the cache.

The LdbCard cassette keeps payment data in the LdbCardPayment database table. Each row in the table represents a payment being managed by the cassette.

Column Name	Value Description
MerchantNumber	The identifier used by the Payment Manager to identify the merchant.
OrderNumber	The identifier used by the Payment Manager to identify the order.
PaymentNumber	The identifier used by the Payment Manager to identify the payment.
ApprovalCode	The identifier returned by the payment appliance and used to identify the payment.
DeclineReason	An extra bit of information used to explain why the payment appliance declined a request (not strictly required by the Payment Manager but handy for problem determination)

Most likely, the first three columns will be the same for your cassette. The MerchantNumber, OrderNumber, and PaymentNumber make up the primary key. The remaining columns need to be replaced to hold the information required to identify the payment in your payment appliance and assist in problem determination when there is a data communication or financial failure.

Credit Data

Typically, credit data contains the information required to identify the credit in the payment appliance as well as information about the result of a credit operation. The cassette writes credit data when a new credit is created or when a credit changes state. The cassette reads credit data to resurrect a credit into the cache.

The LdbCard cassette keeps credit data in the LdbCardCredit database table. Each row in the table represents a credit being managed by the cassette.

Column Name	Value Description
MerchantNumber	The identifier used by the Payment Manager to identify the merchant.
OrderNumber	The identifier used by the Payment Manager to identify the order.
CreditNumber	The identifier used by the Payment Manager to identify the credit.
ApprovalCode	The identifier returned by the payment appliance and used to identify the credit.
DeclineReason	An extra bit of information used to explain why the payment appliance declined a request (not strictly required by the Payment Manager but handy for problem determination)

Most likely, the first three columns will be the same for your cassette. The MerchantNumber, OrderNumber, and CreditNumber make up the primary key. The remaining columns need to be replaced to hold the information required to identify the credit in your payment appliance and assist in problem determination when there is a data communication or financial failure.

Batch Data

Typically, batch data contains the information required to identify the batch in the payment appliance as well as information about the result of a batch operation. The cassette writes batch data when a new batch is opened or when a batch changes state. The cassette reads batch data to resurrect a batch into the cache.

The LdbCard cassette keeps batch data in the LdbCardBatch database table. Each row in the table represents a batch being managed by the cassette.

Column Name	Value Description
MerchantNumber	The identifier used by the Payment Manager to identify the merchant.
BatchNumber	The identifier used by the Payment Manager to identify the batch.
ApprovalCode	The identifier returned by the payment appliance and used to identify the batch.
DeclineReason	An extra bit of information used to explain why the payment appliance declined a request (not strictly required by the Payment Manager but handy for problem determination)
ClosedToNewTrans	An indicator used to determine whether or not this batch can be used for new transactions. This value starts as false when the batch is created and is set to true when the batch is either automatically or manually closed, even if the close attempt fails.

Most likely, the first two columns will be the same for your cassette. The MerchantNumber and BatchNumber make up the primary key. The remaining columns need to be replaced to hold the information required to identify the batch in your payment appliance and assist in problem determination when there is a data communication or financial failure.

Designing Your Protocol Data

Cassette dependent parameters are passed to your cassette as protocol data. You must design a protocol data keyword and value for each cassette dependent parameter required. Follow these guidelines.

- By definition, a protocol data keyword must begin with a '$' character. This prefix is used by the framework to decide how to parse the value, validate the value, and pass it to the cassette.

- By convention, protocol data keywords are defined with every alphabetic character in upper case.

- Whenever possible, use a protocol data keyword that has already been defined by the Payment Manager framework. These keywords have been standardized to improve the flexibility of the Payment Manager and to keep the merchant software independent of the cassette being used. If all cassette developers use the standard protocol data keywords for to pass common values, the merchant can mix and match cassettes without modifying the business system. To find the protocol data keywords that have been standardized, look in the PaymentAPIConstants Javadoc for constants with a **PD_** prefix.

- If you don't find a standard protocol data keyword for a cassette dependent parameter, propose a keyword to be added to the standardized list. Merchants need the ability to smoothly migrate to new technologies and remain independent of the cassette being used.

- For each cassette dependent parameter, choose a protocol data keyword. Document the keyword and data type of the value. Valid data types include
  - Boolean
  - Byte Array
  - Integer
  - Long Integer
  - Numeric String
  - Numeric Token (a numeric string suitable for identifiers like merchant number, order number, and so on)
  - Path (a existing file or directory path on the local workstation)
  - Restricted String (a string where some characters are not allowed)

  - String
  - Timestamp

- For each protocol data keyword documented above, document the secondary return code value that will be used to identify the parameter when the value is not valid. Note that there are already standard secondary return code values defined for every standard protocol data keyword defined in the framework. For non-standard protocol data keywords, you will need to define cassette dependent secondary return code constants. By convention, these return code values are contiguous values that begin with the value 100001.

Understanding the Components You Will Develop

The Cassette Kit Programmer's Guide in the IBM WebSphere Payment Manager Cassette Developer Toolkit contains important for understanding the Payment Manager Framework. Appendix E: in this cookbook contains a formal object model definition of the classes developed for the LdbCard cassette. The following sections describe the component you will develop for your cassette.

Constants

You will implement a Java Interface that will contain constants used by all your other classes. Typically, the interface defines constant values for

- Cassette dependent database table names
- Column names for the cassette dependent tables
- XML labels for cassette dependent data returned on a query command
- Cassette dependent database views for query commands
- Column names for the cassette dependent database views
- Cassette dependent protocol data keywords
- Cassette dependent secondary return codes
- Cassette dependent message numbers.

Before placing any other constant in this interface, be sure the constants have a global scope. A constant is considered global if it used in more than one class (like database table names) or if it used to communicate with other systems (like return codes and message numbers). If the constant is use strictly within one class, it is usually better to define the constant in the class where it is used.

Cassette

You will implement a Cassette class, the first prong of the framework plug-in. The Cassette class is the primary interface between the framework and your cassette. The framework dynamically loads your Cassette class and constructs your Cassette object. As usual for classes loaded dynamically, the no-argument constructor is used when instantiating the object. However, there are a number of other methods that your cassette must implement to initialize your cassette.

The Cassette object is a singleton. The framework will only construct one Cassette object in the Java virtual machine that contains the payment engine.

The Cassette object is a factory. The framework uses it to construct all the other cassette dependent payment objects and administration objects. In addition, the framework uses it to construct new cassette objects and resurrect cassette objects that have been pushed out of the memory cache.

The Cassette object is a proxy. The framework sends all payment and administration commands to the service() method of the cassette. In turn, the cassette sends each request to the appropriate cassette object for processing, obtains a response from that object, and returns the response to the framework.

CassetteQuery

You will implement a CassetteQuery class. This is the second prong of the framework plug-in. The CassetteQuery class provides the interface for query commands that may return cassette dependent information. The framework dynamically loads your CassetteQuery class and constructs your CassetteQuery object. As usual for classes loaded dynamically, the no-argument constructor is used.

The CassetteQuery object is a singleton. The framework will construct one CassetteQuery object in the servlet's Java virtual machine. Note that the CassetteQuery object is in a different virtual machine than the Cassette object. Because of this, the CassetteQuery object does not have access to any of the cassette's payment objects. The cassette's database table provide the information returned on a query command.

The CassetteQuery object is a proxy. The framework sends all query commands to the query() method of the CassetteQuery object.

Account

You will implement an Account class. The Account object is responsible for managing the relationship between the merchant and the payment appliance. It uses the cassette's account configuration table to create account objects and verifies that the configuration is syntactically and semantically correct. It is responsible for making connections to the payment appliance and exchanging data between the cassette and the payment appliance.

It is also responsible for performing all the cassette account administration commands (CreateAccount, ModifyAccount, DeleteAccount) and for archiving cassette dependent account data in persistent storage (in other words, a cassette dependent account table in the database).

Order

You will implement an Order class. The cassette order object is responsible for validating all relevant cassette dependent data (also known as protocol data) passed when the order is created, managing the state of the framework order object, and routing all the order requests. In addition, it is responsible for archiving all cassette dependent order data in persistent storage (in other words, a cassette dependent order table in the database).

Payment

You will implement a Payment Class. The cassette payment object is responsible for performing all cassette's payment requests (Approve, ApproveReversal, Deposit, DepositReversal) and to manage the state of the framework payment object. In addition, it is responsible for archiving all cassette dependent payment data in persistent storage (in other words, a cassette dependent payment table in the database).

Credit

You will implement a credit class. The cassette credit object is responsible for performing all cassette's credit requests (Refund, RefundReversal) and to manage the state of the framework credit object. In addition, it is responsible for archiving all cassette dependent credit data in persistent storage (in other words, a cassette dependent credit table in the database).

Batch

You will implement a batch class. The cassette batch object is responsible for performing all cassette's batch requests (BatchOpen, BatchClose, BatchPurge, DeleteBatch) and to manage the state of the framework batch object. In addition, it is repsonsible for archiving all cassette dependent batch data in persistent storage (in other words, a cassette dependent batch table in the database).

Properties

You will implement a properties file that will be loaded into a Java ResourceBundle. This properties file must contain all the messages that will be logged by your cassette. To support additional national languages, your properties file must be translated.

As you would expect, the format of your properties file is consistent with the Java Properties class. The message number is the key and the message text is the value. Language independent substitution is performed using rules consistent with the Java MessageFormat class.

If you have other language dependent or installation dependent constants, you can place them in this properties file as well.

An XML Document

You will implement an XML document. The framework requires an XML document in order to display and modify cassette dependent information on the user interface panels. This document contains Payment Server Presentation Language (PSPL) tags that are interpreted by the Payment Manager user interface classes.

Using PSPL tags, you will provide cassette dependent extensions to

- The framework's PSMerchantAccount panel
- The framework's PSBatch panel
- The framework's PSOrder panel
- The framework's PSPayment panel
- The framework's PSCredit panel

For each cassette dependent parameter that can be displayed or modified, you will provide a PSPL field tag in the appropriate extension.

Understanding the Incremental Tasks

This cookbook employs a technique commonly called Incremental Refinement. You will start with a cassette skeleton and incrementally add and replace methods for each scenario until you have a fully functional cassette.

- After each task is completed, you should be able to compile your cassette without any errors. The code should not depend on the completion of subsequent tasks
  .
- When all the tasks in a scenario are completed, you should be able to test the scenario successfully. The scenario should not depend on the completion of subsequent scenarios.

- As you progress through the cookbook exercise, you may discover public constants and public methods (usually setters and getters) used by LdbCard that are inappropriate for your cassette. When this happens, you may not be able to delete the code because the constant or method may still be used by code that will not be modified until a subsequent step. Instead of deleting the code, copy the code to a gutter area in the module. At the end of the exercise, there is a step where you can safely remove all the code in the gutter areas. Note that private constants and private methods **can** be deleted in the task where they are discovered.

- For your convenience, each interface and class (including the inner classes in the CassetteQuery module) contains a gutter area that can be identified by the following comments.

```
//<gutter>
//</gutter>
```

- In addition to moving the inappropriate public constant or public method to the gutter area, declare it deprecated. At the end of the cookbook exercise, you can recompile your cassette using the –deprecated switch before deleting the gutters. If warnings occur, you must either fix the code that is using the deprecated item or decide to use the code and move it out of the gutter before the gutters are deleted.

- As you progress through the cookbook exercise, you will be asked to modify methods originally taken from the LdbCard example. The extent of the modification will depend on the similarity between your cassette and the LdbCard cassette: the closer the similarity, the fewer the changes.

- Almost all your modifications will be in the methods of the Account class that interact with your payment appliance and the methods in the other classes that all them. Most other modifications will be related to your cassette's protocol data and persistent data.

Understanding the Author's Coding Style

Like all programmers, experience has taught me many tricks that improve my productivity as well as the code's readability and maintainability. Unfortunately, these tricks have advanced my coding style so far beyond conventional wisdom that it may appear foreign to you.

Variable Naming Conventions

There are many conventions that add information to the name of a variable in order to aid the programmer. For example, the Hungarian notation was used so a programmer would know the variable's type just by looking at the variable name. My notation, the Bader notation, adds a different level of information to the variable name.

Like you, I have found that the most valuable attribute of a variable name is an effective description of its contents but, in my experience, I have found that the second most valuable attribute is information about where the variable is declared. The Bader notation gives immediate information about the source of the variable's value and the implications of modifying the value. (Does the variable have local scope? Is it an argument? Is it a constant?

Here are the conventions that I used in the LdbCard cassette along with some examples taken from the LdbCardAccount class. I encourage you to maintain this convention in your cassette.

- **Constant variables have names where every word is completely capitalized and words are separated with underscores.**

```
private static final String ICV_KEY_ACCOUNT  = "Account";
```

- **Class variables (also known as static data) have a lower case "cls" prefix and the first letter of each subsequent word is capitalized.**

```
private static Hashtable    clsAccountList    = new Hashtable();
```

- **Object variables (also known as instance data or state data) have a lower case "obj" prefix and the first letter of each subsequent word is capitalized.**

```
private String       objMerchant;
```

- **Method arguments have a lower case "arg" prefix and the first letter of each subsequent word is capitalized.**

```
protected LdbCardAccount(
   ResultSet        argResultSet
) throws SQLException {
```

- **Local variables (sometimes called automatic variables, working variables, or temporary variables) have a lower case "var" prefix and the first letter of each subsequent word is capitalized.**

```
String varSelect =
         " SELECT * FROM " +
         ETillArchive.getOwner() + "." + TBL_ACCOUNTS;
```

- **Class names can be distinguished from variable names because they have no lower case prefix and the first letter of each word is capitalized.**

- **Primitive types can be distinguished from variable names and class names because they are completely lower case.**

Class and File Naming Conventions

The classes implemented while you complete this exercise should follow this convention. If you are implementing a cassette for a protocol called FastPay

- FastPayConstants should be the name of your constants interface
- FastPayCassette should be the name of your cassette class
- FastPayCassetteQuery should be the name of your cassette query class
- FastPayAccount should be the name of your account class
- FastPayOrder should be the name of your order class
- FastPayPayment should be the name of your payment class
- FastPayCredit should be the name of your credit class
- FastPayBatch should be the name of your batch class

By a convention required by Java, each class must be implemented in a file with a file name that is the same as the class or interface name and a file type of ".java".

