jBilling

Extension Guide

Community Edition

Published: January 19, 2017

Copyright

Table of Contents

Chapter 1
Architecture

An overview of the jBilling design

CHAPTER 1: Architecture

It is easy to make a complex design to solve a complex problem. The challenge is to produce a simple design that would address complex problems, like we often face for billing requirements. This document will show that the design behind jBilling is simple, and yet the result is an enterprise billing system. Anybody with knowledge of Java should be able to read and understand this document in less than half an hour. After that, you can start modifying and extending jBilling. To quote Martin Fowler *"I like to structure documentation as prose documents, short enough to read over a cup of coffee, using UML diagrams to help illustrate the discussion"*. Coffee is not good for you, so we won't go along with that. Try chocolate milk instead.

The remaining chapters go into the details of each extension point or major module. You will not need to read them all in every case. It depends on what is that you are planning to do. This document is intended to be a reference that goes into more details about the key internals of the system. After that, if you have any questions, please contact us for additional information.

Before you start, you should have a basic understanding of how jBilling works from a user perspective. At the very least, read and follow the 'Getting Started' guide (in the documentation section). Experiment and get to know well the specific part of the system that you will be changing. For example, if you are going to make changes to payments, create a new payment and see how it affects an invoice.

Overview

The most significant characteristic of jBilling's architecture is the 3 tier layout:

- **Client**: Deals with all the interaction with the user (user interface). It only communicates with the server tier, never directly to the database tier. It does not have any business logic. The communication with server tier is done through APIs provided by server. Client calls serve APIs according to the requirement of the task and then the server acts accordingly by processing business logic, interacting with the database, or both as may be required by the request.
- **Server**: It is the holder of the business logic and the only tier that talks with the other two other tiers: client and database.
- **Database**: An RDBMS engine that holds all the data. It only holds data. There are no stored procedures or any other kind of code, let alone business logic here. Only the server tier has access to the database.

Standard names are used to describe these tiers, but to avoid confusion, here is additional information. The client is not the browser running on the user's PC, but the web server dealing with it. The server is actually an application server capable of serving Java servlets (Tomcat, Jetty, etc). These terms are more logical than physical. In practice, you could have one server per tier, or you can have them all in one box; you could even have a cluster of servers for any of the tiers. Also, the components responsible for how the data is accessed are in the database tier, but they are deployed in the application server.

Client Tier

The main factor of this tier is its implementation in Grails. This is an implementation of the model-view-controller design pattern for web-based applications written in Groovy. Users access the system using a web browser, which connects to a web server where the GSPs are deployed. All the requests are handled by the Grails controller, then forwarded to Actions that eventually call the server to get something done. Grails also help with internationalization (i18n), validation, and page layout (views), among other things.

Grails provide a lot of benefits. Grails, built on Groovy, give the immediate benefit of being more productive. The Groovy syntax is much terser than Java. It's much easier to do things in one line of Groovy code than the equivalent several lines in Java. Grails is built on top of SpringMVC, and you can integrate other components using Spring. Some of the advantages of Grails include Database Migrations and Versioning (no more application out of sync with database schema syndrome), Artefacts (which make creating new controllers and components easier), Scaffolding (provides you with all some initial components to allow you to start building your pages and customising), Simpler validation, Built in Webflow (via Spring webflow, which makes creating complex workflows much simpler), Better data binding and a lot more.

The communication between client tier and server tier is done through API calls. These API calls are provided by server tier. When a client has an action/request to process, a read/write data to the database, or a related job to do, the client calls the server APIs according to its requirement. The server tier then process the request, either by processing business logic or database interaction, or both if required. Then the server returns the call to the client with or without result (return value) depending on the type of API called.

The tiered approach has several benefits. The first advantage is that if a customer wants to have the jBilling billing solution without the UI part, because the customer has their own, or wants to develop their own according to customized requirements, this approach ensures the flexibility for jBilling to do that. All customer has to do is take the jBilling billing solution without the UI, i.e. client tier layer. They can build on their existing client tier (UI) and integrate it with jBilling's server tier, and it is good to go. The application is soon up and running, and ready to use. This is an important feature that provides a lot of flexibility for the customers. A second advantage is that tiers allow access to the application by a customer, without actually providing all the code. APIs can be used, providing the customer with just the required parameters and the return types, so that customer can access those APIs without having actual code.

Server Tier

The server tier is where all the work gets done. All the business logic is in this tier. It receives requests from the user interface or from web services, and responds to them by running some business logic code and interacting with the database.

jBilling is a Java EE application that relies on the Spring Framework for enterprise services such as demarcations of transactional boundaries, integration with Hibernate, JMS, etc. All it needs is a Java web server to run. It would be possible to run it outside any application server, as a simple Java application as well (no GUI or web services).

It is worth noting that jBilling started as an EJB application that ran only on JBoss. Session beans, entity beans, etc were all over the code. None of this is now the case. jBilling is today a Spring application, but this start in the EJB world has left 'scars' in the code. The key frameworks in use are Spring and Hibernate, and the standard deployment will use Tomcat and ActiveMQ.

Database Tier

The first rule to keep in mind regarding the database is that we want to stay vendor independent. This means no specific functions or extensions of SQL that are specific to a database engine. Initially, jBilling ran on PostgreSQL, which is powerful and open source. It is easy to run jBilling in any other engine that supports ANSI SQL.

There are a few ways to access the database: through Hibernate persistent classes, Hibernate HQL queries, Hibernate criteria queries, and through JDBC calls. You will find a lot of JDBC queries in the code. These are read only queries—no modifications are done this way. Direct JDBC is one of those 'scars' that EJB left on jBilling: Entity beans produce locks and are slow, and SQL was added to overcome this problem.

The preferred query method is Hibernate criteria, but only when the query is not very complex. For complex queries, HQL is recommended.

Meta Fields

Meta fields were introduced first in jBilling 3.1.0 Community Edition. jBilling gives you valuable flexibility by providing you with the ability to create fields, called meta fields, that can be added to forms in your billing system. Initially available in just five areas, they are now available in most areas, including:

- Account Type
- Agent
- Asset
- Customer
- Invoice
- Order
- Order Change
- Order Line
- Payment
- Payment Method Template
- Payment Method Type
- Plan

- Product
- Product Category

Creating a meta field is easy. For the plug-in example that follows you will need to create the following meta field:

- Click on the configuration tab, and then in the sidebar on the left click on the meta field option. This should bring up the available entity types to choose from.
- Choose the entity type; Product. Click on the “+ Add New” button on the right to take you to the meta field form.
- The first field, “Number,” is the meta fieldID# and it will not be assigned until after the meta field is created (Save Changes).
- Fill out the following fields; Name (call it Discount),, Data type (integer) and Display Order (1). Leave Default Value blank.
- Ensure the check boxes Mandatory and Disabled are not selected and click Save Changes.
- After the meta field is created the system will generate an ID #. You will need this ID # for the plug-in example. Please make a note of it.

If you like, you can go back to the product creation page to see that your newly created field is now part of the product creation form.

The Plug-in System

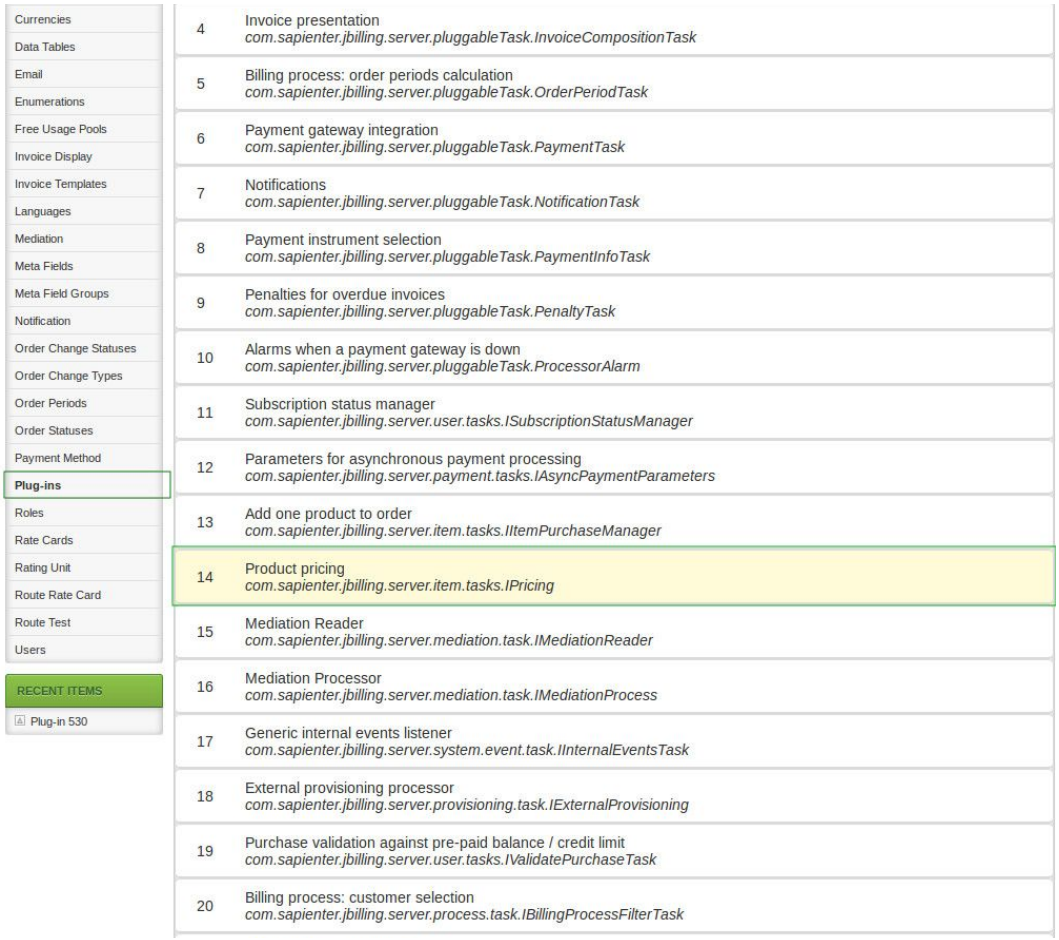
To enhance your billing system flexibility and extensibility jBilling has implemented a system of plug-ins, based on Strategy and Chain of Responsibility designs. This allows you to create the steps needed, using the plug-ins provided or plug-ins you create yourself, that your business rules require, and excluding those that do not apply. For example, you might normally add all products ordered to your invoice but then have a series of taxes such as federal, state, local and maybe a VAT (Value Added Tax) at the end. You would simply select and configure the plug-ins needed to create the tax rules and jBilling takes care of the rest. Let’s walk through a particular example to show you how simple business rules can be established, managed, and maintained.

The example we will use is a basic product discount rule. You will create and maintain a category of products you wish to sell at a discount. You should have already created a customer, as well as a product and category. You can now add a new category for discount products, modify it, and apply your discount rule by implementing a supplied plug-in.