The name of your default properties file must be FastPay.properties.

The name of your XML document, assuming it is for the English Language must be FastPay.en.PSPL.

Database Table and Column Naming Conventions

To prevent naming collisions in the merchant database, all your database table names should begin with your cassette name. If you are implementing a cassette for a protocol called FastPay

- FastPayAccount should be the name of your cassette dependent account configuration table
- FastPayOrder should be the name of your cassette dependent order table
- FastPayPayment should be the name of your cassette dependent payment table
- FastPayCredit should be the name of your cassette dependent credit table
- FastPayBatch should be the name of your cassette dependent batch table.

If your table contains a column with a value that exactly matches a value in a framework database table (MerchantNumber or AccountNumber, for example) your column name should exactly match the column name in the framework table. It is also helpful if your primary key columns are first and ordered by significance (MerchantNumber, OrderNumber, then PaymentNumber for example).

It is important to remember that DB2 identifiers (table names, view names, column names, and so on) are limited to 18 characters. If you choose a cassette name that is longer than 8 characters, you will need to manually abbreviate some of the table names that are automatically generated for you by the cassette exercise.

Stage 4. Preparing the Base Files

This chapter describes the steps required to prepare the base files for your new cassette. It assumes that you have already performed the installation steps in Appendix A and Appendix B. Furthermore, it assumes that you have installed the cookbook materials on your **D:** drive in a directory named **D:\!PMCassette**. Note that the exclamation point cases the directory name to float to the top of an alphabetical directory list.

1. Use your favorite text editor and edit **D:\!PMCassette\CreateCassette.cmd**. Note that near the beginning of this script, the working directory is set to **D:\!PMCassette**.

2. Modify the line that sets the **NEWDIR** environment variable so that the value matches the directory chosen the **Stage 1** chapter.

3. Modify the line that sets the **NEWCASSETTE** environment variable so that the value matches the cassette name chosen in the **Stage 1** chapter.

4. Save the updated **CreateCassette.cmd**.

5. Execute the updated **CreateCassette.cmd**. This will create the raw base files for your new casstte.

6. If you are using Visual SlickEdit, this is a good time to create a project file for your cassette. After performing the sub-steps below, the Compile, Build, Rebuild, Make JAR, and Install tools in the Build menu list will operate on your base cassette files. Note that these tools all assume that the editor's current directory is the directory that contains your base files.

    1.. From the Visual SlickEdit menu bar, select Project, Open Workspace.

    2.. On the Open Workspace panel, navigate to D:\!PMCassette and select !PMCassette.vpw. Press Open.

    3.. From the Visual SlickEdit menu bar, select **Project**, **Workspace Properties**.

    4.. On the Workspace Properties panel, press Add.

    5.. On the Add files to workspace panel, navigate to D:\!PMCassette and select the file with the file name that matches your cassette name and the file type of .vpj.

    6.. Back on the Workspace Properties panel, select the project file you have just added and press Set Active.

    7.. Close the Workspace Properties panel.

    8.. From the Visual SlickEdit menu bar select Project, Project Properties.

    9.. On the Project Properties … panel, select the Open Command tab.

    10.. On the line that begins set target replace the LdbCard file name prefix with the name of your cassette.

    11.. On the line that begins set workdir change the com\ibm\etill\ldbcardcassette directory with the directory name that contains your new base files. Press OK to save the changes and close the panel. **Note:** the remaining steps are required to assure that the changes made on the Open

Command tab take effect.

12.. From the Visual SlickEdit menu bar, select Project, Close Workspace.

13.. From the Visual SlickEdit menu bar, select Project, Open Workspace.

14.. On the Open Workspace panel, navigate to D:\!PMCassette and select !PMCassette.vpw. Press Open.

7. Using your favorite editing tool, change all occurances of LdbCard (be sure to use this exact case) with the name of your cassette (be sure to specify the exact case) in every base file. Remember, your cassette name was determined in Stage 1. If you are using Visual SlickEdit, you can use these sub-steps.

   1.. If necessary, navigate to the directory that contains your cassette files and make it the current directory.

   2.. From the menu bar, select Search Replace.

   3.. On the Replace panel, press Files/Buffers>> to get the expanded Replace panel

   4.. On the expanded Replace panel, enter LdbCard in the Search for: text box, enter your cassette name in the Replace with text box, select the Match case option, enter *.* in the Files: text box, and press Replace All.

8. Using your favorite editing tool, change all occurances of LDBCARD (be sure to use this exact case) with the name of your cassette in **upper case** (be sure to specify the exact case) in every base file. Remember, your cassette name was determined in Stage 1. If you are using Visual SlickEdit, you can use these sub-steps.

   1.. If necessary, navigate to the directory that contains your cassette files and make it the current directory.

   2.. From the menu bar, select Search Replace.

   3.. On the Replace panel, press Files/Buffers>> to get the expanded Replace panel

   4.. On the expanded Replace panel, enter LDBCARD in the Search for: text box, enter your cassette name in upper case in the Replace with text box, select the Match case option, enter *.* in the Files: text box, and press Replace All.

9. Using your favorite editing tool, change all occurances of com\ibm\etill\ldbcardcassette (be sure to use this exact case) with the name of the directory for your casette (be sure to specify the exact case) in every base file. Remember, your directory name was determined in Stage 1. If you are using Visual SlickEdit, you can use these sub-steps.

   1.. If necessary, navigate to the directory that contains your cassette files and make it the current directory.

   2.. From the menu bar, select Search Replace.

   3.. On the Replace panel, press Files/Buffers>> to get the expanded Replace panel

4.. On the expanded Replace panel, enter com\ibm\etill\ldbcardcassette in the Search for: text box, enter your directory name in the Replace with text box, select the Match case option, enter *.* in the Files: text box, and press Replace All.

10. Using your favorite editing tool, change all occurances of com.ibm.etill.ldbcardcassette (be sure to use this exact case) with the package name for your casette (be sure to specify the exact case) in every base file. Remember, your package name was determined in Stage 1. If you are using Visual SlickEdit, you can use these sub-steps.

    1.. If necessary, navigate to the directory that contains your cassette files and make it the current directory.

    2.. From the menu bar, select Search Replace.

    3.. On the Replace panel, press Files/Buffers>> to get the expanded Replace panel

    4.. On the expanded Replace panel, enter com.ibm.etill.ldbcardcassette in the Search for: text box, enter your package name in the Replace with text box, select the Match case option, enter *.* in the Files: text box, and press Replace All.

11. Using your favorite editor, edit your cassette's #StartManager.cmd.

    1.. Find the line that sets the CASSETT_CLASSES environment variable and remove the ";\.japi.zip" from the end of the value set.

    2.. Find the line that sets the ENVIRONMENT environment variable and erase all the characters after the "=" character to the end of the line.

    3.. Find the line that sets the DBOWNER environment variable and replace "ldbader" with user ID you used to logon to the workstation.

    4.. Find the line that sets the DBUSER environment variable and replace "ldbader" with the user ID you used to logon to the workstatoin.

12. If you find it convenient to use shortcuts to execute command scripts, you will need to modify a few of the provided shortcuts so they will execute your cassette's command scripts instead of the LdbCard command scripts.

    1.. From the Start bar, select Start, Programs, Windows, Windows NT Explorer.

    2.. On the Exploring panel, navigate to the directory that contains your cassette and open the Shortcuts directory.

    3.. On the right hand file list, right click on your cassette's Build shortcut and select Properties on the pop-up menu

    4.. On the resulting Properties panel, select the Shortcut tab. In the Target: text field, find and replace LdbCard with the name of your cassette. Note: Do **not** change the working directory. By convention, your command scripts assume that the Payment Manager directory is the current directory. Press OK.

    5.. Repeat the previous two steps for each of the following shortcuts.

- Get Engine Trace
- Get Servlet Trace
- Install
- Remove
- Restore PostInstall
- Restore PreInstall
- Start Payment Manager (In addition, change f5lzvb to the password you use when you logon to the workstation.)
- Wipe Out
- Save DB (In addition, delete the & and all characters from there to the end of the line.)
- Test DB

6.. For convenience, you may want to copy these shortcuts or the ShortCuts directory to your desktop.

13. Using your favorite editor, edit your cassette's .Install.sql file. Examine all occurrences of your cassette name in upper case. If your cassette name is longer than 8 characters, you will need to abbreviate the table and view names so they are all less than 18 characters long.

14. Using your favorite editor, edit your cassette's .Remove.sql file. If your cassette name is longer than 8 characters, you will need to change the table and view names so they match the abbreviated names that you defined above.

15. Using your favorite editor, edit the file that contains your new account class.

16. Find and delete all the import statements that reference classes in a package that begins with "netverify" prefix.

17. Find and delete all the constants with a name that begins with a prefix of "ICV_" including the associated comment blocks.

18. Find the statement that defines objConnection. Change the statement

```
private Connection    objConnection
```

to

```
private Object        objConnection
```

19. Find the connect() method. Leave the blocks that trace the function entry and exit, but remove all the other statements (an entire try{}catch{} block).

20. Find the authorizePayment() method. Right after the block that traces the function entry, you will find an assignment statement for **varResult**. Change the statement

```
boolean varResult = false;
```

to

```
boolean varResult = true;
```

In the comment above the line you have just changed, change the word "declined" to "approved".

Then remove all the statements after this statement up to, but not including, the block that traces the function exit.

21. Repeat step 12 for each of the following methods

    - capturePayment()
    - voidPayment()
    - captureCredit()
    - voidCredit()
    - settleBatch()

22. Build your cassette and copy the updated files into the Payment Manager directories using your cassette's #Build.cmd script or its associated shortcut. Note: The first time you run this script, there are no .class files to delete and an error message will appear. Just ignore this message in this case.

23. Start your favorite browser and navigate to http://localhost:9527/.

24. On the **Login** page, enter **admin** as the user ID and **admin** as the password. Press **Log In**.

25. On the **WebSphere Application Server Administration** page, press **Setup** in the navigation panel to expand the setup tasks.

26. On the updated navigation panel, press **Java Engine**.

27. On the new **Java Engine** panel, click in the **Application Server Classpath** text field. Find the complete path to eTillClasses.zip and copy it to the clipboard (mark it and press Ctrl-C). Note that the short DOS names are used for directory names in this classpath. Move the cursor to the end of the string in this text field and add a semi-colon (;). Then paste the contents of the clipboard (Ctrl-V or right click and paste.) Replace eTillClasses.zip with the name of your cassette JAR file. Press **Save**.

28. On the navigation panel, press **Log Off**.

29. Close the browser, shutdown, and restart your workstation.

30. Install your cassette into the PaymentManager and create your cassette's database tables using your cassette's #Install script or its associated shortcut. Note: This script will automatically backup the Payment Manager database before it creates this version of the cassette database tables. When your cassette tables are incrementally refined, this backup must be restored with the #RestorePreInstall script or its associated shortcut.

31. Congratulations! You now have a primitive cassette. To start the payment manager with your cassette, use your cassette's #StartManager script or its associated shortcut. Use the Payment Manager user interface (from the task bar, press Start, Programs, WebSphere Payment Manager, Payment Manager Logon) to create a new merchant (by convention, use merchant number 100) that is authorized to use your cassette. The create an new account (by convention, use account number 1000) for that merchant to use your cassette). Follow the directions in Appendix D to setup a sample store and create a few orders. Use the Payment Manager user interface to approve them, deposit them, reverse them, refund them, settle the batch, and so on.

Stage 5: Implement Your Data Model

This chapter describes how to enhance your cassette to use the persistent data and protocol data that you designed in Stage 3. You will

- Update the SQL script that creates your database tables and views
- Update the XML document so the Payment Manager user interface can display your cassette dependent data
- Update the constants with your column names and field identifiers
- Update the CassetteQuery class to return your cassette dependent data
- Update the Cassette class to define the parameter validation for your cassette's protocol data.

1. Start your favorite text editor and edit your cassette's Install.sql file.

2. Find the CREATE TABLE command for your cassette's ACCOUNT table. Replace the column names and types with those that you designed in Stage 3. See Appendix F if you need tips on how to specify the data type for a column. Note that you will need to keep the existing columns for MerchantNumber and AccountNumber because they make up the primary key. Also note that it is the cassette's responsibility to make sure that the merchant does not supply an amount with the wrong currency, so if your cassette will every support more than one configurable currency, the Currency column is strongly recommended.

3. Find the CREATE VIEW command for your cassette's ACCOUNTVIEW. This view will be used in your cassette's CassetteQuery class. When performing a query, the merchant provides selection criteria based on the data kept in the framework's AccountAdmin objects. This view will be used inside a larger select statement to obtain the cassette dependent information for the same criteria. Replace the existing explicit column names in the SELECT clause so they match the column names created in the last step. Do **not** include the primary key columns. These columns will be provided by the framework's ETACCOUNTCFG view.

4. Repeat the last two statements for each of the following tables and their views. Note that payments and credits have two views: one for selecting them when examining framework payments/credits and one for examining framework batches.

   - Your cassette's BATCH table and the BATCHVIEW
   - Your cassette's ORDER table and the ORDERVIEW
   - Your cassette's PAYMENT table and both the PAYMENTVIEW and the PAYBATVIEW
   - Your cassette's CREDIT table and both the CREDITVIEW and the CREBATVIEW

5. Save the updated file. Use the #RestorePreInstall script to restore the database to its original state. Then run your cassettes #Install script or its associated shortcut to test your file. If, during the above exercise, you have change the name of a table or view, make sure you make the matching change to your cassette's Remove.sql file. In any case, this would be a good time to test your cassette's Remove.sql file using the #Remove script or its associated shortcut. It does not perform a complete test of the Remove.sql file, but it will assure you that the cassette dependent table and view names match.

6. Start your favorite text editor and edit your cassette's en.PSPL file. You will need to change all the field tags in this file so they will display and optionally update your cassette dependent data. The syntax of the field tag can be found in C:\Program Files\IBM\PaymentManager\pspl\pslp.dtd.

7. Find your cassette's AccountDetails field group. Using the existing field tags as a model, create field tags for your account data and then delete any tag that is no longer used. By convention, if the field is

taken directly from the database, the field's id attribute should match the column name from the database. If the field is used to obtain information from the merchant, the field's updateID attribute should specify the matching protocol data keyword that you chose in Stage 3. See Appendix F for tips on choosing the type and displayType attributes.

8. Repeat the last step for the following field groups. Most likely, the information displayed for these field groups cannot be modified by the merchant so the field's displayType attribute should be set to readOnly.

   - OrderDetails
   - BatchDetails
   - PaymentDetails
   - CreditDetails

9. Save this file for now. Later, the message tags in the Account screen will be updated to display the correct error message when the merchant does not enter the protocol data values correctly.

10. Start your favorite text editor and edit your cassette's Constants.java file.

11. Find the constants that have a TBL_ prefix. Make sure each constant is initialized to the correct cassette dependent database table name.

12. Find the constants that have a COL_ prefix. In this section, make sure there is a constant for every column name you have defined, no matter which table it is in. Constants organized by table will be updated in subsequent steps using the constants you define here. If there are any existing COL_ constants that you do not intend to use, copy them to the gutter area and declare them deprecated. This is required because they will continue to be referenced in a later stage. They must remain so your cassette can be compiled without error.

    Note that you will need both a COL_MERCHANT constant and a COL_MERCHANT_NAME constant. To remain compatible with previous versions of the Payment Manager, some of the framework views use a column name of MerchantName instead of the new, more appropriate, column name of MerchantNumber.

13. Find the constants that have an XDM_ prefix. In this section, make sure there is a constant for every field id attribute that you used in your cassette's .en.PSPL document. If there are any existing XDM_ constants you do not intend to use, copy them to the gutter and declare them deprecated.

14. Find the constants that have a prefix of VIEW_. The first seven parameters define constants for the cassette views. Make sure each is initialized to the correct cassette dependent view name. The remaining VIEW_ constants define the column names used in the view. Make sure there is a constant for each column in a view and that each initializer references the correct COL_ constant defined above. If there are any existing VIEW_ constants that you do not intend to use, copy them to the gutter and declare them deprecated.

15. Find the constants that have a prefix of PD_. Note that they are organized into two categories: PD_PAY_ for protocol data passed on the AcceptPayment command and PD_ACC_ for protocol data passed on CreateAccount or ModifyAccount commands. Make sure there is a PD_ constant for every protocol data keyword designed in Stage 3. Whenever possible, initialize the constant to one of the standard protocol data keyword defined in the framework. If there are any existing PD_ constants that you do not intend to use, copy them to the gutter and declare them deprecated.

16. Find the constants that have a prefix of SRC_. Note these secondary return code values are organized into different categories depending on the class where they are used. Each category has a unique range. Find the return codes in the 10000 range. Make sure there is an SRC_CASSETTE_ constant

for each cassette dependent protocol data value defined above. Note that you should not define a secondary return code value for any of the standard protocol data keywords defined by the framework. Your code should use the framework's matching secondary return code value instead.

17. Save the file. Build your cassette using the #Build script or its associated shortcut. Fix any build errors. Note that at this point, the cassette should compile correctly but cannot be tested.

18. Start your favorite text editor and edit your cassette's CassetteQuery.java file.

19. Find the inner class named OrderInfo.

20. Make sure this class has an instance variable for each column in the cassette dependent order table. Simply delete any existing instance variable that you will not use.

21. Modify the constructor to assign values to each of the instance variables from the appropriate column of the result set argument. Use the VIEW_ constants created above to specify the column name. Simply delete any existing statement that you do not intend to use. Note that you should not use the result set directly. Instead, use the correct EtillArchive class method to read the column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

22. Modify the combine() method so there is an .addProperty() method call for each instance variable that is not part of the primary key. Use the appropriate XML_ constants defined above to identify each field. Simply delete any existing addProperty() method call that you do not intend to use.

23. Repeat the last four steps for each of the following inner classes.

    - PaymentInfo
    - CreditInfo
    - BatchInfo
    - AccountInfo

24. Save the file. Build your cassette using the #Build script or its associated shortcut. Fix any build errors. Note that at this point, the cassette should compile correctly but cannot be tested.

25. Start your favorite text editor and edit your cassette's Cassette.java file.

26. Near the top of this file, there is a static block that initializes the cassette's protocol data validation table. Using the existing statements as a model, construct a framework ValidationItem for each protocol data keyword that your cassette uses and put it into the validation table using the protocol data keyword as a key. See Appendix F for tips on choosing the correct ValidationItem object for each protocol data keyword. Use the protocol data keyword constants that you have defined and either the correct framework secondary return code or the matching secondary return code value constant that you have defined. Simply remove any existing statement that you do not intend to use.

27. Save the file. Build your cassette using the #Build script or its associated shortcut. Fix any build errors. Note that at this point, the cassette should compile correctly and should even start correctly, but cannot be tested.

Stage 6: Implement Your Account Class

In this stage, you will implement about half of your account class and test the commands the create, modify, delete, start and stop an account. Additional methods will be added to the account objects in subsequent stages.

1. Start your favorite editor and edit your cassette's Constants.java file.

2. Find the constants that begin with an ACC_ prefix. Make sure there is a constant for each column name in your cassette account table. Initialize each constant to the matching COL_ constant defined in Stage 5. Simply delete any existing constant that you do not intend to use. After you update the Account.java file, these old values will not be used.

3. Save the file.

4. Start your favorite editor and edit your cassette's Account.java file.

5. Find the instance variables, that is, the variables with a name that begins with a prefix of obj. Skip past objFrameworkAccount and then make sure that there is an instance variable for each column in your cassette's account table **including** the primary key columns. If there are any existing instance variable that you do not intend to use, copy them to the gutter and declare them deprecated.

6. Examine all the existing methods with a prefix of set or get. If these setter and getter methods reference a deprecated instance variable, copy them to the gutter and declare them deprecated as well. At a minimum, you will be left with getMerchant(),and getAccount(), and perhaps getCurrency() as well. Consider adding setter and getter methods for your other instance variables. These methods are highly recommended, even if they are not public, because they offer a convenient place where the values can be transformed to different internal representations if necessary. For example, setters could be used to transform a value taken from the database into a more appropriate internal representation and getters could do the reverse.

7. Find the constructor that takes a ResultSet as an argument. Modify this constructor to assign values to each of the instance variables from the appropriate column. Consider using the setters created earlier to make the assignments. Use the ACC_ constants created above to specify the column name. Simply delete any existing assignment statement that you do not intend to use. Note that you should not use the result set directly. Instead, use the correct EtillArchive class method to read the column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

8. Find the constructor that takes an AdminRequest as an argument. Modify this constructor to assign values to each of the instance variables from values taken from the request or values taken from the protocol data parameter table provided with the request. Consider using the setters created earlier to make the assignments. Simply delete any existing assignment statement that you do not intend to use. Note that there are no validity checks done in this constructor. The validity checks will be performed in a different method.

9. Find the initializeAccount() method. Remove any existing statements that you do not intend to use. If necessary, add any statements required to initialize your account regardless of how it was constructed. This is a good place to initialize instance variables that are computed from other instance variables **that connate be subsequently modified.**. For values computed from instance variables that can changed by a ModifyAccount command, the values should be initialized in the startAccount() method.

10. Find the createRecord() method.

1.. Change varInsert declaration so that it is initialized to the SQL INSERT statement required to add a row to your cassette's account table. Use the ACC_ constants for column names.

2.. Change the Trace.traceDatabaseWrite() method call to trace varInsert and all the values that will be set into it.

3.. Make sure there is a statement that adds a value to the prepared statement for each column in the database. Simply remove any existing statement that you do not intend to use. Note that you should not use the prepared statement directly. Instead, use the correct EtillArchive class method to add the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

11. Find the updateRecord() method.

    1.. Change varUpdate declaration so that it is initialized to the SQL UPDATE statement required to modify a row to your cassette's account table. Use the ACC_ constants for column names.

    2.. Change the Trace.traceDatabaseWrite() method call to trace varUpdate and all the values that will be set into it.

    3.. Make sure there is a statement that adds a value to the prepared statement for each column in the database. Simply remove any existing statement that you do not intend to use. Note that you should not use the prepared statement directly. Instead, use the correct EtillArchive class method to add the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

12. Find the modifyAccount() method. Modify this method to assign values to each of the instance variables from values taken from the protocol data parameter table provided with the request. If a given protocol data keyword is not in the parameter table, its instance data should not be modified. Consider using the setters created earlier to make the assignments. Simply delete any existing assignment statement that you do not intend to use. Note that there are no validity checks done in this constructor. The validity checks will be performed in a different method.

13. Find the connect() and disconnect() methods. These are private methods that are called from the startAccount() and stopAccount() methods respectively. If there is connection activity associated with your payment appliance, add the appropriate code to these two methods. If not, consider removing these methods and their calls in startAccount() and stopAccount().

14. Find the startAccount() and stopAccount(). Modify the startAccount() method to initialize any instance variable value that is computed from instance variables that can be modified by a ModifyAccount method. In addition, consider adding any logic required to start or stop your accounts interface to your payment appliance.

15. Find the verifyAccounts() class method. If there is logic required to verify the account configuration when the account object have been resurrected from the database during Payment Manager initialization, add the logic here. Note that this is specifically for logic that cannot be performed in the constructor because the framework has not been completely initialized when the constructor is called.

16. Find the initializeAccounts() class method. If there is logic required to initialize the account objects resurrected from the database during Payment Manager initialization, add the logic here. As above, this is specifically for logic that cannot be performed in the constructor because the framework has not been completely initialized when the constructor is called. If the initialization logic is limited to the scope of a single account object, it is more appropriate to place this logic in the startAccount()

method. This method is better suited to initialization for the payment appliance or other cassette dependent components with a global scope.

17. Find the validateCreateRequest() class method. This method is used to validate the protocol data passed on a CreateAccount command. Keep in mind that the automatic parameter validation performed by the framework using the validation table you created in Stage 5 cannot perform existence checks, consistency checks, or compatibility checks. Using the existing logic as model, add the logic to abort the request if a required parameter is not supplied. Simply remove any existing statements that you do not intend to use. Then, if necessary, add any logic required to abort the request if there are any inconsistencies or incompatibilities between the protocol data values supplied.

18. Find the validateModifyRequest() class method. This method is used to validate the protocol data passed on a ModifyAccount command. Keep in mind that the automatic parameter validation performed by the framework using the validation table you created in Stage 5 cannot perform existence checks, consistency checks, or compatibility checks. Using the existing logic as model, add the logic to abort the request if a supplied parameter cannot be supplied on a ModifyAccount request. For example, changing the Currency attribute of the account would can not be allowed because there may be existing orders that use the account with the current currency. Simply remove any existing statements that you do not intend to use. Then, if necessary, add any logic required to abort the request if there are any inconsistencies or incompatibilities between the protocol data values supplied.

19. Save the file. Build your cassette using the #Build script or its associated shortcut. Fix any build errors. Start the Payment Manager engine using your #StartManager script or its associated shortcut. Now you should be able to test your code. Use the Payment Manager user interface to create, delete, modify, and display accounts. Examine the payment engine error logs and traces to make sure your code is working correctly. Start and stop the Payment Manager engine to make sure existing accounts are resurrected correctly.

Stage 7: Implement Your Batch Class

In this step, you will implement your batch class. Although batch operations cannot be tested here, they will be tested when payment and credits are added to the batch in subsequent steps.

1. Start your favorite editor and edit your cassette's Constants.java file.

2. Find the constants that begin with an BAT_ prefix. Make sure there is a constant for each column name in your cassette batch table. Initialize each constant to the matching COL_ constant defined in Stage 5. Simply delete any existing constant that you do not intend to use. After you update the Batch.java file, these old values will not be used.

3. Save the file.

4. Start your favorite editor and edit your cassette's Batch.java file.

5. Find the instance variables, that is, the variables with a name that begins with a prefix of obj. Skip past objCassette, objFrameworkBatch, and objCassetteAccount. Then make sure that there is an instance variable for each column in your cassette's batch table **excluding** the primary key columns. (Primary key values can always be obtained from the framework batch object.) If there are any existing instance variables that you do not intend to use, copy them to the gutter and declare them deprecated.

6. Examine all the existing methods with a prefix of set or get. If these setter and getter methods reference a deprecated instance variable, copy them to the gutter and declare them deprecated as well. Consider adding setter and getter methods for your instance variables. These methods are highly recommended, even if they are not public, because they offer a convenient place where the values can be transformed to different internal representations if necessary. For example, setters could be used to transform a value taken from the database into a more appropriate internal representation and getters could do the reverse.

7. Find the resurrectBatch() method.

    1.. Examine the varSelect declaration and note how it selects a row from the cassette batch table. Ordinarily no changes are required.

    2.. Examine the Trace.traceDatabaseRead() method call and note how it traces the varSelect statement. Ordinarily no changes are required.

    3.. After the statement `if (varResult.next())`, make sure that each instance variable created above is initialized from the appropriate column of the result set. Simply remove any existing statement that you do not intend to use. Use the BAT_ constants for column names. Note that you should not use the result set directly. Instead, use the correct EtillArchive class method to read the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

8. Find the createRecord() method.

    1.. Change varInsert declaration so that it is initialized to the SQL INSERT statement required to add a row to your cassette's batch table. Use the BAT_ constants for column names.

    2.. Change the Trace.traceDatabaseWrite() method call to trace varInsert and all the values that will be set into it.

3.. Make sure there is a statement that adds a value to the prepared statement for each column in the database. Simply remove any existing statement that you do not intend to use. Note that you should not use the prepared statement directly. Instead, use the correct EtillArchive class method to add the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

9. Find the updateRecord() method.

1.. Change varUpdate declaration so that it is initialized to the SQL UPDATE statement required to modify a row to your cassette's batch table. Use the BAT_ constants for column names.

2.. Change the Trace.traceDatabaseWrite() method call to trace varUpdate and all the values that will be set into it.

3.. Make sure there is a statement that adds a value to the prepared statement for each column in the database. Simply remove any existing statement that you do not intend to use. Note that you should not use the prepared statement directly. Instead, use the correct EtillArchive class method to add the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

10. Find the batchClose() method. Note the call to objCassetteAccount.settleBatch(). If your payment appliance does not support the batch close there is nothing more to do. The existing code will implement a cosmetic batch for your cassette. If batch close is supported, modify this logic to use your account object to take the appropriate action. You are free to change the method name, the parameters, and anything else as long as the all the other statements in this method are left to do the correct bookkeeping.