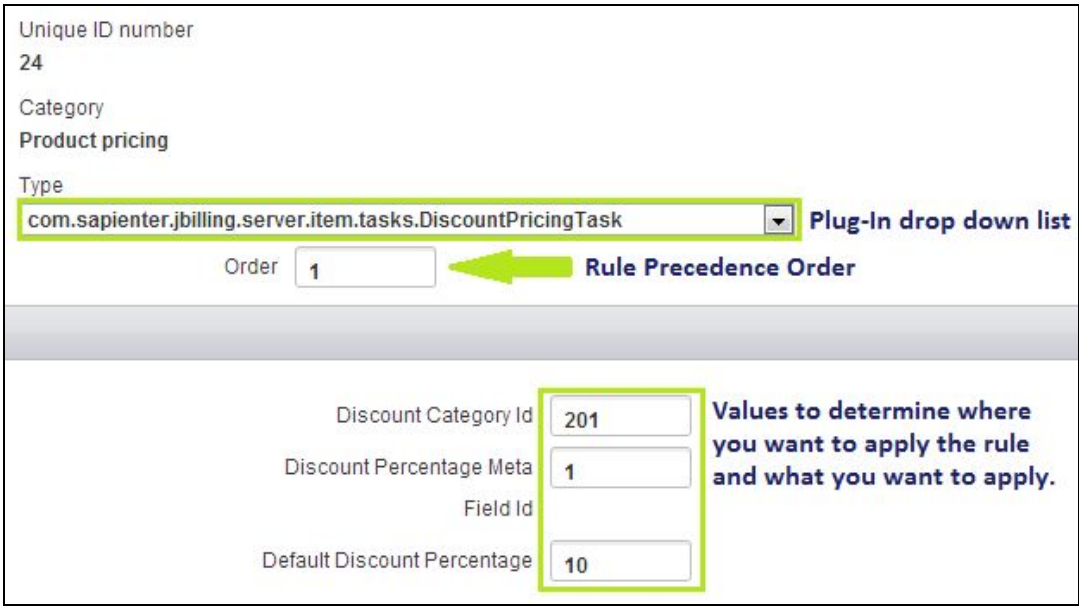
1. Create a new category of products called “Discount Products”. Within that category create a product called “Discount Banners”. You will see your new Discount Meta Field (created above); for now leave the meta field “Discount” blank. Keep track of the category ID # as you will be needing it soon.

2. Click on the configuration tab at the top of the screen and then click on plug-ins on the left sidebar. This brings up a full list of all installed plug-ins. Changing any of the pre-installed plug-ins without a full understanding of what they do may result in unwanted results so please be careful.

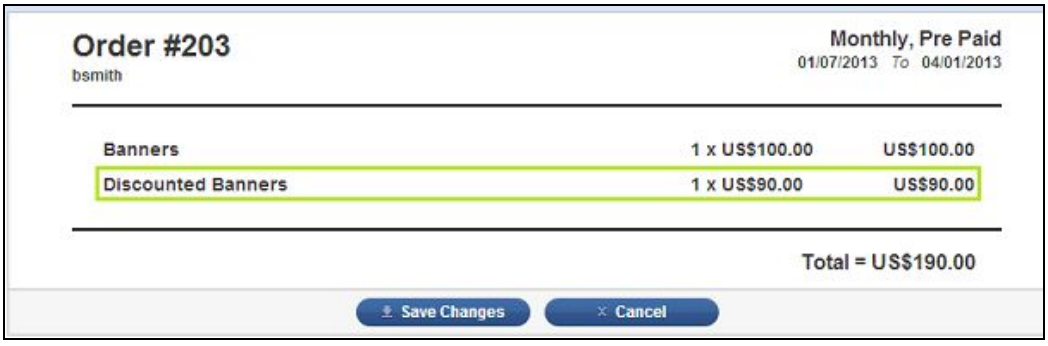
3. Take a moment to click on at least one of these plug-in categories. You will see some pre-installed plug-ins displayed to the right. Click on any of these displayed plug-ins to bring up a detailed description of what that plug-in does.
4. In the list of plug-ins categories you will see a category called “Product Pricing”. Click on it to highlight this category. There is no default plug-in implemented yet but we will install one now by clicking the “+ Add New Button” that will be on the right. This should take you to the plug-in page.



5. You will see a drop down list with several similar sounding plug-ins. Select the one ending with ".DiscountPricingTask".
6. Once selected, you will see a list of windows to fill in. These are the parameters needed to provide jBilling with the right information to implement your new rule. Different plug-Ins will require different fields.



7. The first box is labeled “Order” and this allows you to determine the order in which the rules will be applied. Please enter "1" here.
8. The next parameter is the Discount Category ID box. Please enter the category ID number you have for the “Discount products” category. This will apply the discount to that category of products.
9. You can now define the discount percentage at the product level using a meta field, by entering the meta field ID and entering 10 in the “Default Discount Percentage” box. Whole numbers here represent the percentage amount so 10 = 10%. Your fields should look like this.



10. Now, return to use that category and you will see the prices in the product category have not changed, but when applied to an order, the price is reduced by 10 percent.

You can add new products or edit pre-existing ones, and, in the discount meta-field, determine individual discount prices for those particular products. Try this out by entering 15 in the “Discount” meta field, and you will see it overrides the default value of the plug-in. If you do not add the value to the meta-field, the plug-in will put default value.

Even simpler, you can skip the meta field entirely. Just use the plug-in directly as a default value

for your entire line products. This takes away your ability to modify individual products, but it does help you see just how flexible and scaleable jBilling's plug-in system can be. Including your own plug-ins you build in Java can help your billing system grow even further, by providing you with ultimate customization.

As you can see, there is a lot you can do with this simple plug-in that was provided as an example. The best part is that you have the source of the plug-in. You can change it, enhance it, extend it, or create a new one entirely, all in Java. This is open source enterprise billing software at its best.

jBilling provides you with the tools you need to implement all your business rules quickly and easily, without needing to change core code. You simply chain together the list of plug-ins to generate the invoices you need.

How a company does its billing depends on a huge number of factors. The country where it operates is one, since it affects taxes, accounting rules, etc. The industry it belongs to is also a key factor: a phone company is more likely to bill its customer like its competition does than like a golf club charges its members. But business rules can also change from company to company as companies seek to stand out from their competition.

How do we face this endless list of requirements? We use the 'plug-in' design pattern, where we identify high level common requirements for a billing system, and we provide 'hooks' for areas that change from company to company. We also provide a default implementation for all these hooks, so jBilling is both a billing application framework and a fully operational billing system.

These hooks are designed as Java interfaces. jBilling only knows about these interfaces, not the actual implementations. When it tries to get on-line authorization for a credit card payment, it does not know which payment processor is being used, and how to communicate with it. It only calls 'process' on a Java object implementing the interface 'PaymentTask'. All the configuration of these plug-ins is in the database, so it is easy to change without any recompilation.

You can extend a default implementation, or you can implement from scratch one of the interfaces. In either case, you are extending jBilling to fit your needs without modifying jBilling itself.

Class parade

Types

Eventually, any Java system is just a bunch of classes, and jBilling is no exception. Still, not all classes are created equal. In jBilling they are grouped by their roles. You can tell what kind of role a class is by its name. By going over the major groups, you will have an idea of how the system was designed. Then, we'll present a sequence diagram to illustrate an example.

Actions

These client-tier classes extend Grails controller Action class to process requests coming from the user interface. In most cases, they call a Spring managed bean in the server side to pass

the user's information to the classes responsible of the business logic.

Class name: ***PaymentSessionBean***

Example: ***com.sapienter.jbilling.server.payment.PaymentSessionBean***

Business Logic (BL)

These are POJOs where all the business logic lives. In most cases, these classes act upon one row in the database through a persisted bean that is a member of the class. You use a BL class to find, create update or delete artifacts such as a payment, order or invoice, and to execute business logic related to them.

Class name: ***PaymentBL***

Example: ***com.sapienter.jBilling.server.payment.PaymentBL***

Let's take a look at a simplified version of this class. Note the association to ***PaymentDTO*** that represents one persisted bean, thus, one row in the database. The user clicks on a payment to see its content. After the click, you know the ID of the payment. You just do this:

```
PaymentBL myPayment = new PaymentBL(paymentId);
showPaymentDetails(myPayment.getDTO());
```

This will fetch the payment from the database and populate a Java bean with its content. This bean will be received by the method ***showPaymentDetails***.

Pluggable Tasks

These are the interfaces and concrete implementations of the business rules plug-ins described in the previous section. Here there is a brief list of the types of plug-ins. For a complete list and thorough overview of how plug-ins work, see the plug-ins chapter.

- *InvoiceCompositionTask*: Creates an invoice document based on the orders and/or invoices selected by the billing process.
- *InvoiceFilterTask*: Has the logic to decide if an older invoice should be carry over to a new invoice.
- *NotificationTask*: Knows how send a notification to a customer (for example, sending an email). This allows for other notification types such as fax, automated phone call, etc.
- *OrderFilterTask*: Decides if an order should be included in an invoices for the current billing process or not.
- *OrderPeriodTask*: Decides how many periods an order should include in an invoices.
- *OrderProcessingTask*: Calculates the total of an order when it is created, and might add some additional processing, like calculating sales taxes such as VAT.
- *PaymentInfoTask*: Decides how a customer will pay.
- *PaymentTask*: Submits a payment to a payment processor to get on-line payments for

credit cards or other electronic payment methods.

- *PenaltyTask*: Calculates potential penalties for customers that are late with their payments.

Class name: *name*Task

Example: *com.sapienter.jBilling.server.pluggableTask.PaymentAuthorizeNetTask*

Session Beans

We use the facade pattern, wrapping components and exposing each of them as a Spring managed bean. These classes act as a bridge between the client and the business logic classes, thus implementing the session facade design pattern. They shouldn't do much more than forwarding the calls, although sometimes some code manages to grow inside them :).

An important consideration is that transaction demarcation happens *only* in this session beans. When a client calls the server, a session bean receives the call and starts a transaction. The same applies anywhere in the code when a new transaction is needed: a bean is return by Spring, which starts a transaction. We use only declarative transaction management .

Class name: ***name*SessionBean**

Example: ***com.sapienter.jBilling.server.payment.PaymentSessionBean***

DB Persisted Beans

All direct access to a single row in the database is done with Hibernate managed beans. The result is that almost all database tables have an Hibernate annotated class as a counterpart.

We use Hibernate associations extensively as well. Do you want to get the invoice lines of an existing invoice? It is as easy as ***invoice.getLines()*** and because we know that an invoice doesn't have thousands and thousands of lines, it will perform just fine.

All these Hibernate managed classes used to be EJB entity beans. You will find many scar tissues, like helper methods to help with the migration.

A common complaint is the name of these classes. Why DTO? Especially since DTO is a naming pattern that is used for other purposes. The naming pattern was used as they are transferring data (from the DB to the application) and it's good to have a name for a class that, if modified, modifies the database. PaymentDTO is clearer than Payment.

Class name: ***name*DTO**

Example: ***com.sapienter.jBilling.server.payment.db.PaymentDTO***

Processing Flow

Let's put all these pieces together with a simple example of a complete execution flow. When the user is shown a list of payments, they can select one to see all its details. We'll follow how the major classes interact together across the tiers, starting with a sequence diagram. It does

show the old names from the Entity beans time, but for the most part it still applies today:

- All starts with the user clicking on a payment row. That sends a request to the web server with a parameter with the payment ID to be displayed.
- The request is forwarded to an Grails Controller Action class, in this case ***PaymentController*** class. This class will make some validations, parse the request to extract the payment ID, locate the payment session bean from the Spring context and make the call. No business logic here, since we are still in the client tier.
- The application server gets called through a session bean. In this case it only create the ***PaymentBL*** object and calls one of its methods—literally two lines of code. A transaction is started at this point using the declarative transaction demarcation offered by Spring.
- ***PaymentBL*** is created using the constructor that takes an ID as a parameter. It can right away look for the Hibernate bean that represents this payment in the database.
- A new DTO representing this payment is created.
- This DTO is then passed all the way back to the client tier, where it is placed in the HTTP session.
- The Grails controller then forwards the user to the payment (gsp) view page. This GSP knows how to display a payment based on the DTO object present in the request.