11. Save the file.

12. If your cassette supports batch close, use your favorite text editor to edit your cassette's Account.java file. Replace its settleBatch() method with whatever is expected by your Batch class. Implement the logic to exercise your payment appliance and close the batch. Use the batch object's getters and setters to obtain and update cassette dependent batch information.

13. Save the files. Build your cassette using the #Build script or its associated shortcut. Fix any build errors. Further testing must wait until payments or credits are added to the batch.

Stage 8: Implement Your Order Class

In this step, you will implement your order class and test the AcceptPayment command.

1. Start your favorite editor and edit your cassette's Constants.java file.

2. Find the constants that begin with an ORD_ prefix. Make sure there is a constant for each column name in your cassette order table. Initialize each constant to the matching COL_ constant defined in Stage 5. Simply delete any existing constant that you do not intend to use. After you update the Order.java file, these old values will not be used.

3. Save the file.

4. Start your favorite editor and edit your cassette's Order.java file.

5. Find the instance variables, that is, the variables with a name that begins with a prefix of obj. Skip past objCassette, objFrameworkOrder, and objCassetteAccount. Then make sure that there is an instance variable for each column in your cassette's order table **excluding** the primary key columns. (Primary key values can always be obtained from the framework order object.) If there are any existing instance variables that you do not intend to use, copy them to the gutter and declare them deprecated.

6. Examine all the existing methods with a prefix of set or get. If these setter and getter methods reference a deprecated instance variable, copy them to the gutter and declare them deprecated as well. Consider adding setter and getter methods for your instance variables. These methods are highly recommended, even if they are not public, because they offer a convenient place where the values can be transformed to different internal representations if necessary. For example, setters could be used to transform a value taken from the database into a more appropriate internal representation and getters could do the reverse.

7. Find the constructor that takes three arguments including a ParameterTable. This method is used to validate the protocol data passed on an AcceptPayment command. Keep in mind that the automatic parameter validation performed by the framework using the validation table you created in Stage 5 cannot perform existence checks, consistency checks, or compatibility checks. Using the existing logic as model, add the logic to sent the instance variables from the protocol data values supplied and abort the request if a required parameter is not supplied. Simply remove any existing statements that you do not intend to use. Then, if necessary, add any logic required to abort the request if there are any inconsistencies or incompatibilities between the protocol data values supplied.

8. Find the resurrectOrder() method.

    1.. Examine the varSelect declaration and note how it selects a row from the cassette order table. Ordinarily no changes are required.

    2.. Examine the Trace.traceDatabaseRead() method call and note how it traces the varSelect statement. Ordinarily no changes are required.

    3.. After the statement `if (varResult.next())`, make sure that each instance variable created above is initialized from the appropriate column of the result set. Simply remove any existing statement that you do not intend to use. Use the ORD_ constants for column names. Note that you should not use the result set directly. Instead, use the correct EtillArchive class method to read the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

9. Find the createRecord() method.

    1.. Change varInsert declaration so that it is initialized to the SQL INSERT statement required to add a row to your cassette's order table. Use the ORD_ constants for column names.

    2.. Change the Trace.traceDatabaseWrite() method call to trace varInsert and all the values that will be set into it.

    3.. Make sure there is a statement that adds a value to the prepared statement for each column in the database. Simply remove any existing statement that you do not intend to use. Note that you should not use the prepared statement directly. Instead, use the correct EtillArchive class method to add the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

10. Find the updateRecord() method.

    1.. Change varUpdate declaration so that it is initialized to the SQL UPDATE statement required to modify a row to your cassette's order table. Use the BAT_ constants for column names.

    2.. Change the Trace.traceDatabaseWrite() method call to trace varUpdate and all the values that will be set into it.

    3.. Make sure there is a statement that adds a value to the prepared statement for each column in the database. Simply remove any existing statement that you do not intend to use. Note that you should not use the prepared statement directly. Instead, use the correct EtillArchive class method to add the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

11. Find the acceptPayment() method. Note the logic that assures that the order is using a currency that the account supports. If necessary, change this logic to whatever is appropriate for your cassette. If necessary add additional runtime checks and abort the request if your cassette cannot process payments and credits for the new order. No other changes should be necessary. The remaining logic is for bookkeeping.

12. Save the files. Build your cassette using the #Build script or its associated shortcut. Fix any build errors. Start the Payment Manager using the #StartManager script or its associated shortcut. Use the Sample Store (see Appendix E) to issue AcceptPayment requests and test your Order class. In addition, use the Payment Manager user interface to display and cancel your orders.

Stage 9: Implement Your Payment Class

In this step, you will implement your payment class and test the Approve, ApproveReversal, Deposit, and Deposit reversal commands. In addition, the BatchClose and CloseOrder commands can be tested.

1. Start your favorite editor and edit your cassette's Constants.java file.

2. Find the constants that begin with a PAY_ prefix. Make sure there is a constant for each column name in your cassette payment table. Initialize each constant to the matching COL_ constant defined in Stage 5. Simply delete any existing constant that you do not intend to use. After you update the Payment.java file, these old values will not be used.

3. Save the file.

4. Start your favorite editor and edit your cassette's Payment.java file.

5. Find the instance variables, that is, the variables with a name that begins with a prefix of obj. Skip past objFrameworkPayment. Then make sure that there is an instance variable for each column in your cassette's payment table **excluding** the primary key columns. (Primary key values can always be obtained from the framework payment object.) If there are any existing instance variables that you do not intend to use, copy them to the gutter and declare them deprecated.

6. Examine all the existing methods with a prefix of set or get. If these setter and getter methods reference a deprecated instance variable, copy them to the gutter and declare them deprecated as well. Consider adding setter and getter methods for your instance variables. These methods are highly recommended, even if they are not public, because they offer a convenient place where the values can be transformed to different internal representations if necessary. For example, setters could be used to transform a value taken from the database into a more appropriate internal representation and getters could do the reverse.

7. Find the resurrectPayment() method.

    1.. Examine the varSelect declaration and note how it selects a row from the cassette payment table. Ordinarily no changes are required.

    2.. Examine the Trace.traceDatabaseRead() method call and note how it traces the varSelect statement. Ordinarily no changes are required.

    3.. After the statement `if (varResult.next())`, make sure that each instance variable created above is initialized from the appropriate column of the result set. Simply remove any existing statement that you do not intend to use. Use the PAY_ constants for column names. Note that you should not use the result set directly. Instead, use the correct EtillArchive class method to read the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

8. Find the createRecord() method.

    1.. Change varInsert declaration so that it is initialized to the SQL INSERT statement required to add a row to your cassette's payment table. Use the PAY_ constants for column names.

    2.. Change the Trace.traceDatabaseWrite() method call to trace varInsert and all the values that will be set into it.

3.. Make sure there is a statement that adds a value to the prepared statement for each column in the database. Simply remove any existing statement that you do not intend to use. Note that you should not use the prepared statement directly. Instead, use the correct EtillArchive class method to add the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

9. Find the updateRecord() method.

   1.. Change varUpdate declaration so that it is initialized to the SQL UPDATE statement required to modify a row to your cassette's payment table. Use the PAY_ constants for column names.

   2.. Change the Trace.traceDatabaseWrite() method call to trace varUpdate and all the values that will be set into it.

   3.. Make sure there is a statement that adds a value to the prepared statement for each column in the database. Simply remove any existing statement that you do not intend to use. Note that you should not use the prepared statement directly. Instead, use the correct EtillArchive class method to add the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

10. Find the approve() method. Note the call to varAccount.authorizePayment() and varAccount.depositPayment(). Modify this logic to use your account object to take the appropriate action. You are free to change the method name, the parameters, and anything else as long as the all the other statements in this method are left to do the correct bookkeeping.

11. Find the approveReversal() method. Note the call to varAccount.voidPayment() and varAccount.authorizePayment(). Modify this logic to use your account object to take the appropriate action. You are free to change the method name, the parameters, and anything else as long as the all the other statements in this method are left to do the correct bookkeeping.

12. Find the deposit() method. Note the call to varAccount.capturePayment(). Modify this logic to use your account object to take the appropriate action. You are free to change the method name, the parameters, and anything else as long as the all the other statements in this method are left to do the correct bookkeeping.

13. Find the depositReversal() method. Note the call to varAccount.voidPayment() and varAccount.authorizePayment(). Modify this logic to use your account object to take the appropriate action. You are free to change the method name, the parameters, and anything else as long as the all the other statements in this method are left to do the correct bookkeeping.

14. Save the file.

15. Use your favorite text editor to edit your cassette's Account.java file. Replace its authorizePayment(), voidPayment(), and capturePayment() method with whatever is expected by your Payment class. Implement the logic to exercise your payment appliance and perform the required action. Use the getters and setters of the cassette order object and the cassette payment object to obtain and update cassette dependent information.

16. Save the files. Build your cassette using the #Build script or its associated shortcut. Fix any build errors. Start the Payment Manager using the #StartManager script or its associated shortcut. Use the Sample Store (see Appendix E) to and the Payment Manager user interface to create orders, approve payments, deposit payments, void payments, close batches, purge batches, close orders, delete orders,

and delete batches.

Stage 10: Implement Your Credit Class

In this step, you will implement your credit class and test the Refund and RefundReversal commands. In addition, the BatchClose and CloseOrder commands will be tested again.

1. Start your favorite editor and edit your cassette's Constants.java file.

2. Find the constants that begin with a CRE_ prefix. Make sure there is a constant for each column name in your cassette payment table. Initialize each constant to the matching COL_ constant defined in Stage 5. Simply delete any existing constant that you do not intend to use. After you update the Credit.java file, these old values will not be used.

3. Save the file.

4. Start your favorite editor and edit your cassette's Credit.java file.

5. Find the instance variables, that is, the variables with a name that begins with a prefix of obj. Skip past objFrameworkCredit. Then make sure that there is an instance variable for each column in your cassette's credit table **excluding** the primary key columns. (Primary key values can always be obtained from the framework credit object.) If there are any existing instance variables that you do not intend to use, copy them to the gutter and declare them deprecated.

6. Examine all the existing methods with a prefix of set or get. If these setter and getter methods reference a deprecated instance variable, copy them to the gutter and declare them deprecated as well. Consider adding setter and getter methods for your instance variables. These methods are highly recommended, even if they are not public, because they offer a convenient place where the values can be transformed to different internal representations if necessary. For example, setters could be used to transform a value taken from the database into a more appropriate internal representation and getters could do the reverse.

7. Find the resurrectCredit() method.

    1.. Examine the varSelect declaration and note how it selects a row from the cassette credit table. Ordinarily no changes are required.

    2.. Examine the Trace.traceDatabaseRead() method call and note how it traces the varSelect statement. Ordinarily no changes are required.

    3.. After the statement `if (varResult.next())`, make sure that each instance variable created above is initialized from the appropriate column of the result set. Simply remove any existing statement that you do not intend to use. Use the CRE_ constants for column names. Note that you should not use the result set directly. Instead, use the correct EtillArchive class method to read the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

8. Find the createRecord() method.

    1.. Change varInsert declaration so that it is initialized to the SQL INSERT statement required to add a row to your cassette's credit table. Use the CRE_ constants for column names.

    2.. Change the Trace.traceDatabaseWrite() method call to trace varInsert and all the values that will be set into it.

3.. Make sure there is a statement that adds a value to the prepared statement for each column in the database. Simply remove any existing statement that you do not intend to use. Note that you should not use the prepared statement directly. Instead, use the correct EtillArchive class method to add the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

9. Find the updateRecord() method.

   1.. Change varUpdate declaration so that it is initialized to the SQL UPDATE statement required to modify a row to your cassette's credit table. Use the CRE_ constants for column names.

   2.. Change the Trace.traceDatabaseWrite() method call to trace varUpdate and all the values that will be set into it.

   3.. Make sure there is a statement that adds a value to the prepared statement for each column in the database. Simply remove any existing statement that you do not intend to use. Note that you should not use the prepared statement directly. Instead, use the correct EtillArchive class method to add the value for each column. These methods are provided to remove platform and database inconsistencies in these operations. See Appendix F for tips on how to select the correct class method for your column.

10. Find the refund() method. Note the call to varAccount.captureCredit(). Modify this logic to use your account object to take the appropriate action. You are free to change the method name, the parameters, and anything else as long as the all the other statements in this method are left to do the correct bookkeeping.

11. Find the refundReversal() method. Note the call to varAccount.voidCredit(). Modify this logic to use your account object to take the appropriate action. You are free to change the method name, the parameters, and anything else as long as the all the other statements in this method are left to do the correct bookkeeping.

12. Save the file.

13. Use your favorite text editor to edit your cassette's Account.java file. Replace its captureCredit() and voidCredit methods with whatever is expected by your Credit class. Implement the logic to exercise your payment appliance and perform the required action. Use the getters and setters of the cassette order object and the cassette credit object to obtain and update cassette dependent information.

14. Save the files. Build your cassette using the #Build script or its associated shortcut. Fix any build errors. Start the Payment Manager using the #StartManager script or its associated shortcut. Use the Sample Store (see Appendix E) to and the Payment Manager user interface to create orders, approve payments, deposit payments, void payments, refund, void refunds, close batches, purge batches, close orders, delete orders, and delete batches.

Stage 11: Test Your Cassette

In this stage, you will test your cassette.

The directory **D:\!PMCassette\PayGenTestCases** contains all the PayGen test scripts used to perform regression testing on the LdbCardCassette. Although there are only 8 scripts that exercise the cassette, nearly 70% of the cassettes code is tested. Note that the file type of a PayGen script is **.ps**. To execute a given script, you simply pass its file name as a parameter to the batch file that is customized in step 1.

1. Using your favorite text editor, edit **D:\!PMCassette\PayGen\LdbCardTest0.bat**. If necessary, modify the path to **java.exe** and the **classpath** values so they are suitable for your environment. Save the file using a file name that you will recognize. For example, if your cassette name is **FastPay** you could choose a name like **FastPayTest0.bat**. The **0** suffix is convenient because, to run multiple independent test scenarios at the same time, you will need a unique batch file for each active scenario

2. Using your favorite text editor, edit **D:\!PMCassette\PayGen\LdbCardTest0.properties**. Modify the **HOSTNAME** property value to match the host name where your Payment Manager test system is running. Save the file using the same file name that you used in the step above with a file type of **.properties**. PayGen finds the correct properties file by matching the name of the batch file that invokes the PayGen application.

3. Test your batch file and your properties file by running the test script named **About.ps**.

4. Update and execute the other test scripts in **D:\!PMCassette\PayGenTestCases** to make them suitable for your cassette. To update the scripts,

    - Find and modify every **:StartTest** clause that contains a **CreateAccount** command. Change the value of the **CassetteName** parameter and modify the protocol data parameter names and values to match those necessary for your cassette.
    - Find and modify every **:StartTest** clause that contains an **AcceptPayment** command. Change the value of the **PaymentType** parameter and modify the protocol data parameter names and values to match those necessary for your cassette.
    - Find and modify every **:StartTest** clause **:ExpectedString-<CassetteProperty …** statement and a **:ExpectedNoString-<CassetteProperty…** statement. Remove any statement that tests for cassette dependent values that your cassette does not supply. Add additional statements to test the cassette dependent values that you do supply.
    - If your cassette does not support the **MaxBatchSize** design, you will also need to modify the **D:\!PMCassette\PayGenTestCases\OrderExercise.ps** file by adding explicit **BatchClose** commands where the current script expects the batch to be closed automatically.

5. Before this stage, your cassette was tested using the Sample Store and the Payment Manager user interface but this methodology is not sufficient. There will always be some error case that cannot be tested and automation is required for reliable and repeatable testing. Using the scripts in in **D:\!PMCassette\PayGenTestCases** as an example, design, create, and execute additional PayGen test scripts to fully test your cassette. For convenience, you may want to make a hardcopy of the file **D:\PMCassette\PayGen\Readme.txt**. This file contains all the PayGen documentation.

    - Be sure to validate that the database tables are updated correctly. Since the Query command examines the database, not the objects in memory, it can be used to automate this test.
    - Be sure to validate the primary and secondary return code values returned.
    - Be sure to validate that the framework object states change correctly.
    - Be sure to validate that the state checks are performed correctly.
    - Be sure to test every supported platform.

Stage 12: Package Your Cassette

In this stage, you will complete your cassette.

1. Find all the code that you have copied to a gutter and delete it. Rebuild your cassette to verify that the code is no longer needed. Fix any build errors.

2. Build an install package for every supported platform and test them. The Cassette Kit Programmer's Guide in the IBM WebSphere Payment Manager Cassette Developer Toolkit contains information that may help you with this task.

3. Use e-mail and tell this author what you liked and what you didn't like about this cookbook.

4. Congratulate yourself, you are ready to ship the cassette to your customers.

Appendix A: Installing the Java Development Kit

This appendix describes the steps necessary to install a Java Development Kit (JDK) on the Microsoft Windows NT workstation where you will be developing your cassette**. If you have already installed a JDK on your workstation, you still must perform steps 9 and beyond.**

1. Start your favorite browser and navigate to your favorite Java JDK supplier (for example, **http://java.sun.com**, or if you are an IBM employee, **http://w3.hursley.ibm.com/java**) and obtain the Java Development Kit (JDK) Version 1.1.x where x is 7 or larger. (**Note:** JDK 1.2.x is not supported for cassette development.)

2. Open the file obtained in the previous step. The following steps assume that you are using the Install Shield JDK package for Win32 systems provided by IBM. The panels displayed by the package you selected may be different than those discussed below, but the following should still provide a valuable guide.

3. On the **InstallShield Self-extracting EXE** panel, press **Yes**. A progress bar appears while the files are extracted and a setup program will start automatically.

4. On the resulting **Welcome** panel, press **Next**.

5. On **the Software License Agreement** panel, press **Yes** after reviewing the agreement.

6. On the **Select Components** panel, accept the default components at the minimum. Use the default destination directory if possible. Subsequent steps assume that the JDK is installed in **C:\jdk1.1.8**. In any case, record the destination directory for later use. Press **Next**.

7. On the **Start Copying Files** panel, review the settings and press **Next** if acceptable. Otherwise return to the previous panels and correct the settings. A progress bar will be displayed as the components are installed.

8. On the **Setup Complete** panel, uncheck the **Yes, I want to view…** and press **Finish**.

9. On the task bar, press **Start**, **Settings**, **Control Panel**.

10. On the **Control Panel** panel, open **System**.

11. On the **System Properties** panel, select the **Environment** tab. Select any variable in **the System Variables:** list box. (Note: This is required to ensure that the JAVA_HOME environment variable about to be created will be placed in the system's environment list. The JAVA_HOME variable must be available no matter who is logged on to the workstation.) Enter **JAVA_HOME** in the **Variable** text box and the JDK destination directory, which you recorded above, in the **Value** text box. Press **SET**. (Note: if you press OK before you press Set, your changes will be lost) The new environment variable should now appear in the **System Variables:** list box. Press **OK**.

Appendix B: Installing the Payment Manager

This appendix describes the steps required to install the IBM WebSphere Payment Manager on the Microsoft Windows NT workstation where you will be developing your cassette.

1. Put the IBM WebSphere Payment Manager 2.1 Windows NT CD in your drive.

2. From the task bar, press **Start**, **Run**.

3. On the **Run** panel, press **Browse**.

4. On the **Browse** panel, navigate to the CD drive and select **install.cmd**. Press **Open**

5. Back on the **Run** panel, press **OK**.

6. On the **Payment Manager Install** panel, press **Next**.

7. On the **Software License Agreement** panel, press **Accept** if the terms are acceptable. The installation program will check for running applications that may cause installation conflicts.

8. On the **Choose Destination Location**, use the default destination directory if at all possible. Subsequent steps assume that the payment manager is installed in **C:\Program Files\IBM\PaymentManager**.

9. On the **Select Web Server Option** panel, you should see a message informing you that the IBM WebSphere Application Server products were not detected and will be installed and configured. Press **Next**.

10. On the **Select Database to Install** panel, select **Install IBM Universal Database 5.2** and press **Next**.

11. On the **Payment Manager Database Access Information** panel, the user ID used to logon to the workstation should already be displayed along with the default Payment Manager Database name (psadmin). Enter the appropriate password and press **Next**.

12. On the **Payment Manager Configuration Information** panel, use the default TCP port if at all possible. Subsequent steps assume that the payment manager listens to port 8611. Press **Next**.

13. On the **Payment Engine Shortcut** panel, select **Create a Payment Engine shortcut** and press **Next**.

14. On the **Installation Summary**, review the options selected. Use Back to return to a previous panel if you need to correct a value. When everything is acceptable, press **Next**. An **Install Progress** panel will appear with a progress bar. After a preliminary step, IBM Universal Database DB2 will be installed followed by the IBM WebSphere Application Server. Finally, the Payment Manager will be installed and its database tables will be created.

15. On the **IBM WebSphere Payment Manager …** panel, press **Next**. Your default web browser will be started and the README file will be displayed. Review the file, print the file if you deem it necessary, and close the web browser.

16. On the **Information Panel**, note the message and press **OK**.

17. Close all the open windows.

18. On the **Task Bar**, select **Start**, **Shutdown**.

19. On the **Shut Down Windows** panel, select **Restart the computer?** and press **Yes**.

20. When the workstation restarts, logon using the same user ID and password you used when installing the Payment Manager.

21. On the **IBM Software Registration Tool** panel, press **Next**.

22. On the subsequent **IBM Software Registration Tool** panel, enter values to identify yourself and press **Next**.

23. On the subsequent **IBM Software Registration Tool** panel, enter values to identify your organization and press **Next**.

24. On the subsequent **IBM Software Registration Tool** panel, enter the country of your mailing address and press **Next**.

25. On the subsequent **IBM Software Registration Tool** panel, enter your mailing address and press **Next**.

26. On the subsequent **IBM Software Registration Tool** panel, enter your telephone numbers and e-mail address. Press **Next**.

27. On the subsequent **IBM Software Registration Tool** panel, select the benefits desired and if you care to, agree to provide customer survey responses. If you have previously registered this software, in other words, this is not the first time you have installed this software, it is recommended that you choose the **"I do not wish to receive…"** option and do not agree to answer customer survey questions.

28. On the subsequent **IBM Software Registration Tool** panel, select the desired registration option. If you have previously registered this software, it is recommended that you choose the Telephone option.

29. On the subsequent **IBM Software Registration Tool** panel, follow the directions. If you have previously registered this software and have followed the recommendations above, Your panel will have a text field for the registration number. If you know your registration number, enter it here. Otherwise, just enter 5. Press **Next**.

30. On the final **IBM Software Registration Tool** panel, press **Exit**.

31. On the **Task Bar**, select **Start**, **Settings**, **Control Panel**.

32. On the **Control Panel** panel, open **Services**.

33. On the **Services** panel, the following services should show a status of **Active** and a Startup value of **Automatic**.

    - DB2 – DB2
    - DB2 - DB2DAS00
    - IBM HTTP Administration
    - IBM HTTP Server

34. On the same **Services** panel, the WebSphere Servlet Service should have blank Status value and a Startup value of Manual. This is correct because this service is integrated with the IBM HTTP Server.

35. Close the **Services** panel and the **Control Panel** panel.

Appendix C: Practice with LdbCard

The LdbCard cassette was written to use a payment appliance developed by IBM but never released as a product. This payment appliance was chosen mostly because it was convenient but also because it illustrates most of the implementation details you are likely to find in your own development project. However, there are legal restrictions that prohibit the distribution of this appliance outside of my team.

If you would like to use LdbCard as a practice cassette, the following steps describe how modify the LdbAccount class to make it independent of the payment appliance.

1. Start your favorite editor and edit LdbCardAccount.class.

2. Find the connect() method. Leave the blocks that trace the function entry and exit, but remove all the other statements.

3. Find the authorizePayment() method. Right after the block that traces the function entry, you will find an assignment statement for **varResult**. Change the statement

   ```
   boolean varResult = false;
   ```

   to

   ```
   boolean varResult = true;
   ```

   Then remove all the statements after this statement up to, but not including, the block that traces the function exit.

4. Repeat step 3 for each of the following methods.
   - capturePayment()
   - voidPayment()
   - captureCredit()
   - voidCredit()
   - settleBatch()

5. Recompile the LdbCard cassette using the Rebuild tool in the Visual SlickEdit project or with the following commands.

   ```
   D:
   CD D:\!PMCassette
   SET JDKDIR=C:\jdk1.1.8
   SET WORKDIR=com\ibm\etill\ldbcardcassette
   SET ETILLDIR=C:\Program Files\IBM\PaymentManager
   SET CLASSPATH=C:\JDK1.1.8\lib\classes.zip
   SET CLASSPATH=%ETILLDIR%\eTillClasses.zip;%CLASSPATH%
   SET CLASSPATH=%WORKDIR%\lib\japi.zip;%CLASSPATH%
   SET CLASSPATH=.\;%CLASSPATH%
   ERASE %WORKDIR%\*.class
   %JDKDIR%\bin\javac %WORKDIR%\LdbCardCassette.java
   %JDKDIR%\bin\javac %WORKDIR%\LdbCardCassetteQuery.java
   ```

6. Rebuild the LdbCardCassette.jar using the **Make JAR** tool in the Visual SlickEdit project or with the following commands.

```
D:
CD D:\!PMCassette
SET JDKDIR=C:\jdk1.1.8
SET WORKDIR=com\ibm\etill\ldbcardcassette
SET TARGET=LdbCardCassette.jar
XCOPY %WORKDIR%\*.properties .\
%JDKDIR%\bin\jar cf0 %TARGET% %WORKDIR%*.class *.properties
ERASE *.properties
COPY %TARGET% %WORKDIR%\lib
```

7. Use your favorite text editor to edit **LdbCard#StartManager.cmd**. Find all instances of the string "ldbader" and replace them with the workstation user ID used to install the Payment Manager. Note that this command file takes one parameter (find "%1"), the password used to authenticate this user ID.

8. Copy the LdbCard files into the production system using the **Install** tool in the Visual SlickEdit project or with the following commands.

```
D:
CD D:\!PMCassette
SET WORKDIR=com\ibm\etill\ldbcardcassette
SET ETILLDIR=C:\Program Files\IBM\PaymentManager
XCOPY %WORKDIR%\*.cmd "%ETILLDIR%"
XCOPY %WORKDIR%\*.sql "%ETILLDIR%\Archive"
XCOPY %WORKDIR%\*.PSPL "%ETILLDIR%\pspl"
XCOPY %WORKDIR%\lib\*.* "%ETILLDIR%"
```

9. Install the LdbCard cassette into your production system by opening a DB2 command window (**Start, Programs, DB2 for Windows NT, Command Windos)** and executing the LdbCard#Install.cmd script that was copied into your production system in step 8 in the resulting command window.

10. Start the Payment Manager using the LdbCard#StartManager.cmd script that was copied into your production system in step 8. Be sure to specify a parameter, namely the password necessary to authenticate the user ID you placed into the command file in step 7.

11. From the task bar, select **Start**, **Programs**, **WebSphere Payment Manager**, **Payment Manager Logon**. Logon (the default user ID and password are both admin), create a merchant that is authorized to use LdbCard, and create an LdbCard account for the merchant (the values specified for account parameters won't matter).

12. Make some purchases with the sample store.

13. Return to the Payment Manager user interface and have some fun approving, voiding, depositing, crediting, and settling.

Appendix D: Using the Sample Store

The sample buy page that is provided with the payment manager is unsuitable for testing a cassette. Its servlet contains hard coded assumptions for the PAYMENTTYPE parameter and the protocol data values to be passed on the AcceptPayment verb. Furthermore, the servlet invoked by the sample buy page does not provide any feedback about the result of its actions.

The Sample Store provided with the cookbook materials overcomes these limitations and can be used to generate Payment Manager orders to help you test your cassette. The index.html file included in the Sample Store contains a sample buy page containing an HTML form that uses an HTTP POST request to invoke the Sample Store servlet. In turn, the servlet uses the parameters passed on the POST request to issue an AcceptPayment command to the Payment Manager. See the following table to understand the relationship between the form and the servlet's actions.