Chapter 2
Report templates

Extracting real-time data

CHAPTER 2: Report Templates

Behind every jBilling report is a Jasper Report file that queries the database and formats the information into a readable report. These reports can include logos, charts, and other graphical elements to provide information to the user in a clean and concise manner.

You need to be familiar with the database tables of the jBilling schema and SQL in order to write a new report. You also need to be familiar with Jasper Report JRXML files and the Jasper iReport design tool.

What Is a Report?

As mentioned, a report is a Jasper Report file with a query that jBilling loads and runs against the database to produce a readable report. To accomplish this task, all reports must be located in an appropriate place in the file system so they are accessible to jBilling.

You tell jBilling a report exists by creating an entry in the **REPORT** database table and giving it an appropriate type ID, a report name, and the filename of the Jasper Report. Every report must belong to a report type. These types are used for organizational purposes in the jBilling menu system, and also to dictate where the Jasper Report file can be found in the file system.

A Jasper Report file must be located within the correct folder on the “resources/reports/” path to be used with jBilling. Each report type has its own subfolder within the “resources/reports/” path, named the same as the report type.

In addition to the Jasper Report file, there is also a GSP (Groovy Server Pages) template page that is used to display the appropriate input fields and UI elements for each report when it is viewed in jBilling. As with the Jasper Report file, the report type dictates the path to the template page, and each report type has its own subfolder.

Type	Report Path	UI Template Path
1. Invoice	resources/reports/invoice	grails-app/views/report/invoice
2. Order	resources/reports/order	grails-app/views/report/order
3. Payment	resources/reports/payment	grails-app/views/report/payment
4. User	resources/reports/user	grails-app/views/report/user

Let's review the **REPORT** database table and how it relates to the Jasper Report file and the GSP template page.

Column Name	Description
ID	A unique identifier for this report.

TYPE_ID	Report type ID that refers to one of the types in the list above. The report type dictates the list in which the report appears in the jBilling UI, and where to find the Jasper Report file and GSP template page.
NAME	A unique name for this report that should only contain letters, numbers and underscores – e.g., “total_payments”.
FILE_NAME	The filename of the compiled .jasper report file as it can be found in the resources/reports/type/ folder – e.g., “total_payments.jasper”
OPTLOCK	Version of the report. Currently not used by jBilling. Set to “0” for new reports.

Jasper Report File Location

The path to the Jasper Report file is built by adding the report FILE_NAME to the path of the report type sub-folder, for example:

Report type: Payment—“resources/reports/payment/”

File name: “total_payments.jasper”

Jasper Report file = “/resources/reports/payment/total_payments.jasper”

GSP Template Page Location

The path to the GSP template page is built by adding the report name to the path of the report-type sub-folder and adding a .gsp suffix, for example:

Report type: Payment—“grails-app/views/report/payment”

Report name: “total_payments”

GSP template page = “grails-app/views/report/payment/_total_payments.gsp”

Note that Grails denotes a template file by prefixing it with an underscore.

Report Parameters

Every report has a collection of parameters with a name and a type. The type ensures that the parameter value is stored and passed on to the reporting engine in the correct format so that no conversion is necessary to parse the value in the Jasper Report JRXML.

The parameter is passed to the reporting engine by name, meaning you can reference it in the Jasper Report JRXML using the parameter keyword and name in the format $P{myParameter}.

Column Name	Description
ID	A unique identifier for this report parameter.
REPORT_ID	The unique identifier of the report that this parameter belongs to.
DTYPE	Musb be one of “integer”, “date”, or “string”. This denotes the type of object holding the value that is passed to Jasper Reports. • “integer” = java.lang.Integer • “date” = java.lang.Date • “string” = java.lang.String
NAME	Parameter name to be passed to Jasper Reports. The parameter value can be referenced by this name in the Jasper Report JRXML file.

Global Parameters

These report parameters are passed into the reporting engine for every report:

- REPORT_LOCALE—The local of the user running the report. Used for formatting.
- SUBREPORT_DIR—The path in the file system of this reports sub-folder.
- entity_id—The entity (company) ID of the user running the report.

GSP Template Page

As mentioned above, the GSP page templates can be found in the jBilling source code in a subfolder reflecting the type of the report. The template name itself must match the report name so that it can be rendered when the report is selected from the menu.

The GSP page templates are only used to provide the input elements to gather parameters for each report. By having a template for each report, you can ensure that only relevant input fields and options are presented to the user.

The simplest reports require no parameters and have templates that only serve to tell the user that there are no parameters to be entered:

```
<div class="form-columns">
	<p>
		<em><g:message code="report.no.parameters"/></em>
	</p>
```

```
</div>
```

More complex GSP page templates provide a form that can contain input fields, drop-down selection menus and date controls to allow the user to enter their own parameters values. To make things simple the report engine will extract entered values from the form by name. You only need to make sure the “name=” attribute of the field matches the report parameter name.

Parameters:

- Start Date (type: “date”, name: “start_date”)
- Number (type: “integer”, name: “number”)

```
<div class="form-columns">
       <g:applyLayout name="form/date">
           <content tag="label">Start Date</content>
            <content tag="label.for">start_date</content>
            <g:textField class="field" name="start_date"/>
       </g:applyLayout>

       <g:applyLayout name="form/select">
            <content tag="label">Number</content>
            <content tag="label.for">number</content>
            <g:select name="number" from="[1, 2, 3]" valueMessagePrefix="period"/>
       </g:applyLayout>
</div>
```

For more examples of input forms, take a look at the existing templates in the grails-app/views/report/ folder of the jBilling source code.

Internationalization

Internationalization of reports only covers the report name in jBilling menus, the parameter names and a brief description shown in the report view. To localize the report itself you'll need to refer to the Jasper Report documentation and make use of the passed **REPORT_LOCALE** parameter.

Report Name and Parameters

The text shown for report parameters in the jBilling report view can be localized by using the built-in Grails g:message tag, and adding the appropriate text to the messages.properties resource bundle (located in the grails-app/i18n/ folder) for your locale.

GSP template page:

```
<g:applyLayout name=form/date>
       <content tag="label">
       <g:message code="start.date"/>
       </content>
      ...
</g:applyLayout>
```

Messages.properties:

```
start.date=Start Date
```

Report Description (optional)

Reports can be can be given an optional internationalized description in different languages by adding entries to the jBilling **INTERNATIONAL_DESCRIPTION** database table using the ID of the target language. This is entirely optional, if there is no description set for a report then the “description” of the report shown in the report view will be left blank.

Example description for report ID 4:

```
insert into international_description
       (table_id, foreign_id, psudo_column, language_id, content)
values
       (100, 4, 'description', 1, 'Total payment amount received.');
```

Available languages can be determined by examining the **LANGUAGE** database table.

Example of New Report

Define the Report in the database

Report

Type: Payments—ID 3

Name: “total_payments”

File Name: “total_payments.jasper”

```
insert into report (id, type_id, name, file_name, optlock)
values (4, 3, 'total_payments', 'total_payments.jasper', 0);
```

Report Parameters

1. start_date type “date”
2. end_date type “date”
3. period type “integer”

```
insert into report_parameter (id, report_id, dtype, name)
values (5, 4, 'date', 'start_date');

insert into report_parameter (id, report_id, dtype, name)
values (6, 4, 'date', 'end_date');

insert into report_parameter (id, report_id, dtype, name)
values (7, 4, 'integer', 'period');
```

International Description

```
insert into international_description
       (table_id, foreign_id, psudo_column, language_id, content)
values
       (100, 4, 'description', 1, 'Total payment amount received.');
```

Mapping Report to a Description

```
insert into entity_report_map (report_id, entity_id)
values (4, 1);
```

Adding Report Files

Add a new Jasper Report JRXML file:

`descriptors/reports/payment/total_payments.jrxml`

You can compile the JRXML file using the "**grails compile-reports**" command from the jBilling source code. Alternatively, you can compile the file from within the Jasper iReports designer and move it to:

`resources/reports/payment/total_payments.jasper`

Add a new GSP template page:

`grails-app/views/report/payment/_total_payments.gsp`

```
<div class="form-columns">
       <g:applyLayout name=form/date>
           <content tag="label">Start Date</content>
            <content tag="label.for">start_date</content>
            <g:textField class="field" name="start_date"/>
       </g:applyLayout>

       <g:applyLayout name=form/date>
           <content tag="label">End Date</content>
            <content tag="label.for">end_date</content>
            <g:textField class="field" name="end_date"/>
       </g:applyLayout>

       <g:applyLayout name=form/select>
            <content tag="label">Period</content>
            <content tag="label.for">period</content>
            <g:select name="period" from="[1, 2, 3]" valueMessagePrefix="period"/>
       </g:applyLayout>
</div>
```

Add the report name to the messages.properties bundle:

```
total_payments=Total Payments
period.1=Day
period.2=Week
period.3=Month
```

Chapter 3
Business Rule Plug-ins

The Key to Extending jBilling

CHAPTER 3: Business Rule Plug-ins

A billing system needs to face a very difficult challenge: it needs to work following a company's business rules, and different companies have different business rules. Some industries work with prepayments, other get paid after the service is given. Every country has different tax and accounting rules, and even when many factors are the same, such as industry and location, companies may still decide on different approaches to billing.

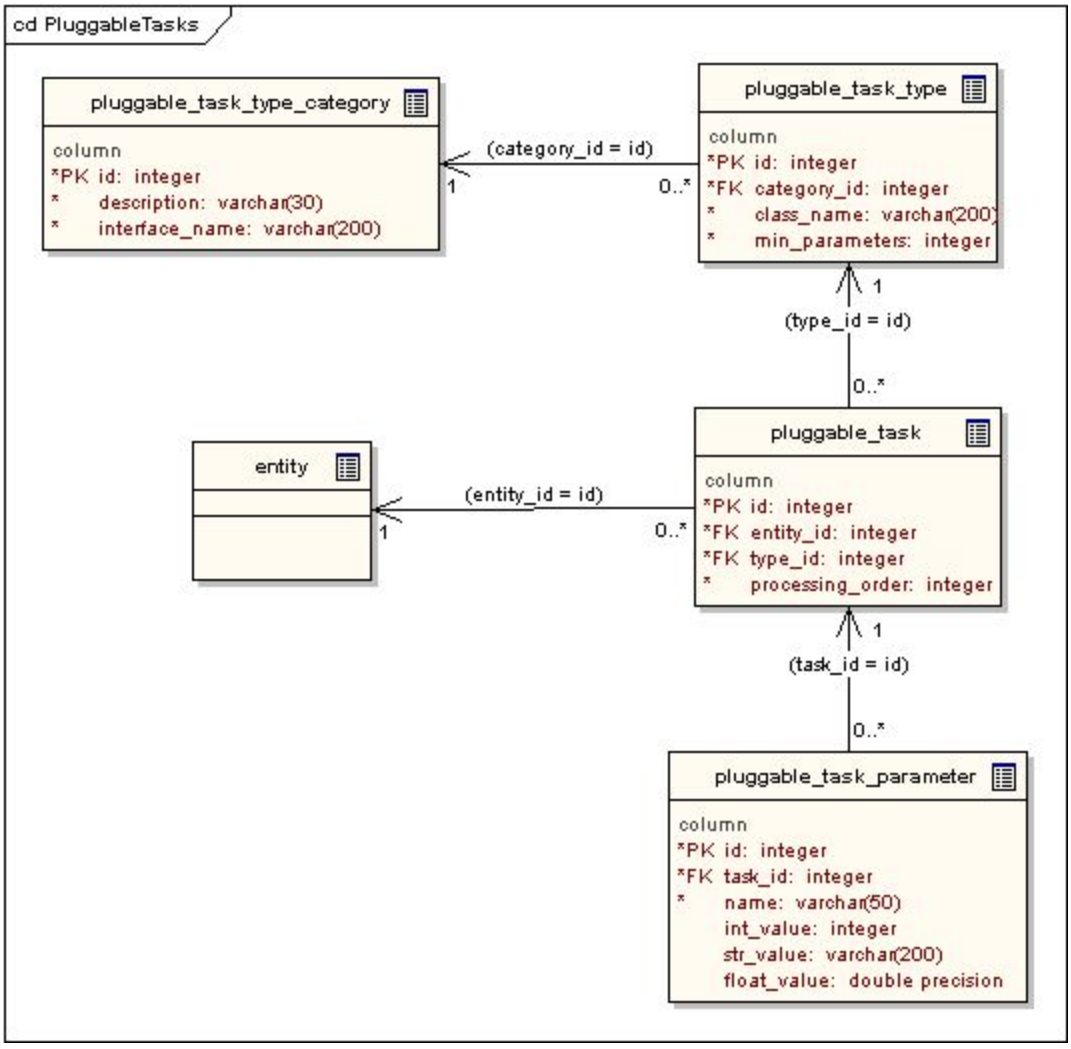
jBilling addresses this by allowing its key objects to be parametrized. Orders can be prepaid or postpaid. But that is not enough: there is need for further flexibility.

A plug-in is a design pattern that tackles this problem. The basic idea is to identify those areas of the billing system that are subject to a lot of different requirements. Then, we encapsulate them into objects and design the system to find out only at run time what objects needs to use. The configuration of jBilling, stored in the database, is what determines the class name that will be used.

One instance of jBilling can serve multiple companies, and those companies do not “know” about each other. We can have one instance running for a company in Italy and for another company in Canada. When the billing process runs, the right plug-in is used to calculate the taxes, which are very different between Italy and Canada.

The Business Rule Plug-in Architecture

There are many areas of the billing system that need their business logic encapsulated as a plug-in. Each of these areas are represented as a plug-in category. Then each category maps to a specific Java interface, which then can have many implementations. The implementations are named 'plug-in types'. So categories are interfaces and types are concrete classes.



Plug-ins are named in the code as pluggable tasks. Let's take a look at how the configuration data of the plug-in engine is represented as tables. From the previous diagram, we can make some statements:

You should not need to directly modify the contents of these tables. This can be done through the GUI by clicking on Configuration and then 'Plug-ins'. See the user guide for more information.

- Categories are Java interfaces, and they have system wide scope (*pluggable_task_type_category*). Each category (interface) can have multiple implementations. These will be Java classes; their scope is also system wide (*pluggable_task_type*).
- Each company (entity) might use a different class to do the same thing. What each company is using is mapped by *pluggable_task* table.
- Plug-ins have parameters. Many companies might share the same class to deal with a business rule, but each can have its own parameters (*pluggable_task_parameter*).

Let's put all this in an example. A payment processor is a plug-in; it handles how to get a credit

card payment cleared by a payment gateway. In our example, we'll have three companies (red, blue, yellow) and two payment gateways (big and small). Red and blue are going to use the big gateway, while yellow will use the small one.

The configuration will look like:

- The category is already set by the initialization data of jBilling. In this case, the row is the ID 6 that declares the interface *PaymentTask*.
- For the type, we will have two classes, one for each payment processor.
- In pluggable_task, we will have three rows, one for each company. Two of these rows point to the same type, because companies red and blue both use the 'big' gateway.
- Each row in *pluggable_task* will have its own parameters in *pluggable_task_parameter*. Here is where data such as the username and password to use the payment gateway go. With it, we can give company red its own credentials to use the big gateway, and the same thing goes for blue.

How It Works

In jBilling, originally plug-ins were tasks which were only called from specific points within the jBilling system. For example, the Notification Task. These tasks were not scheduled or schedulable and for that matter could not even handle or respond to events. Later, new plug-ins or tasks were added that could be hooked to Events. There plug-ins were 'Event aware' and got invoked as a result an Event. A full description of the jBilling Event architecture can be understood in Chapter 8 Internal Events.

Today, plug-ins or tasks may not be explicitly called by the code but can be scheduled as Quartz jobs. At the time of system startup, these plug-ins or Pluggable Tasks, that are already configured, are pulled from the database and scheduled as Quartz jobs depending on each plug-in's parameter values.

Therefore, jBilling has three varieties of plug-ins. These are also called Pluggable Tasks, stressing their extensibility, and can be classified as below:

1. Core driven—These are the original jBilling plug-ins or tasks that are called from various locations within the jBilling system
2. Event driven – These are the plug-ins that subscribe to one or more jBilling Event or a custom jBilling Event
3. Schedule driven – The plug-ins which can be scheduled based on parameters like date, time etc. or a Cron expressions fall under this category

Core-Driven Plug-ins

These are some of the originally developed plug-ins that belong to the core of the jBilling system. These are invoked or called from various points within the code depending on their use and functionality.

For Example, Notification Task (***INotificationSessionBean***) can be called on various places such as during Invoice generation, successful payment or an order being placed.

Other notable examples of Core driven plug-ins are Mediation Plug-in, Payment Processor and the Interest plug-in.

Event-Driven Plug-ins

These plug-ins act as handlers for a jBilling Event by subscribing to one or more Events (Refer Chapter 8 Internal Events). The plug-in class defines a ***process(..)*** method that performs the business-logic of the plug-in. As a plug-in writer, you would be responsible for subscribing the right events that need to be processed by performing the encapsulated business logic in this method.

As an example, the ***FileInvoiceExportTask*** performs the task of writing new Invoices to an export file. Therefore, this plug-in subscribes to the ***NewInvoiceEvent***. Whenever, a new Invoice is created within the jBilling system, the ***NewInvoiceEvent*** is fired. The jBilling system handles the process of invoking the subscribing ***FileInvoiceExportTask*** class in this case.

Schedule-Driven or Scheduled Plug-ins

Scheduled Plug-ins are Java classes within jBilling that extend from the abstract class ***ScheduledTask***. These classes may contain any business logic that is required to be executed at an instance of time or period, which may be repeatable or non-repeatable. Once scheduled, it will be the system's responsibility to execute these classes at the designated time and interval.

Like all plug-ins, Scheduled plug-ins can take 'pluggable parameters' as described in the **'plug-in architecture'** in the previous section. Depending on the type of parameters, the Scheduled plug-in can further be classified as Simple Scheduled Tasks and Cron Scheduled Tasks.

Simple Scheduled Tasks

Simple Scheduled Tasks are instances of an abstract Java class ***AbstractSimpleScheduledTask***. These tasks may require following parameters for scheduling namely:

- Start Time —Start time for the task in *yyyyMMdd-HHmm* format
- End Time – End time for the task in *yyyyMMdd-HHmm* format
- Repeat – A number to represent the number of times this task should repeat, default is infinite times
- Interval – Hours between two schedules of execution; default is 24 hours

A Simple Scheduled Plug-in may also be backward compatible for scheduling purposes. This means that certain tasks/processes or plug-ins that were originally scheduled via jbilling.properties configuration may also be implemented as a Simple Scheduled plug-in and jBilling may continue to schedule them based on the prior configurations. These plug-ins are instances of an abstract Java class ***AbstractBackwardSimpleScheduledTask***.

An example of the Simple Scheduled Task is the *BillingProcessTask*. This task runs the Billing Process within the jBilling system. The *MediationProcessTask* is also an example of *AbstractBackwardSimpleScheduledTask*. This is deprecated and we should use cron based

schedule tasks from now on.

Cron Scheduled Tasks

Cron Scheduled Tasks are instances of abstract Java class *AbstractCronTask*. These tasks can be scheduled using a Cron expression. Therefore, there is an additional flexibility and control that comes with a cron expression and the same can be put to good use for suitable business purposes.

Plug-in Categories

Categories are predefined in jBilling. They are associated with an area of the system that is intended to be easily extended. For example, which order should go into an invoice. This could be a simple or very complex algorithm, and can vary a lot from company to company, so there is a plug-in category to allow the implementation of this logic in a way that keeps it encapsulated and easy to plug-in to jBilling.

The following is a list of plug-in categories. It includes a brief description of each as an overview of them. To fully understand when the category is used and for what, it is necessary to review an implementation. These are explained in the remaining chapters.

ID	1
Name	Order Processing
Interface	com.sapienter.jBilling.server.pluggableTask.OrderProcessingTask
Description	Calculates the total amount of an order, based on the order lines. Typically extended to add 'automatic' items, such as taxes (VAT, GST, etc).

ID	2
Name	Order Filter
Interface	com.sapienter.jBilling.server.pluggableTask.OrderFilterTask
Description	Verifies if an order should be included in an invoice for the billing process

ID	3
Name	Invoice filter
Interface	com.sapienter.jBilling.server.pluggableTask.InvoiceFilterTask

Description	Decides if an invoice with outstanding balance should be carried over to a new invoice.

ID	4
Name	Invoice composition
Interface	com.sapienter.jBilling.server.pluggableTask.InvoiceCompositionTask
Description	Creates an invoice from a given order/s or invoice/s.

ID	5
Name	Order Period
Interface	com.sapienter.jBilling.server.pluggableTask.OrderPeriodTask
Description	Calculates the start and end dates of the period of an order to be included in an invoice.

ID	6
Name	Payment Gateway
Interface	com.sapienter.jBilling.server.pluggableTask.PaymentTask
Description	Submits a payment request to a payment gateway, usually to clear a credit card or ACH payment.

ID	7
Name	Notification
Interface	com.sapienter.jBilling.server.pluggableTask.NotificationTask
Description	Sends a notification to a customer.

ID	8
Name	Payment method

Interface	com.sapienter.jBilling.server.pluggableTask.PaymentInfoTask
Description	Finds and selects the payment information prior to submitting a payment.

ID	9
Name	Interests
Interface	com.sapienter.jBilling.server.pluggableTask.PenaltyTask
Description	Decides if a penalty (interest) is required for an overdue invoices, and if so it calculates the amount.

ID	10
Name	Gateway down alarm
Interface	com.sapienter.jBilling.server.pluggableTask.ProcessorAlarm
Description	Sends a notification if a payment gateway is down.

ID	11
Name	User subscription status manager
Interface	com.sapienter.jBilling.server.user.tasks.ISubscriptionStatusManager
Description	Handles the state machine where the transitions from statuses is defined.

ID	12
Name	Asynchronous payment parameters.
Interface	com.sapienter.jBilling.server.payment.tasks.IAsyncPaymentParameters
Description	Can add additional parameters to help distribute load for asynchronous payment processing.