Parameter provided by the HTML Form	**Action Taken by the Servlet on its AcceptPayment Command**
PAYMENTTYPE	The value is used as the value for the PAYMENTTYPE keyword.
MERCHANTNUMBER	The value is used as the value for the MERCHANTNUMBER keyword.
AMOUNT	The value is used as the value for the AMOUNT keyword.
CURRENCY	The value is used as the value for the CURRENCY keyword.
ORDERNUMBER	The value is ignored. The servlet automatically generates a value for the ORDERNUMBER parameter by using the date and time. This allows multiple orders to be generated from the same form.
EXPIRYYEAR and EXPIRYMONTH	The four digit year value is combined with the 2 digit month value (YYYYMM) to make the value for the $EXPIRY protocol data keyword.
Any parameter that begins with $	The keyword and its value are passed as cassette dependent protocol data.

Follow these steps to use the Sample Store.

1. Use your favorite text editor to modify D:\!PMCassette\SampleStore\index.html. Change every occurrence of LdbCard to your cassette name. Save the file.

2. If necessary, use your favorite text editor to modify D:\!PMCassette\SampleStore\index.html. For example, if your cassette has additional protocol data values that must be specified by the shopper, add the required input fields to the HTML table inside the HTML form. Use the $PAN parameter as an example. If your cassette has additional protocol data values that, for the purposes of a test, are constants, add the required hidden input fields to the HTML form. Use the PAYMENTTYPE parameter as an example. Save the file.

3. If necessary, use your favorite text editor to modify D:\!PMCassette\SampleStore\SampleStore.java. Typically this is unnecessary. It is only required when protocol data values must be computed instead of passing through. For an example, look how EXPIRYYEAR and EXPIRYMONTH values are combined to make the $EXPIRY parameter value.

4. Recompile the Sample Store cassette using the **Compile** tool in the Visual SlickEdit project, the SampleStore#Build.cmd provided, or with the following commands.

```
D:
CD D:\!PMCassette
SET JDKDIR=C:\jdk1.1.8
```

```
SET WORKDIR=SampleStore
SET ETILLDIR=C:\Program Files\IBM\PaymentManager
SET JAVAXDIR=%ETILLDIR%\IBMWebAS
SET CLASSPATH=%JDKDIR%\lib\classes.zip
SET CLASSPATH=%JAVAXDIR%\lib\jsdk.jar;%CLASSPATH%
SET CLASSPATH=%ETILLDIR%\eTillClasses.zip;%CLASSPATH%
SET CLASSPATH=%WORKDIR%;%CLASSPATH%
ERASE %WORKDIR%\*.class
%JDKDIR%\bin\javac %WORKDIR%\SampleStore.java
```

5. Copy the Sample Store files into the production system using the **Install** tool in the Visual SlickEdit project, the SampleStore#Build.cmd provided, or with the following commands.

```
D:
CD D:\!PMCassette
SET WORKDIR=SampleStore
SET ETILLDIR=C:\Program Files\IBM\PaymentManager
SET HTMLDIR=%ETILLDIR%\httpd\htdocs
SET SERVLETDIR=%etilldir%\IBMWebAS\servlets
xcopy %WORKDIR%\*.class "%SERVLETDIR%"
xcopy %WORKDIR%\*.html  "%HTMLDIR%\SampleStore\*.*"
```

6. Start your favorite browser and go to http://host/SampleStore where *host* is replaced with the host name of the workstation where the Payment Manager is installed. To create an order, fill out the form and press the **Buy** button.

Appendix E: LdbCard Class Descriptions

This appendix describes the classes implemented for the LdbCard cassette.

LdbCardCassette

Class Name	LdbCardCassette
Extends	Cassette
Implements	LdbCardConstants
Responsibilities	This object is responsible for managing the cassette's resources within the Payment Manager framework. The framework uses cassette methods for initialization, creating new objects, resurrecting existing objects from the database, and for servicing API requests that affect administration, payments, and credits.
Construction	During initialization, the framework dynamically loads this class and constructs a single instance when the cassette is referenced by a row in the ETCASSETTECFG database table.
Destruction	Never. The object exists until the Payment Manager is terminated.
Contains	A payment type, a resource bundle identifier, and a Hashtable that contains validation objects for parsing and validating protocol data
References	None
Referenced by	The framework Supervisor

LdbCardConstants

Interface Name LdbCardConstants

Extends PaymentApiConstants

Responsibilities This interface defines any constant that is used in more than one class.

Contains

- The payment type
- The trace identifier
- All cassette dependent database table names
- All cassette dependent database view names
- All column names for the cassette dependent database tables
- All XML data member names
- All protocol data keywords
- All cassette dependent secondary return codes
- All message number constants

LdbCardCassetteQuery

Class Name	LdbCardCassetteQuery
Extends	CassetteQuery
Implements	LdbCardConstants
Responsibilities	This object is responsible for extracting cassette dependent information for Payment Manager query requests. The framework uses the methods for processing query requests that include cassette resources. Inner classes that extend the QueryRequest class are used to construct the SQL statements that obtain the cassette dependent information and process the result set.
Construction	During initialization, the framework dynamically loads this class and constructs a single instance when the cassette is referenced by a row in the ETCASSETTECFG database table.
Destruction	Never. The object exists until the Payment Manager is terminated.
Contains	A payment type and a resource bundle identifier
References	Nothing.
Referenced by	The framework Servlet.

LdbCardAccount

Class Name LdbCardAccount

Extends Object

Implements LdbCardConstants, Archivable

Responsibilities

- This **class** is responsible for managing a collection of account objects that are used by the cassette to process merchant payment requests.
- An **object** is responsible for selecting and retrieving an open batch to contain a payment or credit that uses the account for processing.
- An **object** is responsible for managing the relationship between the merchant and the financial network. It is responsible for making connections to the financial network and exchanging data between the cassette and the payment network. It is also responsible for processing administration requests that start, stop, create, modify, or delete accounts from the configuration.

Construction During initialization, the cassette constructs an account object for each row in the cassette account table. After initialization, the cassette constructs an account object while processing a CREATE_ACCOUNT administration request.

Destruction An account object becomes available for garbage collection after the cassette processes a DELETE_ACCOUNT administration request.

Contains

- A merchant number
- An account number
- Other cassette dependent data that defines the merchant account options (for example, the currency supported)
- Other cassette dependent data that identifies the merchant in the financial network
- Other cassette dependent data used to connect to the financial network

References The associated framework AccountAdmin object

Referenced by

- This class's account collection
- Any order that uses this account to perform payment processing

LdbCardBatch

Class Name LdbCardBatch

Extends Object

Implements CassetteBatch, LdbCardConstants

Responsibilities An **object** is responsible for recording cassette dependent persistent batch data, managing the batch state kept in the framework batch object, and processing the merchant batch requests, namely

- CloseBatch
- DeleteBatch
- PurgeBatch.

Construction An object is constructed when the framework asks the cassette to create or resurrect a batch.

Destruction An object becomes available for garbage collection after the batch is closed or deleted.

Contains Cassette dependent data for managing a batch

References

- The cassette object that owns the batch
- The associated framework batch object
- The cassette account object that will be used to process the batch's payment requests

Referenced by

- The associated framework batch object
- The open batch collection in the associated framework AccountAdmin object

LdbCardOrder

Class Name LdbCardOrder

Extends Object

Implements CassetteOrder, LdbCardConstants

Responsibilities An object is responsible for associating the account object to be used for payment requests, recording all the cassette dependent persistent order data (typically protocol data values passed when the order is created), managing the state of the framework order object, and processing or routing all the requests that affect an order, namely

- ReceivePayment
- AcceptPayment
- Approve
- ApproveReversal
- Deposit
- DepositReversal
- Refund
- RefundReversal
- CloseOrder
- CancelOrder
- CloseBatch

Construction An object is constructed when the framework asks the cassette to create or resurrect an order.

Destruction An object becomes available for garbage collection when the framework removes the associated framework order object from the cache.

Contains Cassette dependent order information (for example, the credit card brand, the credit card number, the expiration date, and any address verification data)

References

- The cassette object that owns the order
- The associated framework order object
- The associated cassette account object that will be used to process the order’s payment requests

Referenced by The associated framework order object

LdbCardPayment

Class Name LdbCardPayment

Extends Object

Implements CassetteTransaction, LdbCardConstants

Responsibilities An object is responsible for recording all the cassette dependent persistent payment data (typically values that identify the payment in the financial network and values that record the outcome of any payment transactions performed), for managing the state of the framework payment object, and for processing all the API requests that affect a payment, namely

- Approve
- ApproveReversal
- Deposit
- DepositReversal.
- CloseBatch
- PurgeBatch

Construction An object is constructed when the framework asks the cassette to create or resurrect a payment.

Destruction An object becomes available for garbage collection when the framework removes the associated framework payment object from the cache.

Contains Cassette dependent payment information (for example, financial network payment identifiers, approval codes, and reason codes)

References

- The cassette object that owns the payment
- The associated framework payment object

Referenced by

- The associated framework payment object
- The transaction key in the payment list container of the framework batch object that contains this payment

LdbCardCredit

Class Name LdbCardCredit

Extends Object

Implements CassetteTransaction, LdbCardConstants

Responsibilities An object is responsible for recording all the cassette dependent persistent credit data (typically values that identify the credit in the financial network and values that record the outcome of any credit transactions performed), for managing the state of the framework credit object, and for processing all the API requests that affect a credit, namely

- Refund
- RefundReversal
- CloseBatch
- PurgeBatch.

Construction An object is constructed when the framework asks the cassette to create or resurrect a credit.

Destruction An object becomes available for garbage collection when the framework removes the associated framework credit object from the cache.

Contains Cassette dependent credit information (for example, financial network credit identifiers, approval codes, and reason codes)

References

1. The cassette object that owns the credit
2. The associated framework credit object

Referenced by

- The associated framework credit object
- The transaction key in the credit list container of the framework batch object that contains this credit

Appendix F: Tips and Techniques

This appendix contains miscellaneous tips and techniques that may help you design and develop your cassette.

How To Add Multiple Brand Administration Objects

1. Make sure you want to do this. This is going to add a great deal of complexity to your cassette. Make sure it is worth it. If the number of brands is a finite short list, consider making a few BRAND columns in the Account table (one group of columns for each brand). If the number of brands varies too much for this approach, consider making an account per brand. Yes, for a given merchant, some information will be duplicated in all the accounts, but that is a small price to pay for the simplicity. Note that, in this respect, both the SET cassette and the CyberCash cassette are bad examples. They consumed programmer years of effort to implement brands when other design points would have been much better for the customers. Continue only if you are absolutely certain that you want to implement brands.

2. Start with the initialization SQL statements. You will need to create a brand table and a view that will select rows from the brand table when the merchant queries an account. Be sure to add the table name, view name, and column names to your Constants interface.

```
CREATE TABLE xxxBrand(MerchantNumber   VARCHAR(9) NOT NULL,
                      AccountNumber    VARCHAR(9) NOT NULL,
                      BrandID          VARCHAR(32) NOT NULL,
                      .
                      . add your cassette dependent columns here
                      .
                      Enabled          SMALLINT,
                      Active           SMALLINT,
                      Valid            SMALLINT,
                      Pending          SMALLINT,
                      MessagesKey      VARCHAR(40),
                      PRIMARY KEY  (MerchantNumber, AccountNumber, BrandID));

CREATE VIEW xxxBrandView AS SELECT
                       ETACCOUNTCFG.*,
                       xxxBrand.BrandID,
                       .
                       . add your cassette dependent columns here
                       .
                       xxxBrand.Enabled as BrandEnabled,
                       xxxBrand.Active as BrandActive,
                       xxxBrand.Valid as BrandValid,
                       xxxBrand.Pending as BrandPending,
                       xxxBrand.MessagesKey as BrandMessagesKey
    FROM xxxBrand, ETACCOUNTCFG WHERE
      xxxBrand.MerchantNumber = ETACCOUNTCFG.MerchantNumber
      and xxxBrand.AccountNumber = ETACCOUNTCFG.AccountNumber;
```

3. Update your PSPL to add a panel that adds, modifies, and deletes brands associated with an account. Be sure to add your XML field IDs to your constants interface.

```
  <screen id="brand" updateID="BRAND" configures="PSMerchantAccount">
    <name>Brands</name>
    <shortHelp>Add, edit, or delete brands for this account.</shortHelp>
    <header><![CDATA[Click on a brand to edit or delete the brand.]]></header>
    <emptyList>No brands exist for this account.  Create a brand by clicking
Add a Brand.</emptyList>
    <fieldGroup id="brand">
      <header>Enter this information as instructed by your financial
institution or certificate authority.</header>

       .
       . Add your cassette dependent field tags here
       .

    </fieldGroup>

    <action id="add">Add a Brand...</action>
    <action id="delete">Delete Selected Brands...</action>
    <action id="create">Create Brand</action>
    <action id="update">Update</action>
    <action id="revert">Revert</action>



    .
    . Add your cassette dependent messages here.  Below are a few examples.
    . You will need to define your own secondary return codes.
    .
    <message id="PRC=5-SRC=11012" type="error">Error: The Brand Name must be
between 1 and 32 characters.</message>
    <message id="PRC=5-SRC=11013" type="error">Error: The Certificate Authority
must be between 1 and 32 characters.</message>
    <message id="PRC=3-SRC=11012" type="error">Error: The Brand Name is
required.</message>
    <message id="PRC=6-SRC=400"   type="error">Error: The given character
encoding is not supported.</message>
    <message id="PRC=2-SRC=11100" type="error">Error: The specified Brand Name
does not exist for this account.</message>
    <message id="PRC=8-SRC=11100" type="error">Error: A brand with the given
Brand Name already exists for this account.</message>
    <message id="PRC=59-SRC=204"  type="error">Error: A brand cannot be deleted
unless all related orders are in Canceled or Closed state.</message>
  </screen>
```

4. Update your CassetteQuery class to obtain and combine all the brand information whenever an AccountQueryRequest is processed. Model the BrandQuery and BrandInfo inner classes after the AccountQuery and AccountInfo inner classes, respectively.

```
      } else if (argRequest instanceof AccountQueryRequest) {
         AccountQuery varQuery = new AccountQuery(
            (AccountQueryRequest)argRequest
         );
         varQuery.combine(argFrameworkObjects);
         BrandQuery varBrand = new BrandQuery(
            (AccountQueryRequest)argRequest
```

```
        );
        varBrand.combine(argFrameworkObjects);
    }
```

5. Update your Cassette class to handle the AdminRequests that create, modify, and delete merchant cassette objects.

6. Update your Cassette class to initialize all the brand objects during the readConfiguration(), verifyConfiguration(), and initialize() methods.