ID	13
Name	Item Management
Interface	com.sapienter.jBilling.server.item.tasks.IItemPurchaseManager
Description	Executes adding an item into an order. It can decide to manipulate that item or the order by, for example, adding other items.

ID	14
Name	Item pricing (rating)
Interface	com.sapienter.jBilling.server.item.tasks.IPricing
Description	Gives an item a price.

ID	15
Name	Mediation record reader
Interface	com.sapienter.jBilling.server.mediation.task.IMediationReader
Description	Reads records from a source for the mediation process.

ID	16
Name	Mediation processor.
Interface	com.sapienter.jBilling.server.mediation.task.IMediationProcess
Description	Takes an event record and translates its fields to data jBilling can understand: which items are involved, the customer responsible for the event and the date of the event.

ID	**17**
Name	Internal Events.
Interface	com.sapienter.jBilling.server.system.event.task.IInternalEventsTask
Description	Plug-ins of this category will be called every time there is an internal

	event. The plug-in can subscribe to only some events. The information related to the event is passed to the plug-in as an ***Event*** object parameter

ID	18
Name	External Provisioning
Interface	com.sapienter.jBilling.server.provisioning.task.IExternalProvisioning
Description	Handles communication with external provisioning systems. It receives a command string it must interpret, communicates with the external system, then returns a *Map* of response parameters.

ID	19
Name	Purchase validation
Interface	com.sapienter.jbilling.server.user.tasks.IValidatePurchaseTask
Description	Purchase validation against pre-paid balance / credit limit.

ID	20
Name	Customer selection
Interface	com.sapienter.jbilling.server.process.task.IBillingProcessFilterTask
Description	Billing process: customer selection

ID	21
Name	Mediation Error Handler
Interface	com.sapienter.jbilling.server.mediation.task.IMediationErrorHandler
Description	Mediation Error Handler

ID	22

Name	Scheduled Tasks
Interface	com.sapienter.jbilling.server.process.task.IScheduledTask
Description	Plug-ins of this type are scheduled as Quartz jobs using a Quartz scheduler at the time of Application startup. Depending on the type of parameters, a Cron Expression or Start and Repeat instructions, this plug-in can be an AbstractCronTask or an AbstractSimpleScheduledTask.

ID	23
Name	Rules Generators
Interface	com.sapienter.jbilling.server.rule.task.IRulesGenerator
Description	Interface for Rules Generator task, which handles calls made to the generateRules API method. First, the unmarshal method is called to parse and validate the input string. The process method is then called to generate, compile and save the rules.

ID	24
Name	Ageing (collections) for customers with overdue invoices
Interface	com.sapienter.jbilling.server.process.task.IAgeingTask
Description	The business logic of what to do with customers that do not pay is encapsulated in this plug-in category.

ID	25
Name	Partner Commission Task
Interface	com.sapienter.jbilling.server.user.partner.task.IPartnerCommissionTask
Description	Agent Commission Calculation Process.

ID	26

Name	File Exchange Task
Interface	com.sapienter.jbilling.server.process.task.IFileExchangeTask
Description	Files exchange with remote locations, upload and download.

Plug-in Types

The default distribution of jBilling comes with several implementations of the plug-in categories. These implementations are the plug-in types, which we review briefly in this section.

Use the following list to quickly find the class that you need. From there, you can study, change or extend the class. We go into more details for each of the classes in the remaining chapters.

Category: 1	
Name	Default order totals
Class	com.sapienter.jBilling.server.pluggableTask.BasicLineTotalTask
Description	Calculates the order total and the total for each line, considering the item prices, the quantity and if the prices are percentage or not.

Category: 1	
Name	VAT
Class	com.sapienter.jBilling.server.pluggableTask.GSTTaxTask
Description	Adds an additional line to the order with a percentage charge to represent the value added tax.

Category: 1	
Name	Rules Line Total
Class	com.sapienter.jBilling.server.order.task.RulesLineTotalTask
Description	This is a rules-based plug-in (see chapter 7). It calculates the total for an order line (typically this is the price multiplied by the quantity), allowing for the execution of external rules.

Category: 1	
Name	Rules Line Total 2
Class	com.sapienter.jbilling.server.order.task.RulesLineTotalTask2
Description	This is a rules-based plug-in, compatible with the mediation process of jBilling 2.2.x and later. It calculates the total for an order line (typically this is the price multiplied by the quantity), allowing for the execution of external rules.

Category: 1	
Name	Avalara Tax
Class	com.sapienter.jbilling.server.payment.tasks.avalara.AvalaraTaxTask
Description	This plug-in provides integration with the Avalara Avatax service. The plug-in intercepts several events in the jBilling order lifecycle, e.g. order processing, invoice composition/creation and Payment etc.

Category: 2	
Name	Order Filter
Class	com.sapienter.jBilling.server.pluggableTask.BasicOrderFilterTask
Description	Decides if an order should be included in an invoice for a given billing process. This is done by taking the billing process time span, the order period, the active since/until, etc.

Category: 2	
Name	Anticipated order filter
Class	com.sapienter.jBilling.server.pluggableTask.OrderFilterAnticipatedTask
Description	Extends *BasicOrderFilterTask*, modifying the dates to make the order applicable a number of months before it would be by using the default filter.

Category: 3	
Name	Default Invoice Filter
Class	com.sapienter.jBilling.server.pluggableTask.BasicInvoiceFilterTask
Description	Always returns true, meaning that the invoice will be carried over to a new invoice.

Category: 3	
Name	No invoice carry over
Class	com.sapienter.jBilling.server.pluggableTask.NoInvoiceFilterTask
Description	Always returns false, meaning that the invoice will never be carried over to a new invoice.

Category: 3	
Name	Filters out negative invoices for carry over
Class	com.sapienter.jbilling.server.invoice.task.NegativeBalanceInvoiceFilterTask
Description	This filter will only filter out invoices with a positive balance to be carried over to the next invoice.

Category: 4	
Name	Invoice due date
Class	com.sapienter.jBilling.server.pluggableTask.CalculateDueDate
Description	A simple implementation that sets the due date of the invoice. The due date is calculated by adding the period of time to the invoice date.

Category: 4	
Name	Default invoice composition.
Class	com.sapienter.jBilling.server.pluggableTask.BasicCompositionTask

Description	This task will copy all the lines on the orders and invoices to the new invoice, considering the periods involved for each order, but not the fractions of periods. It will not copy the lines that are taxes. The quantity and total of each line will be multiplied by the amount of periods.

Category: 4	
Name	Country Tax Invoice Composition Task
Class	com.sapienter.jbilling.server.process.task.CountryTaxCompositionTask
Description	A pluggable task of the type SimpleTaxCompositionTask to apply tax item to the invoice if the partner's country code is matching.

Category: 4	
Name	Payment Terms Penalty Task
Class	com.sapienter.jbilling.server.process.task.PaymentTermPenaltyTask
Description	A pluggable task of the type AbstractChargeTask to apply a penalty to an Invoice having a due date beyond a configurable days period.

Category: 4	
Name	Suretax plug-in
Class	com.sapienter.jbilling.server.process.task.SureTaxCompositionTask
Description	This plug-in adds tax lines to invoice by consulting the Suretax Engine.

Category: 5	
Name	Default Order Periods
Class	com.sapienter.jBilling.server.pluggableTask.BasicOrderPeriodTask
Description	Calculates the start and end period to be included in an invoice. This is done by taking the billing process time span, the order period, the

	active since/until, etc.

Category: 5	
Name	Anticipate order periods.
Class	com.sapienter.jBilling.server.pluggableTask.OrderPeriodAnticipateTask
Description	Extends *BasicOrderPeriodTask*, modifying the dates to make the order applicable a number of months before it would be by using the default task.

Category: 6	
Name	Payment process for the Intrannuity payment gateway
Class	com.sapienter.jBilling.server.payment.tasks.PaymentAtlasTask
Description	Integration with the Intraanuity payment gateway.

Category: 6	
Name	Test payment processor
Class	com.sapienter.jBilling.server.pluggableTask.PaymentFakeTask
Description	A test payment processor implementation to be able to test jBilling's functions without using a real payment gateway.

Category: 6	
Name	CCF Router payment processor
Class	com.sapienter.jBilling.server.payment.tasks.PaymentRouterCCFTask
Description	Allows a customer to be assigned a specific payment gateway. It checks a custom contact field to identify the gateway and then delegates the actual payment processing to another plug-in.

Category: 6	
Name	Currency Router payment processor
Class	com.sapienter.jBilling.server.payment.tasks.PaymentRouterCurrencyTask
Description	Delegates the actual payment processing to another plug-in based on the currency of the payment.

Category: 6	
Name	Email and process authorize.net
Class	com.sapienter.jBilling.server.pluggableTask.PaymentEmailAuthorizeNetTask
Description	Extends the standard authorize.net payment processor to also send an email to the company after processing the payment.

Category: 6	
Name	ACH Commerce payment processor
Class	com.sapienter.jBilling.server.user.tasks.PaymentACHCommerceTask
Description	Integration with the ACH commerce payment gateway.

Category: 6	
Name	Blacklist filter payment processor.
Class	com.sapienter.jBilling.server.payment.tasks.PaymentFilterTask
Description	Used for blocking payments from reaching real payment processors. Typically configured as first payment processor in the processing chain. See the "*Blacklist*" section in the "*User Guide*" document.

Category: 6	
Name	Authorize.net payment processor

Class	com.sapienter.jBilling.server.pluggableTask.PaymentAuthorizeNetTask
Description	Integration with the authorize.net payment gateway.

Category: 6	
Name	Testing plug-in for partner payouts
Class	com.sapienter.jbilling.server.pluggableTask.PaymentPartnerTestTask
Description	Plug-in useful only for testing

Category: 6	
Name	Payment processor for Payments Gateway.
Class	com.sapienter.jbilling.server.payment.tasks.PaymentsGatewayTask
Description	Integration with the Payments Gateway payment processor.

Category: 6	
Name	Test payment processor for external storage.
Class	com.sapienter.jbilling.server.pluggableTask.PaymentFakeExternalStorage
Description	A fake plug-in to test payments that would be stored externally.

Category: 6	
Name	WorldPay integration
Class	com.sapienter.jbilling.server.payment.tasks.PaymentWorldPayTask
Description	Payment processor plug-in to integrate with RBS WorldPay

Category: 6

Name	WorldPay integration with external storage
Class	com.sapienter.jbilling.server.payment.tasks.PaymentWorldPayExternalTask
Description	Payment processor plug-in to integrate with RBS WorldPay. It stores the credit card information (number, etc) in the gateway.

Category: 6	
Name	Beanstream gateway integration
Class	com.sapienter.jbilling.server.payment.tasks.PaymentBeanstreamTask
Description	Payment processor for integration with the Beanstream payment gateway

Category: 6	
Name	Sage payments gateway integration
Class	com.sapienter.jbilling.server.payment.tasks.PaymentSageTask
Description	Payment processor for integration with the Sage payment gateway

Category: 6	
Name	Paypal integration with external storage
Class	com.sapienter.jbilling.server.payment.tasks.PaymentPaypalExternalTask
Description	Submits payments to PayPal as a payment gateway and stores credit card information in PayPal as well

Category: 6	
Name	Authorize.net integration with external storage
Class	com.sapienter.jbilling.server.payment.tasks.PaymentAuthorizeNetCI

	MTask
Description	Submits payments to authorize.net as a payment gateway and stores credit card information in authorize.net as well

Category: 6	
Name	Payment method router payment processor
Class	com.sapienter.jbilling.server.payment.tasks.PaymentMethodRouterTask
Description	Delegates the actual payment processing to another plug-in based on the payment method of the payment.

Category: 7	
Name	PDF invoice notification
Class	com.sapienter.jBilling.server.pluggableTask.PaperInvoiceNotificationTask
Description	Will generate a PDF version of an invoice to be included in batch for the billing process.

Category: 7	
Name	Email notifications
Class	com.sapienter.jBilling.server.pluggableTask.BasicEmailNotificationTask
Description	This implementation will send an email as a notification. It is the most typical way to notify a customer.

Category: 7	
Name	Notification task for testing
Class	com.sapienter.jBilling.server.notification.task.TestNotificationTask

Description	This plug-in is only used for testing purposes. Instead of sending an email (or other real notification), it simply stores the text to be sent in a file named *emails_sent.txt*.

Category: 8	
Name	Default payment information
Class	com.sapienter.jBilling.server.pluggableTask.BasicPaymentInfoTask
Description	Finds the information of a payment method available to a customer, giving priority to credit card. In other words, it will return the credit card of a customer or the ACH information in that order.

Category: 8	
Name	Payment information without validation
Class	com.sapienter.jBilling.server.user.tasks.PaymentInfoNoValidateTask
Description	This is the same as the standard payment information task, except that it does not validate if the credit card is expired. Use this plug-in only if you want to submit payment with expired credit cards.

Category: 8	
Name	Alternative Payment Info Task
Class	com.sapienter.jbilling.server.pluggableTask.AlternativePaymentInfoTask
Description	A pluggable task of the type Payment Info Task that first checks the preferred payment method, then if there is no data for the preferred method it searches for alternative payment methods

Category: 10	
Name	Email processor alarm
Class	com.sapienter.jBilling.server.pluggableTask.ProcessorEmailAlarmTask

Description	Sends an email to the billing administrator as an alarm when a payment gateway is down.

Category: 11	
Name	Default subscription status manager
Class	com.sapienter.jBilling.server.user.tasks.BasicSubscriptionStatusManagerTask
Description	It determines how a payment event affects the subscription status of a user.

Category: 12	
Name	Dummy asynchronous parameters
Class	com.sapienter.jBilling.server.payment.tasks.NoAsyncParameters
Description	A dummy task that does not add any parameters for asynchronous payment processing. This is the default.

Category: 12	
Name	Router asynchronous parameters
Class	com.sapienter.jBilling.server.payment.tasks.RouterAsyncParameters
Description	This plug-in adds parameters for asynchronous payment processing to have one processing message bean per payment processor. It is used in combination with the router payment processor plug-ins.

Category: 13	
Name	Basic Item Manager
Class	com.sapienter.jBilling.server.item.tasks.BasicItemManager
Description	It adds items to an order. If the item is already in the order, it only updates the quantity.

Category: 13	
Name	Rules Item Manager
Class	com.sapienter.jBilling.server.item.tasks.RulesItemManager
Description	This is a rules-based plug-in (see chapter 7). It will do what the basic item manager does (actually calling it), but then it will execute external rules as well. These external rules have full control on changing the order that is getting new items.

Category: 13	
Name	Rules Item Manager 2
Class	com.sapienter.jbilling.server.order.task.RulesItemManager2
Description	This is a rules-based plug-in compatible with the mediation module of jBilling 2.2.x. It will do what the basic item manager does (actually calling it), but then it will execute external rules as well. These external rules have full control on changing the order that is getting new items.

Category: 14	
Name	Rules Pricing
Class	com.sapienter.jBilling.server.item.tasks.RulesPricingTask
Description	This is a rules-based plug-in (see chapter 7). It gives a price to an item by executing external rules. You can then add logic externally for pricing. It is also integrated with the mediation process by having access to the mediation pricing data.

Category: 14	
Name	Rules Pricing 2
Class	com.sapienter.jbilling.server.item.tasks.RulesPricingTask2
Description	This is a rules-based plug-in compatible with the mediation module of jBilling 2.2.x. It gives a price to an item by executing external

	rules. You can then add logic externally for pricing. It is also integrated with the mediation process by having access to the mediation pricing data.

Category: 14	
Name	Mediation Process Task
Class	com.sapienter.jbilling.server.pricing.tasks.PriceModelPricingTask
Description	A scheduled task to execute the Mediation Process.

Category: 15	
Name	Separator file reader
Class	com.sapienter.jBilling.server.mediation.task.SeparatorFileReader
Description	This is a reader for the mediation process. It reads records from a text file whose fields are separated by a character (or string). The mediation module is covered in the document *"Telecom Guide"*.

Category: 15	
Name	Fixed length file reader
Class	com.sapienter.jBilling.server.mediation.task.FixedFileReader
Description	This is a reader for the mediation process. It reads records from a text file whose fields have fixed positions,and the record has a fixed length. The mediation module is covered in the document *"Telecom Guide"*.

Category: 15	
Name	JDBC Mediation Reader.
Class	com.sapienter.jBilling.server.mediation.task.JDBCReader
Description	This is a reader for the mediation process. It reads records from a JDBC database source. The mediation module is covered in the

	document *"Telecom Guide"*.

Category: 15	
Name	MySQL Mediation Reader.
Class	com.sapienter.jBilling.server.mediation.task.MySQLReader
Description	This is a reader for the mediation process. It is an extension of the JDBC reader, allowing easy configuration of a MySQL database source . The mediation module is covered in the document *"Telecom Guide"*.

Category: 16	
Name	Rules mediation processor
Class	com.sapienter.jBilling.server.mediation.task.RulesMediationTask
Description	This is a rules-based plug-in (see chapter 7). It takes an event record from the mediation process and executes external rules to translate the record into billing meaningful data. This is at the core of the mediation component, see the "*Telecom Guide*" document for more information.

Category: 16	
Name	Subscription event processor
Class	com.sapienter.jbilling.server.mediation.task.SubscriptionEventProcessor
Description	This subscription event based plug-in. It takes records from the mediation process and translates them into billing meaningful data. It centers on recurring orders. If no order is available it creates a new one.

Category: 16	
Name	Simple Mediation Processor

Class	com.sapienter.jbilling.server.mediation.task.SimpleMediationTask
Description	This is a simple mediation processor runs on file mediation-sample.csv.

Category: 16	
Name	Example Mediation Processor
Class	com.sapienter.jbilling.server.mediation.task.ExampleMediationTask
Description	This is an example mediation processor runs on file asterisk.csv.

Category: 16	
Name	Example Mediation Step Resolver
Class	com.sapienter.jbilling.server.mediation.step.ExampleMediationStep ResolverTask
Description	Example mediation plug-in used by jbilling tests. This implementation uses mediation steps to resolve the mediation business logic

Category: 17	
Name	Automatic cancellation credit.
Class	com.sapienter.jBilling.server.order.task.RefundOnCancelTask
Description	This plug-in will create a new order with a negative price to reflect a credit when an order is canceled within a period that has been already invoiced.

Category: 17	
Name	Fees for early cancellation of a plan.
Class	com.sapienter.jBilling.server.order.task.CancellationFeeRulesTask
Description	This plug-in will use external rules (see the BRMS chapter) to determine if an order that is being canceled should create a new

	order with a penalty fee. This is typically used for early cancels of a contract.

Category: 17	
Name	Blacklist user when their status becomes suspended or higher.
Class	com.sapienter.jBilling.server.payment.blacklist.tasks.BlacklistUserStatusTask
Description	Causes users and their associated details (e.g., credit card number, phone number, etc.) to be blacklisted when their status becomes suspended or higher. See the "*Blacklist*" chapter from the "*User Guide*" document.

Category: 17	
Name	Provisioning commands rules task.
Class	com.sapienter.jBilling.server.provisioning.task.ProvisioningCommandsRulesTask
Description	Responds to order related events. Runs rules to generate commands to send via JMS messages to the external provisioning module.

Category: 17	
Name	Default interest task
Class	com.sapienter.jbilling.server.pluggableTask.BasicPenaltyTask
Description	Will create a new order with a penalty item. The item is taken as a parameter to the task.

Category: 17	
Name	File invoice exporter.
Class	com.sapienter.jbilling.server.invoice.task.FileInvoiceExportTask

Description	It will generate a file with one line per invoice generated.

Category: 17	
Name	Rules caller on an event.
Class	com.sapienter.jbilling.server.system.event.task.InternalEventsRulesTask
Description	It will call a package of rules when an internal event happens.

Category: 17	
Name	Dynamic balance manager
Class	com.sapienter.jbilling.server.user.balance.DynamicBalanceManagerTask
Description	It will update the dynamic balance of a customer (prepaid or credit limit) when events affecting the balance happen.

Category: 17	
Name	Credit cards are stored externally.
Class	com.sapienter.jbilling.server.payment.tasks.SaveCreditCardExternallyTask
Description	Saves the credit card information in the payment gateway, rather than the jBilling DB.

Category: 17	
Name	Auto recharge
Class	com.sapienter.jbilling.server.user.tasks.AutoRechargeTask
Description	Monitors the balance of a customer and upon reaching a limit, it requests a real-time payment

Category: 17	
Name	Penalty Task on Overdue Invoice
Class	com.sapienter.jbilling.server.pluggableTask.OverdueInvoicePenaltyTask
Description	This task is responsible for applying a %-age or a fixed amount penalty on an Overdue Invoice. This task is powerful because it performs this action just before the Billing process collects orders.

Category: 17	
Name	Event based custom notification task
Class	com.sapienter.jbilling.server.user.tasks.EventBasedCustomNotificationTask
Description	The event based custom notification task takes the custom notification message and does the notification when the internal event occurred

Category: 17	
Name	Example web-service plug-in
Class	com.sapienter.jbilling.server.user.tasks.NewContactWebServiceTask
Description	This is an example plug-in that invokes a remote web-service when a new contact is created.

Category: 17	
Name	Credit on negative invoice
Class	com.sapienter.jbilling.server.invoice.task.ApplyNegativeInvoiceToPaymentTask
Description	This plug-in will set the balance and total of a negative invoice to 0 and create a 'credit' payment for the remaining amount.

Category: 17	
Name	Custom user notifications per ageing (collections) steps
Class	com.sapienter.jbilling.server.user.tasks.UserAgeingNotificationTask
Description	This plug-in provides mapping between the user status(ageing steps) and notifications that needs to be sent for each status

Category: 17	
Name	User Balance threshold notification task
Class	com.sapienter.jbilling.server.user.tasks.BalanceThresholdNotificationTask
Description	A pluggable task of the type InternalEventsTask to monitor if users pre-paid balance is below a threshold level and send notifications.

Category: 17	
Name	Suretax Delete Invoice plug-in
Class	com.sapienter.jbilling.server.process.task.SuretaxDeleteInvoiceTask
Description	This plug-in is used for sending cancel request to Suretax in case of invoice deletion. A cancel request is required by Suretax to update its records about cancellation of earlier made tax quotation request.

Category: 17	
Name	Example provisioning plug-in
Class	com.sapienter.jbilling.server.provisioning.task.ExampleProvisioningTask
Description	Example provisioning plug-in for jbilling tests.

Category: 17	
Name	ACH are stored externally.
Class	com.sapienter.jbilling.server.payment.tasks.SaveACHExternallyTask
Description	Saves the ACH information in the payment gateway, rather than the jBilling DB.