7. Using the LdbCardAccount as a model, implement your cassette's Brand class. Your Brand class should extend AdminObject and implement your constants. Most likely you will need these methods

- Static methods
  - resurrectBrands
  - verifyBrands
  - initializeBrands
  - retreiveBrand
  - retreiveAccountBrands
  - validateCreateRequest
  - validateModifyRequest
  - validateDeleteRequest
- InstanceMethods
  - two constructors: one for creating a brand from protocol data, one for ressurrecting a brand during initialization
  - createRecord
  - updateRecord
  - deleteRecord
  - modifyBrand
  - deleteBrand
  - startBrand
  - stopBrand

8. Update your account so that it cascades functions to all its brands. For example

   - If you start an account, all its brands should be started
   - if you stop an account, all its brands should be stopped
   - if you delete an account, all its brands should be deleted.

How to Create Specialized Parameter Validation Objects

You can create specialized parameter validation objects.

1. Extend the ParameterValidationItem class or any of its existing subclasses.

2. Implement a constructor with

   - A String parameter for the parameter keyword
   - A short parameter for the secondary return code
   - A boolean parameter that indicates whether or not null value are allowed
   - Any other parameter used by your class to validate the parameter value

3. In your constructor, call the appropriate constructor for your super class.

4. Implement a validateAndInsertValue() method with

A byte[] parameter that contains the value
A ParameterTable parameter where the result will be placed
A throws ETillAbortOperation clause.

5. In your validateAndInsertValue() method perform the required conversions and validity checks. Throw an ETillAbortOperation exception with the appropriate primary return code and the secondary return code passed in the constructor if there is any conversion error or validity check failure. Otherwise, save the converted value in the supplied ParameterTable using the parameter keyword passed on the constructor as the key.

The cookbook materials contain an example that performs the Luhn Check on credit card numbers. Look in D:\!PMCassette\com\ibm\ldbtestcassette\LuhnStringValidationItem.java.

Appendix G: WebSphere Commerce Suite V4.1

The IBM WebSphere Commerce Suite Version 4.1 is a descendent of and replacement for IBM Net.Commerce. The WebSphere Payment Manager is included with every copy of the WebSphere Commerce Suite. Every WebSphere Commerce Suite customer is potentially a customer for your cassette.

The WebSphere Commerce Suite is extraordinarily flexible and customizable. There is no doubt that, given enough effort, any merchant can customize a store to use your cassette to its full capability while providing the shopper with a delightful shopping experience. But most merchants cannot afford the effort necessary develop the best possible implementation. Typically, they rely on default conventions and default programming provided by IBM. When customization is necessary, they rely on sample source that can be easily modified to perform correctly in their environment.

As a cassette developer, you are faced with several choices.

- You can ignore the WebSphere Commerce Suite. However, this will limit your customers to those that can afford to design and develop customized software that exploits your cassette and its financial network.

- You can study the WebSphere Commerce Suite just enough to assure that your cassette will work with the default programming provided by IBM. However, this will probably reduce the usefulness of your cassette to the smallest common subset of functions supported by all payment cassettes. The advanced features of your cassette, and therefore your competitive edge, will be lost.

- You can invest in extra effort and provide the WebSphere Commerce Suite merchants with instructions and software that can be used to customize the merchant's store. However, this option comes with its own challenge: you must sustain the cost of the additional development and testing while providing a deployment package that does not overwhelm the merchant.

THE FOLLOWING SECTIONS IN THIS APPENDIX WILL

- Help you understand the system behavior of the default programming provided by IBM
- Help you understand how you can provide software that the merchant can use to extend the system beyond the default programming
- Describe IBM's vision for a better system where cassette developers do not have to provide special merchant software for the WebSphere Commerce Suite or any other merchant business system.

The Default Behavior for Cassettes That Support a Wallet

To trigger this behavior, the merchant must customize the macro assigned to the ORD_DSP_PEN view task. Roughly, ORD_DSP_PEN translates to "display the pending order". This task is invoked when the shopper presses a button indicating the desire to checkout. By default, the macro displays the order in its final form and gives the shopper an opportunity to select a payment method.

The customization provided by the merchant gives the shopper a way to indicate that they wish to pay for the order using their wallet. (The wallet is a browser plug-in that will wake up after your cassette supplies the wake-up message in response to a ReceivePayment command.) This web page will be coded so that when the shopper makes this choice

- An HTTP form will be posted to the WebSphere Commerce Suite OrderProcess command or the first command in a chain of commands that eventually posts the OrderProcess command
- The form will contain a hidden parameter named do_payment_type where do_payment_type=INIT
- The form will contain a hidden parameter named payment_type where the value is set to your cassette name
- The form will contain a hidden parameter named merchant_rn where the value matches the merchant's merchant number
- The form will contain a hidden parameter named order_rn where the value matches the shopper's order number.

When the post request reaches the OrderProcess command, the following events occur.

- The OrderProcess command reaches a point where it decides how to collect payment information. In this case, do_payment_type is INIT so it invokes the overridable function that has been assigned to the DO_PAYINIT process task. By default, the RecvPaymentPM function provided by IBM is assigned to this task.

- RecvPaymentPM invokes the overridable function assigned to the PAY_PROTOCOL_DATA process task. By default, this task is assigned to a function that does nothing.

- RecvPaymentPM invokes the overridable function assigned to the PAY_ORDER_DESC task. By default, this task is assigned to the OrderDesc function provided by IBM. It generates a string that describes the order. Here is an example.

```
K.J. Holdings Inc.
Order #30164
SKU#       Description          Quantity  Price/Unit       Price
-------------------------------------------------------------------
91-123H    Claw Hammer                 1       15.89       15.89
91-648B    Ballpein Hammer             1       13.99       13.99
-------------------------------------------------------------------
                                          Sub-Total       29.88
                                                Tax        4.48
                                           Shipping        5.00
                                       Shipping Tax        0.00
                                                      ------------
                                        Grand Total       39.36
```

- RecvPaymentPM sends a ReceivePayment command to the WebSphere Payment Manager where

  - AMOUNT is set to the order's amount
  - CURRENCY is set to the store's currency
  - MERCHANTNUMBER is set to the value of the merchant-rn
  - ORDERNUMBER is set to the value of the order_rn
  - PAYMENTTYPE is set to the value of payment_type
  - AUTOAPPROVE is set to the value that is found in the AutoApprove column of the PAYOPTIONS table where the merchant number and payment type is used as an index to find the row
  - When AutoApprove=1, PAYMENTAMOUNT is set to the same value as AMOUNT
  - When AutoApprove=1, PAYMENTNUMBER is set to 1

- When AutoApprove=1, AUTODEPOSIT is set to the value found in the AutoDeposit column of the PAYOPTIONS table using the same row that found the AutoApprove configuration value.
- Additional protocol data keywords and values are taken from a name/value pair table, empty by default but potentially updated by the overrideable function assigned to the PAY_PROTOCOL_DATA task.

- When ReceivePayment completes successfully, RecvPaymentPM extracts the wake-up message and returns it to the shopper's browser. Payment protocols are then exchanged between the shopper's wallet and your cassette. When the ReceivePayment fails, RecvPaymentPM sets the error handler to the DO_PAYMENT_ERR exception task and returns false. This error task does nothing but inform the shopper that an error occurred in the most general terms. The primary and secondary return codes from the Payment Manager are **not** used to provide any detailed information about the cause even when the failure was caused by bad shopper input.

- Meanwhile, a background server thread is watching the state of the Payment Manager orders. When an order state changes from "Requested" to "Ordered" or "Refundable", the UPDATE_INV task is used to commit the inventory to the shopper's order. In unsuccessful, the WebSphere Commerce Suite order is set to "Insufficient Inventory" and merchant intervention is required to resolve the problem.

- Meanwhile, a background server thread is watching the state of the Payment Manager payments. When a payment state changes to "Approved" and the WebSphere Commerce Suite order is not "Insufficient Inventory", the WebSphere Commerce Suite order is set to "Complete" (in other words, ready to ship).

The Default Behavior for Cassettes That Support Explicit Shopper Input

To trigger this behavior, the merchant must customize the macro assigned to the ORD_DSP_PEN view task. Roughly, ORD_DSP_PEN translates to "display the pending order". This task is invoked when the shopper presses a button indicating the desire to checkout. By default, the macro displays the order in its final form and gives the shopper an opportunity to select a payment method.

The customization provided by the merchant gives the shopper a way to indicate that they wish to pay for the order by manually entering payment information. The shopper enters payment information into an HTML form and, eventually, the values entered are sent to the Payment Manager using an AcceptPayment command. This web page will be coded so that when the shopper makes this choice

- An HTTP form will be posted to the WebSphere Commerce Suite OrderProcess command or the first command in a chain of commands that eventually posts the OrderProcess command
- The form will **not** contain a hidden parameter named do_payment_type or, optionally, will contain a hidden parameter named do_payment_type where the value is unspecified (in other words, the value is null).
- The form will contain a hidden parameter named do_payment_type_suffix where do_payment_suffix=PMACCEPT.
- The form will contain a hidden parameter named payment_type where the value is set to your cassette name
- The form will contain a hidden parameter named merchant_rn where the value matches the merchant's merchant number
- The form will contain a hidden parameter named order_rn where the value matches the shoppers order number
- The form will contain a parameter named cctype where the value is the brand name of the supported payment instrument chosen by the shopper (for example, Visa or Discover)

- The form will contain a parameter named ccnum where the value is a personal account number, entered by the shopper, for the payment instrument chosen
- The form will contain a parameter named ccxyear where the value is a year, selected by the shopper, when the payment instrument will expire
- The form will contain a parameter named ccxmonth where the value is a month, selected by the shopper, when the payment instrument will expire.

When the post request reaches the OrderProcess command, the following events occur.

- The OrderProcess command reaches a point where it decides how to collect payment information. In this case, do_payment_type is null so it examines the do_payment_suffix. It builds a new task name by appending an underscore and the value of the do_payment_suffix keyword to the end of DO_PAYMENT. In this case, the resulting task name is DO_PAYMENT_PMACCEPT. It then invokes the overridable function that has been assigned to the process task with the task name just constructed. By default, the DoPaymentPMAccept function provided by IBM is assigned to the DO_PAYMENT_PMACCEPT process task.

- DoPaymentPMAccept performs validation checks on the ccnum value.

  - The value specified for cctype is used to select the appropriate rows in the CCCHECK table.
  - For each row found, the value of the Prefix column is compared to the prefix of the value specified for ccnum. If a match is not found, an exception is thrown and the shopper is informed that the specified value is invalid. If a match is found, the row is used for the next checks.
  - The length of the ccnum value is compared with the Length column of the selected row. If there is no match, an exception is thrown and the shopper is informed that the specified value is invalid. If a match is found, the row is used for the next check.
  - A task name is obtained from the CheckAlgorithm column of the selected row. That task is invoked to complete the ccnum checks. If this task discovers a problem, an exception is thrown and the shopper is informed that the specified value is invalid. Otherwise, control is returned to DoPaymentPmAccept.

  Typically, the CheckAlgorithm column contains the value DO_LUHN_CHECK. By default, the DoLuhnCheck function provided by IBM is assigned to this task.

- DoPaymentPMAccept performs validation checks on the ccxyear and ccxmonth values. If the indicated expiration date has passed, an exception is thrown and the shopper is informed that the instrument has expired. If the indicated expiration date is 10 years or more in the future, an exception is thrown and the shopper is informed that the expiration date is invalid.

- DoPaymentPMAccept invokes the overridable function assigned to the PAY_PROTOCOL_DATA process task. By default, this task is assigned to a function that does nothing.

- DoPaymentPMAccept invokes the overridable function assigned to the PAY_ORDER_DESC task. By default, this task is assigned to the OrderDesc function provided by IBM. It generates a string that describes the order. Here is an example.

```
K.J. Holdings Inc.
Order #30164
SKU#       Description          Quantity  Price/Unit       Price
-------------------------------------------------------------------
91-123H    Claw Hammer                 1       15.89       15.89
91-648B    Ballpein Hammer             1       13.99       13.99
```

```
------------------------------------------------------------
                                        Sub-Total       29.88
                                              Tax        4.48
                                         Shipping        5.00
                                     Shipping Tax        0.00
                                                  ------------
                                      Grand Total       39.36
```

- DoPaymentPMAccept sends an AcceptPayment command to the WebSphere Payment Manager where

  - AMOUNT is set to the order's amount
  - CURRENCY is set to the store's currency
  - MERCHANTNUMBER is set to the value of the merchant-rn
  - ORDERNUMBER is set to the value of the order_rn
  - PAYMENTTYPE is set to the value of payment_type
  - AUTOAPPROVE is always set to 0
  - $BRAND is set to the value of cctype
  - $PAN is set to the value of ccnum
  - $EXPIRY is set to the value of ccxyear concatenated with the value of ccxmonth
  - Additional protocol data keywords and values are taken from a name/value pair table, empty by default but potentially updated by the overrideable function assigned to the PAY_PROTOCOL_DATA task.

  Note that AUTOAPPROVE is always set to 0 here, even if the merchant has requested an automatic approval. DoPaymentPMAccept runs on a critical thread that cannot afford to be suspended while the Payment Manager is exchanging protocols to authorize the payment. A background scheduler will be used to issue an APPROVE command on a different thread.

- When AcceptPayment completes successfully, DoPaymentPMAccept examines the PAYOPTIONS table using the merchant number and the payment type to select a row. If the AutoApprove column indicates that an automatic approval is desired. DoPaymentPMAccept schedules a work item to be performed by the background scheduler. When the AcceptPayment fails, DoPaymentPMAccept sets the error handler to the DO_PAYMENT_ERR exception task and returns false. This error task does nothing but inform the shopper that an error occurred in the most general terms. The primary and secondary return codes from the Payment Manager are **not** used to provide any detailed information about the cause even when the failure was caused by bad shopper input.