Category: 17	
Name	Updates Asset Transitions
Class	com.sapienter.jbilling.server.item.tasks.AssetUpdatedTask
Description	This plug-in will update AssetTransitions when an asset status changes.

Category: 17	
Name	Remove Assets From FINISHED Orders
Class	com.sapienter.jbilling.server.item.tasks.RemoveAssetFromFinishedOrderTask
Description	This plug-in will remove Asset owners when the linked order expires.

Category: 17	
Name	Order Cancellation Task
Class	com.sapienter.jbilling.server.order.task.OrderCancellationTask
Description	A pluggable task of type InternalEventsTask to monitor when activeUntil changes of an order.

Category: 17	
Name	Customer Usage Pool Update Task

Class	com.sapienter.jbilling.server.usagePool.task.CustomerUsagePoolUpdateTask
Description	This plug-in will update Customer Usage Pool when one of the following events occurs: NewOrderEvent, NewQuantityEvent, OrderDeletedEvent.

Category: 17	
Name	Processes Unsubscription Event of plan for customer. Updates Customer Usage Pools to expire them.
Class	com.sapienter.jbilling.server.usagePool.task.CustomerPlanUnsubscriptionProcessingTask
Description	This plug-in will expire Customer Usage Pools when a plan is unsubscribed by the customer.

Category: 17	
Name	Customer Usage Pool Consumption Task
Class	com.sapienter.jbilling.server.usagePool.task.CustomerUsagePoolConsumptionActionTask
Description	When a Usage Pool Consumption Event take place, this plug-in will fire defined action on the FUP for the given Percentage consumption

Category: 17	
Name	Generate payment commissions for agents.
Class	com.sapienter.jbilling.server.payment.tasks.GeneratePaymentCommissionTask
Description	This plug-in will create payment commissions when payments are linked and unlinked from invoices.

Category: 17	
Name	User Credit Limitation notification task

Class	com.sapienter.jbilling.server.user.tasks.CreditLimitationNotificationTask
Description	A pluggable task of the type InternalEventsTask to monitor if users' pre-paid balance is below a credit limitation 1 or 2 levels and send notifications.

Category: 17	
Name	Change order status on order change apply
Class	com.sapienter.jbilling.server.order.task.OrderChangeApplyOrderStatusTask
Description	This plug-in will change the status of order during order change apply to selected in order change.

Category: 17	
Name	Usage Pool Consumption Fee Charging Task
Class	com.sapienter.jbilling.server.usagePool.task.UsagePoolConsumptionFeeChargingTask
Description	When a Usage Pool Consumption Fee Charging Event take place, this plug-in will charge fees to the Customer

Category: 18	
Name	Test external provisioning task.
Class	com.sapienter.jBilling.server.provisioning.task.TestExternalProvisioningTask
Description	This plug-in is only used for testing purposes. It is a test external provisioning task for testing the provisioning modules.

Category: 18	
Name	CAI external provisioning task.

Class	com.sapienter.jBilling.server.provisioning.task.CAIProvisioningTask
Description	An external provisioning plug-in for communicating with the Ericsson Customer Administration Interface (CAI).

Category: 18	
Name	MMSC external provisioning task.
Class	com.sapienter.jBilling.server.provisioning.task.MMSCProvisioningTask
Description	An external provisioning plug-in for communicating with the TeliaSonera MMSC.

Category: 19	
Name	Balance validator based on the customer balance.
Class	com.sapienter.jbilling.server.user.tasks.UserBalanceValidatePurchaseTask
Description	Used for real-time mediation, this plug-in will validate a call based on the current dynamic balance of a customer.

Category: 19	
Name	Balance validator based on rules.
Class	com.sapienter.jbilling.server.user.tasks.RulesValidatePurchaseTask
Description	Used for real-time mediation, this plug-in will validate a call based on a package or rules

Category: 19	
Name	Example validate purchase
Class	com.sapienter.jbilling.server.user.tasks.ExampleValidatePurchaseTask

Description	Example validate purchase plug-in used by jbilling tests.

Category: 20	
Name	Standard billing process users filter
Class	com.sapienter.jbilling.server.process.task.BasicBillingProcessFilterTask
Description	Called when the billing process runs to select which users to evaluate. This basic implementation simply returns every user not in suspended (or worse) status

Category: 20	
Name	Selective billing process users filter
Class	com.sapienter.jbilling.server.process.task.BillableUsersBillingProcessFilterTask
Description	Called when the billing process runs to select which users to evaluate. This only returns users with orders that have a next invoice date earlier than the billing process.

Category: 20	
Name	Billable User Orders Filter
Class	com.sapienter.jbilling.server.process.task.BillableUserOrdersBillingProcessFilterTask
Description	This plug-in is used to find the users that are billable and have orders to be included in billing process.

Category: 21	
Name	Mediation file error handler
Class	com.sapienter.jbilling.server.mediation.task.SaveToFileMediationErrorHandler

Description	Event records with errors are saved to a file

Category: 21	
Name	Mediation database error handler.
Class	com.sapienter.jbilling.server.mediation.task.SaveToJDBCMediationErrorHandler
Description	Event records with errors are saved to a database table

Category: 22	
Name	Ageing (collections) process task
Class	com.sapienter.jbilling.server.process.task.AgeingProcessTask
Description	A scheduled task to execute the Ageing Process.

Category: 22	
Name	Mediation process task
Class	com.sapienter.jbilling.server.mediation.task.MediationProcessTask
Description	Scheduled mediation process plug-in, executing the mediation process on a simple schedule.

Category: 22	
Name	Billing process task
Class	com.sapienter.jbilling.server.billing.task.BillingProcessTask
Description	Scheduled billing process plug-in, executing the billing process on a simple schedule.

Category: 22	
Name	Delete old files

Class	com.sapienter.jbilling.server.pluggableTask.FileCleanupTask
Description	This task will delete files older than a certain time.

Category: 22	
Name	Customer Usage Pool Evaluation and Update Task
Class	com.sapienter.jbilling.server.usagePool.task.CustomerUsagePoolEvaluationTask
Description	This plug-in will evaluate and update Customer Usage Pools to set the Cycle End Dates as per Usage Pool Period and resets the quantities on them as per Usage Pool definition parameter.

Category: 22	
Name	Calculate Agents' Commissions
Class	com.sapienter.jbilling.server.user.partner.task.CalculateCommissionTask
Description	This task schedules the process that calculates the Agents' Commissions.

Category: 22	
Name	Trigger File Download/Upload
Class	com.sapienter.jbilling.server.process.task.FileExchangeTriggerTask
Description	Triggers File Download/Upload against third party system

Category: 24	
Name	Basic ageing
Class	com.sapienter.jbilling.server.process.task.BasicAgeingTask
Description	Ages a user based on the number of days that the account is overdue.

Category: 24	
Name	Business day ageing
Class	com.sapienter.jbilling.server.process.task.BusinessDayAgeingTask
Description	Ages a user based on the number of business days (excluding holidays) that the account is overdue.

Category: 25	
Name	Basic Agents' Commissions task
Class	com.sapienter.jbilling.server.user.partner.task.BasicPartnerCommissionTask
Description	This task calculates the Agents' Commissions.

Creating Your Own Plug-ins

When the default types do not meet your requirements, you can create your own. The most common result is an extension of a current type, or a new one that chains to an existing type.

An example of an extension is “Anticipate order periods”. It extends the default type to modify its behavior without having to redo the basic logic of it. A type that is meant to be chained is the VAT type. It will be called after the standard type to add an additional order line.

Take advantage of the fact that jBilling is open source, all the source code is there for you to study! Creating your own plug-in boils down to a new Java class and some inserts in the database to configure the system to use this class. As mentioned above, your new plug-in will be implementing an existing jBilling interface, or extending one of the existing types.

The first step for this is to identify the interface of the plug-in category. Next, see which are the existing types for that category. The easiest way to move forward is to take a look to the code of those types. Most of them are not large pieces of code and can be well understood with the help of this document. The following sections will go over those types in more detail.

As a general requirement, all types have to :

- Implement the interface that represents the plug-in category
- Extend the abstract class ***PluggableTask***

Once you have the new Java class that represents your type, you will need to make jBilling aware of this new type. This is done with one insert into the table *pluggable_task_type*. The following are its columns:

id: This is a unique, sequential integer that identifies this type. You need to find out the latest used number and add one to it:

```
select max(id)+1 from pluggable_task_type
```

category_id: The ID of the category that your type will belong to.

class_name: The full class name, including the package name. For example:

com.sapienter.jBilling.server.pluggableTask.BasicLineTotalTask

min_parameters: This in an integer with the minimum number of parameters that this type takes. It is used only for validation. If the type is wrongly configured, with less than this number of parameters, an exception will be thrown.

With your type registered in this table, you can proceed to add it to your company by clicking on 'Configuration', then 'Plug-ins'. The new type should be present in the drop down list of classes. This configuration screen is explained in the 'System' section of the user guide.

It is good practice to avoid modifying the default plug-in types. Ideally, you would either extend one or create your own from scratch. This would avoid running on a 'forked' jBilling source base.

Creating your Own Scheduled Plug-in

Creating a custom Scheduled Plug-in in jBilling is a multi-step process as below:

1. A custom Scheduled plug-in will be a new Java class that extends from one of the following classes depending on the requirement or configuration：
    a. *AbstractCronTask*
    b. *AbstractSimpleScheduledTask*
    c. *AbstractBackwardSimpleScheduledTask*
2. Provide customized implementation to the methods
    a. Method getTaskName()—This method returns a string to identify the task by a name
    b. Method execute()—This method is invoked by the Scheduler to execute the custom logic for this Pluggable Task. This method receives a *JobExecutionContext*, which can be used to retrieve the Plug-in parameters configurable via the jBilling system
    c. Method getTrigger()—Depending on the type of the parameters used for execution, the Quartz Scheduler requires an instance of *SimpleTrigger* class. A Pluggable Task is scheduled using the Job Parameters and an instance of Trigger class. A trigger requires a minimum of 2 properties; Start Time and Repeat Interval. Start time is in the format *yyyyMMdd-HHmm* where as the Repeat Interval is specified in hour
3. Configure the plug-in into the jBilling system by providing necessary parameters. jBilling Pluggable Tasks are configurable via the jBilling Web Interface as mentioned in a

previous section.

a. The *AbstractSimpleScheduledTask* requires a minimum of 4 parameters namely; *start_time*, *end_time*, *repeat* and *interval* between repeats. If however, these parameters are not configured, they default as follows: Start Date = Midnight 1st Jan, 2010, End Time = NULL meaning Infinite, Repeat = Repeat Indefinitely, Interval = 24 Hours

b. The *AbstractCronTask* requires a minimum of 1 parameter namely; *cron_exp* for the Cron Expression. A Cron Expression takes the form of a string with 5 to 6 numeric attributes separated by space. The definition of a Cron Expression can be looked up in the Quartz Scheduler manual. If not configured, this value defaults to "0 0 12 * * ?", which means, the task will be scheduled to run at 12 noon everyday.