- Meanwhile, the background scheduler is watching its work queue. When a work item arrives. The background scheduler sends an Approve command to the WebSphere Payment Manager where

  - AMOUNT is set to the order's amount
  - MERCHANTNUMBER is set to the value of the merchant-rn
  - ORDERNUMBER is set to the value of the order_rn
  - PAYMENTNUMBER is set to 1
  - AUTODEPOSIT is set to the value that is found in the AutoDeposit column of the PAYOPTIONS table where the merchant number and the payment type is used to find the row.

- Meanwhile, a background server thread is watching the state of the Payment Manager payments. When a payment state changes to "Approved", the WebSphere Commerce Suite order is set to "Complete" (in other words, ready to ship).

Planning Questions and Answers

The following questions and answers should help you make plans for the effort required to support the WebSphere Commerce Suite.

Q: What if I do nothing?

A: Software engineers call this "exporting the cost to the customer". In this case, the merchant will need to justify the effort to do the research, design, and development necessary to customize their installation to utilize your cassette. If your financial system is attractive enough and valuable enough, merchants may find the effort justified. Otherwise, they will choose a competitor's cassette which requires less effort to integrate.

But watch out! Consider what will happen if you take this approach and you are outrageously successful. It is very likely that merchants will have questions and problems with their integration and will over load your customer support system.

Q: What is the lowest reasonable investment I can make and still support WebSphere Commerce Suite with my cassette?

A: These are the steps for the lowest reasonable investment.

1. Design your cassette so that it will operate correctly given the default behavior of the WebSphere Commerce Suite. You will need to examine the details provided earlier in this appendix but, for example,

   - If you support ReceivePayment, don't require any protocol data
   - If you support AcceptPayment, require only $PAN, $BRAND, $EXPIRY and define their values in a way that is consistent with the automatic checks that will be performed before AcceptPayment is issued.

2. Expand your test effort to include

   - The installation and configuration of WebSphere Commerce Suite
   - The customization necessary to use your cassette
   - Enough system level testing to assure that everything interoperates correctly
   - There are two vastly different paths to the OrderProcess command, so be sure to test in both types of stores – a mall and a one-stop-shop.

3. Expand your documentation to show a WebSphere Commerce Suite merchant how to customize their system so that your cassette is used when the shopper chooses the proper brand or payment instrument. In this case, the documentation is very simple.

   - When a wallet is used, instruct the merchant to include do_payment_type=INIT as a hidden variable when your cassette is chosen. When a wallet is not used instruct the merchant to include do_payment_suffix=PMACCEPT as a hidden variable when your cassette is chosen.
   - In either case, instruct the merchant to include payment_type=X (where X is replaced with your cassette name) when your cassette is chosen.

- Include several samples. At least two examples are required: one for when the store is part of a mall and another for when the store is a one-stop-shop.

Q: That's almost good enough, but the account numbers used by my cassette will not pass the automatic checks. What can I do?

A: First consider adding information to the CCCHECK table to support your account numbers. To find an example, search the WebSphere Commerce Suite installation directory tree for a file named cccheck_add.v32.sql. This SQL script is used to define the default table.

Note that the CCCHECK table will require a row for every Type/Prefix/Length combination that you support. Consider a case where a type can have any possible prefix and can vary between 5 and 15 digits. In this case, there will be 10 (each digit 0 through 9) times 11 (11 different lengths) or 110 rows defined for that type. Hopefully, your case won't be that bad.

For each row in the CCHECK table, you can specify a task to be used to validate that the digits have been entered correctly. If your account numbers can be verified with the Luhn Check, specify the 'DO_LUHN_CHECK' task. If there is a different algorithm, you can specify a task name for an overridable function that you develop and supply. (At this time, there is no example available.) If there is no algorithm for verifying the account numbers, leave the column null.

IF THIS STRATEGY IS ACCEPTABLE, ADD THE FOLLOWING STEPS TO YOUR DEVELOPMENT EFFORT.

1. Do all the lowest reasonable investment steps documented above.

2. Design and develop a SQL script that can be used to add the required rows to the CCCHECK table.

3. If necessary, design and develop an overridable function that will validate that the account number digits have been entered correctly. Then design and develop a SQL script that will add a task name for your overridable function and assign your overridable function to that task.

4. Update your cassette installation facility to detect the presence of the WebSphere Commerce Suite. If detected and approved by the administrator, have the facility add your rows to the CCCHECK table and, if necessary, copy your DLL into the system and configure the system for your new task/overridable function.

5. Expand your documentation to describe WebSphere Commerce Suite extensions that you supply

If this strategy is not acceptable, continue with the following question.

Q: The shopper input for my cassette just doesn't match the cctype/ccnum/ccxyear/ccxmonth paradigm used by the default behavior. What can I do?

A: Add the following steps to your development effort and then continue to the next question. These steps demonstrate a way to bypass the default checks and the next question shows how to get a different type of shopper input into your cassette.

1. Design and develop a SQL script that can be used to add one row to the CCCHECK table. . To find an example, search the WebSphere Commerce Suite installation directory tree for a file named cccheck_add.v32.sql. In this row,

   - Set the CCTYPE column to your cassette name

- Set the CCLENGTH column to 5
- Set the CCPREFIX column to 9
- Leave the CCALGTASKRN column null (in other words, empty)

2. Enhance your documentation and your examples to show the merchant how to add hidden parameters that will always pass the automatic checks performed.

   - Use cctype="X" where X is the name of your cassette.
   - Use ccnum="99999".
   - Use ccxyear="2009".
   - Use ccxmonth="01".

   Note that this will cause an interesting defect if the code is still in production after January 2009. You may opt for a more robust approach. For example, consider using JavaScript to set ccxyear to one year in the future and ccxmonth to the current month.

3. Add support for your cassette dependent shopper input using the steps outlined below.

Q: I am not satisfied with WebSphere Commerce Suite's default behavior. I don't want my cassette to be reduced to the lowest functional level provided by all cassettes. I want every competitive advantage. To exploit the advanced features of my cassette, it needs additional data that is supplied by the shopper. For example,

- **Cardholder name and billing address for an address verification system**
- **A cardholder verification code like VISA's card verification value 2 (CCV2) or Mastercard's Card Verification Code 2 (CVC2)**
- **A shopper account number that does not match the credit card paradigm.**

What can I do?

A: These are the steps to customize protocol data.

1. If cctype/ccnum/ccxyear/ccxmonth parameters used by the default behavior do not apply to your cassette, start with the steps outlined above.

2. Expand your development effort to include the design and implementation of an overridable function for the PAY_PROTOCOL_DATA task. Good examples can be found in the adt\samples\payment\protdata directory that was placed in the WebSphere Commerce Suit installation directory. In this overridable function, you will include the logic to find the values specified by the shopper, manipulate the values if necessary, and save the appropriate protocol data keyword and value in the name/value pair collection provided by the caller.

3. Expand your test effort to include

   - The installation and configuration of WebSphere Commerce Suite
   - The customization necessary to use your cassette
   - Enough system level testing to assure that everything interoperates correctly
   - There are two vastly different paths to the OrderProcess command, so be sure to test in both types of stores – a mall and a one-stop-shop.

4. Update your cassette installation facility to detect the presence of the WebSphere Commerce Suite. If detected and approved by the administrator, have the facility add your overridable functions and any related tasks to the system.

5. Expand your documentation to show a WebSphere Commerce Suite merchant how to customize their system so that your cassette is used when the shopper chooses the proper brand or payment instrument.

   - When a wallet is used, instruct the merchant to include do_payment_type=INIT as a hidden variable when your cassette is chosen. When a wallet is not used instruct the merchant to include do_payment_suffix=PMACCEPT as a hidden variable when your cassette is chosen.
   - In either case, instruct the merchant to include payment_type=X (where X is replaced with your cassette name) when your cassette is chosen.
   - Tell the merchant how to add the correct HTML form elements necessary to get the additional information from the shopper. Be sure to explicitly define the parameter names and the values allowed, including any length restrictions.
   - Include several samples. At least two examples are required: one for when the store is part of a mall and another for when the store is a one-stop-shop.

6. Expand your documentation to show a WebSphere Commerce Suite merchant how to customize their system when it requires more than one overridable function for the PAY_PROTOCOL_DATA task. WebSphere only allows one overridable function to be assigned to a task per merchant. If the merchant wants to use more than one cassette and several cassettes require overridable functions for the PAY_PROTOCOL_DATA task, the merchant must perform extra customization.

   - The merchant must assign a payment type task switch to the existing PAY_PROTOCOL_DATA task. An example is provided with the cookbook. Look in the D:\!PMCassette\WebSphereCommerce\Suit directory. The example switches to the correct task by appending the value of the payment_type parameter to the current task name.

   - The merchant must define a new task for every cassette used, even if the cassette does not require an overridable function. Each task name must be of the form PAY_PROTOCOL_DATA_X where X is the cassette name.

   - For each task created above, the merchant must assign an overridable function. If the cassette does not require an overridable function, the DoNothingNoArgs overridable function supplied by IBM can be used. For cassettes that required a specific overridable function, the function supplied with the cassette must be used.

Q: To exploit the advanced features of my cassette, it needs additional data that is supplied, not by the shopper, but by the merchant system. For example,

- **A shopper may be annoyed if asked to specify cardholder billing address information when the information is already known by the system (probably through shopper registration)**
- **Line item detail is required for advanced fraud detection and billing.**

What can I do?

A: USE THE STEPS TO CUSTOMIZE PROTOCOL DATA DOCUMENTED ABOVE, BUT ADD EXTRA EFFORT TO THE DESIGN, DEVELOPMENT, AND DOCUMENTATION OF YOUR OVERRIDABLE FUNCTION. DO YOUR BEST NOT TO UNDERESTIMATE THIS EXTRA EFFORT. IT IS MUCH MORE DIFFICULT THAN OBTAINING EXTRA INFORMATION FROM THE SHOPPER.

The WebSphere Commerce System can contain many bits of information that may be valuable to a financial network. For example,

- Line item detail for the order
- The merchant's store name
- The merchant's address
- The merchant's contact information
- Shipping address
- Shopper billing address
- Shopper e-mail address
- Shopper telephone number.

However, a given merchant system may not collect all the possible bits of information. To encourage merchants to fully utilize your cassette, you may need to provide detailed documentation explaining the value of collecting the information and suggesting techniques for collecting it.

Worse, the techniques for extracting data from the system are not well documented and there aren't very many examples. You may need significant investment to find where the data is kept and what technique is used to obtain it.

There is a good example for obtaining line item detail. The examples for the PAY_ORDER_DESC task use a good technique for getting line item detail in order to build the order description. Look in the adt\samples\payment\orderdesc directory that was placed in the WebSphere Commerce Suite installation directory.

Q: Following the guidelines in this cookbook, my cassette will validate any data provided by the shopper and return primary and secondary return codes that precisely identify the cause of any failures. However, the WebSphere Commerce Suite does not give this information to the shopper. Shoppers and merchants will be annoyed if the shopper gets some obscure message reporting that "The payment system has failed. Please try again or contact the merchant." whenever the shopper enters an invalid value. I want to facilitate a delightful shopper experience. I want the shopper to see a message that precisely identifies the parameter and states why the value is invalid.

What can I do?

A: Use the steps to customize protocol data documented above, but add extra effort to the design, development, and documentation of your overridable function.

1. Define a new error task name that will be used whenever you need to inform the shopper that a supplied value was entered incorrectly. You may be tempted to reuse the existing DO_PAYMENT_ERR task, but that may cause collisions when the merchant uses more than one cassette that wants to override this task.

2. Design and implement a cassette dependent error macro that will be assigned to your new error task. Examples can be found in the adt\samples\macro directory that was placed in the WebSphere Commerce Suite installation directory. Error information can be extracted from the environment so that this macro will generate the correct message for the shopper. Keep in mind that this macro may need to support national languages other than English and may need to be customized by the merchant.

3. Enhance your overridable function to set the error handler task to your new error exception task.

4. Enhance your overridable function to validate any data that could potentially be provided by the shopper. True, this duplicates the effort that will be performed by the cassette, but it is unavoidable

given the current state of the art. If a validation error is detected, save the appropriate error information for your error task and return a boolean false.

5. Enhance your documentation to inform the merchant about the validation checks, how the error macro is used, and how to customize the error macro if necessary.

6. Enhance your cassette installation facility to add the new task and its default macro to the system.

A Vision for a Better System

The statements in this section cannot be construed as a commitment by IBM to supply the software or product enhancements discussed. Rather it is an acknowledgment that integration with WebSphere Commerce Suite can place an excessive burden on cassette development and there is a requirement to reduce the integration effort. IBM's ability to meet these requirements will depend on market and business considerations. Furthermore, advancing technology may make this vision obsolete.

In the near term, the vision is to enhance the cookbook materials with better examples for overridable functions. Better, the hope is to establish conventions and techniques that allow the development of overridable functions that can be used by most cassettes without modification. Use the cassette developers forum to request examples and comment on beneficial conventions.

In the long term, the vision is to remove the need for cassette dependent customization of merchant systems by providing a profiling system. With this system, the cassette developer would provide an XML document (perhaps using Payment Server Presentation Language (PSPL) tags) that describes the protocol data keywords supported by the cassette on ReceivePayment and AcceptPayment commands. Just as for the Payment Manager user interface today, the document will show how to translate Payment Manager return codes in to the correct operator message.

During the installation phase, a commerce system administrator would use this XML document to generate a profile. The completed profile defines which protocol data keywords will be used and how the values will be supplied. Values could be supplied

- As a constant
- As a value from a given parameter from the HTTP form submitted by the shopper
- As a value from a given column of a supplied database query.

IBM would supply software that would perform payment processing. It would examine the profile, automatically marshal the parameters as directed by the profile, send the correct command to the Payment Manager, and, if necessary, display the correct error message to the shopper.

Although the merchant may still need to modify the catalog system to obtain extra information from the shopper, no other merchant system software would need to be implemented by either the merchant or the cassette developer.