Chapter 4
Payment Plug-ins

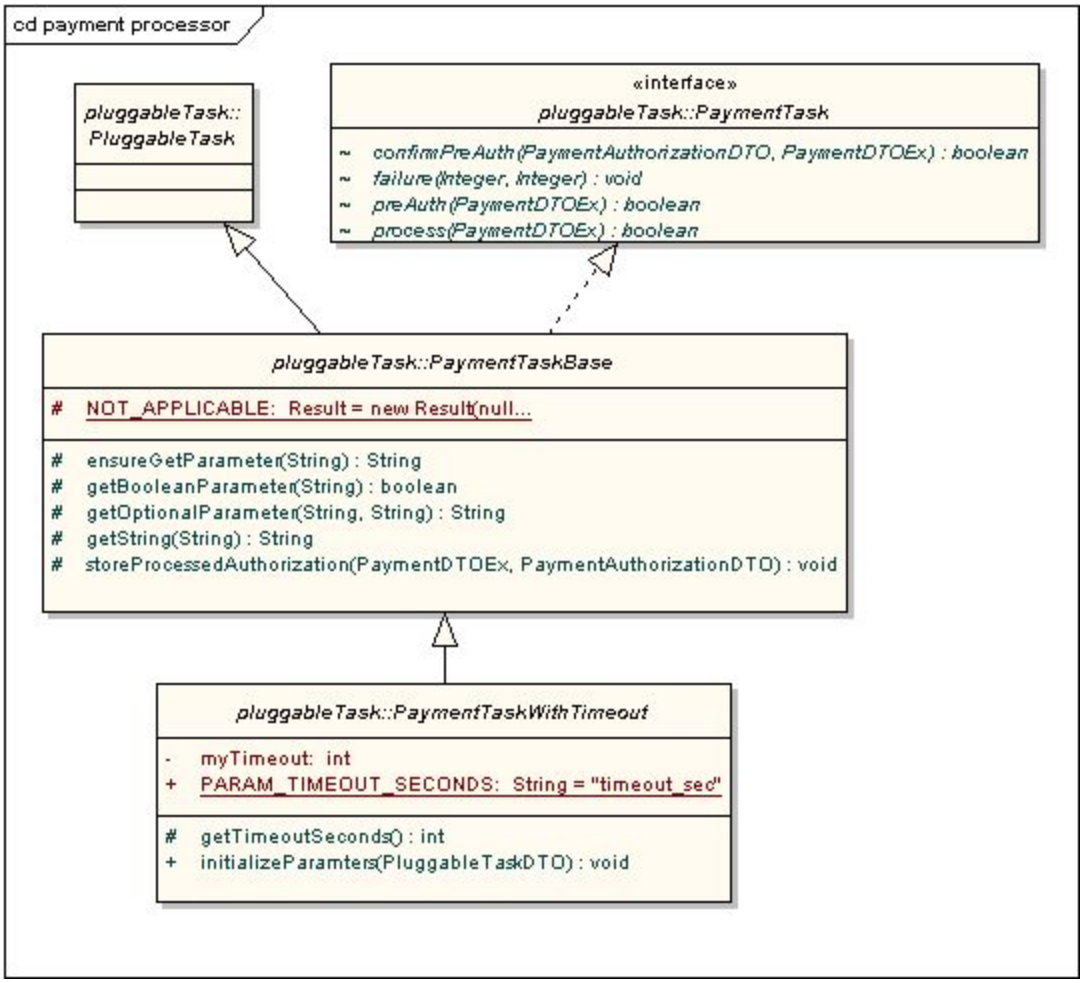
CHAPTER 4: Payment Gateway Integration

There are a number of payment gateways that provide payment processing services. Each of them implement their own API and require a particular transport protocol. However, a payment processor has one basic job to do: given a set of payment information (typically, an amount and credit card details), process the payment and return either success or failure.

In jBilling, we use the business plug-in architecture to implement payment processors. This means that each payment processor plug-in is just an implementation of an interface, and the configuration of which payment processor jBilling will use is stored in the database. Thus, the billing administrator can switch from one payment processor to another by just changing some rows in the database (which is done through the GUI), without any code changes or restarting the application server.

There is also the need to allow failover functionality: if a payment processor is down, try another one. Of course, a company would have to have an merchant account in more than one payment gateway for this to be possible.

You need to create a new business rule plug-in class. As shown in the diagram, it will extend ***PluggableTask*** and implement the interface ***PaymentTask***. Since there are many functions that are in common to pretty much any payment processor plug-in, we have grouped those functions into a convenient (abstract) class that implements the interface. The class is ***PaymentTaskWithTimeout***.



The PaymentTask Interface

Let's take a look to the interface *PaymentTask*:

```
public interface PaymentTask {
    boolean process(PaymentDTOEx paymentInfo)
             throws PluggableTaskException;
    void failure(Integer userId, Integer retry);
    boolean preAuth(PaymentDTOEx paymentInfo)
             throws PluggableTaskException;
    boolean confirmPreAuth(PaymentAuthorizationDTOEx  auth,
            PaymentDTOEx paymentInfo)
            throws PluggableTaskException;
}
```

A payment processor plug-in is nothing more than an implementation of the *PaymentTask*

interface. Typically, you will extend *PaymentTaskWithTimeout* and use all the helper methods that it provides. This will help you to deal with the plug-in parameters and to store the results of the payment in the database.

Where should you place your plug-in? To follow the jBilling standard, make your new class part of the following package: com.sapienter.jBilling.server.payment.tasks

The main method for you to code is ***process***, which takes a ***PaymentDTOEx*** object as a parameter. This way, the processor is not necessarily tied to a payment type, it can process credit cards, direct debit or whatever the market offers for real-time payment.

This method (as well as the others) will return **true** or **false**, which indicates if the next payment processor should be called by the business plug-in manager. This allows to fail-over to other payment gateway if this one is unavailable. Thus, the return value has nothing to do with the result of the payment, but if the payment was processed at all. In other words, if the payment goes through and the result is either success or failure, the return value should be false. If the communication with the payment processor fails (server down, timeout, etc), then return value should be true.

The ***failure*** method is called by the business plug-in manager after calling process if the result of the payment was a failure. The concept behind this is that the payment processor plug-in can then do something with the customer account, like suspend it, for example. This, though, has been obsoleted in favor of the ageing process, so you can make an empty implementation of this method.

The method ***preAuth*** will do a credit card pre-authorization of a fixed amount. This is usually done to verify that a credit cards is valid, without making a real charge. A payment gateway will drop the charge after some number of days if this pre-authorization is not confirmed by another call (also called capturing a pre-authorization).

Just like with **process**, we need to allow the caller to know if it is necessary to call another processor because this one is unavailable by returning 'true' or 'false'.

The method **confirmPreAuth** will take a transaction done with 'preAuth' and confirm it (aka 'capture'). With this, the original pre-authorization becomes a real charge to the credit card. This method takes as a parameter the return value of **preAuth**.

A payment processor will typically need the transaction ID to perform a previously authorized capture. In a way, this method does the same as process, but instead of doing it from scratch, it does it from an existing transaction, which translates on a higher chance that the process will be successful. The return value is the same as the previous methods..

Implementation Responsibilities

When implementing the ***PaymentTask*** interface, you need to follow these rules :

- Implement a timeout: When calling the payment processor, you cannot take forever. There has to be a maximum amount of time that, if reached, the result should be that the payment processor is unavailable. This timeout should be a parameter of the plug-in (see next point).
- Use parameters: Do not hardcode parameters like the username and password of the merchant account of the company using the payment processor. Use the business rules plug-ins parameters instead, that are easily available by methods in the parent abstract

class ***PluggableTask***.

- Update the payment with the result: When *process* is called, the payment object needs to get the result is updated. Make a call to ***setPaymentResult*** with a new instance of PaymentResultDTO using one of:
  Constants.RESULT_OK,
  Constants.RESULT_FAIL, OR
  Constants.RESULT_UNAVAILABLE
- Parse the processor results: Every processor will return the information about what happen with the transaction in its own way. You will have to parse these results and create an object ***PaymentAuthorizationDTO*** to hold this information. This objects goes into the database, as explained later.
- Return true/false for process: Always return false, except when the processor is not responding. This gives a chance to other processor to be called an attempt the payment.
- Add the authorization result to 'paymentInfo': The result of the authorization will go into a ***PaymentAuthorizationDTOEx*** object, and this object has to be added to the payment object you are getting as a parameter. This is to let the caller now the details of the transaction, and translates to one simple line of code:

  paymentInfo.setAuthorization(authorizationDto);
- Create an authorization record in the database: It is important to log the interaction with the payment processors. Here you need to create a record in a table meant for this, and link this record to the payment record. Note that the payment object is also updated with the authorization. This is easy because it is all encapsulated in their own object.

Here's an example for ***process***:

```
// create the response object with the data returned by the
// payment processor
PaymentAuthorizationDTO response = ...;
// set the processor name
response.getPaymentAuthorizationDTO().setProcessor("New processor");
// update the payment with the response: success, failure
// unavailable, etc.
payment.setPaymentResult(new PaymentResultDAS().find(Constants.RESULT_OK)); // parse
from response
// now create the db row with the results of this
// authorization call using a method from the parent
storeProcessedAuthorization(payment, response);
```

All the methods (***process***, ***preAuth*** and ***confirmPreAuth***) have to create this authorization

record linked to the payment record.

Example

The best way to get started might be to take a look to existing payment processor plug-ins. There are a few in the package:

com.sapienter.jBilling.server.payment.tasks

Any class with a name ending in 'Task' within that package is a payment processor plug-in. It is a good idea to follow this naming convention for your plug-in.

Testing

Once you have finished coding the class, how do you test it? There are three steps for testing:

1. Create a new plug-in type (this is your new class).
2. Configure your company to use that new type.
3. Submit a real-time payment.

To create a new type, you only need to add a new row to the **PLUGGABLE_TASK_TYPE** table. See the database diagram in Chapter 2. The new type will belong to category 6, which is the one for payment processing.

Here's an example of a type that takes at least 2 parameters:

```
INSERT INTO pluggable_task_type VALUES(38, 6,
'com.sapienter.jBilling.server.payment.tasks.PaymentMyGatewayTask', 2);
```

Now to the company configuration. Login to the GUI as an administrator, then click on 'Configuration' → 'Plug-ins'. Remove any plug-in that belongs to category 6. By default, that would be the *'PaymentFakeTask'*.

Create a new plug-in entry and select your class from the drop-down menu (it shows up now because of the previous step). Click on Submit so the changes are saved. You will probably need to add some parameters to your plug-in configuration as well: account number, password and the like.

Now to the real testing: submitting a payment to the gateway. Normally, you will be working with credit cards, so click on 'Payments' → 'Credit Card'. (ACH or other payment methods follow the same procedure, just with a different payment type).

Select the customer to create the payment for, and follow the normal steps to submit a payment. Just make sure that the 'Process real-time' check-box is selected, otherwise the payment will be entered without sending it to the payment gateway through your plug-in.

What should you expect? It is up to the payment gateway, not jBilling. It is common for gateways to provide test accounts and a set of credit card numbers that you can use and expect a specific response. Others choose to determine the response from the amount that is passed.

It varies from gateway to gateway.

Deciding on a Payment Method

Before a payment can be submitted to a payment gateway for processing, jBilling needs to determine how is that the customer is going to pay. The common answer is by credit card, but there are many other payment methods that can be used for automatic electronic payments.

The key concept for this plug-in type, is that there is a step in the billing process where the system determines the payment method to use for the customer being processed. That step is designed as a plug-in to allow companies to add new logic to this.

The default plug-in type is ***BasicPaymentInfoTask***. It will fetch the customer's credit card or ACH information depending which one of them has the flag 'Use for automated payments'. It filters credit cards that are not valid (expired, for example), to avoid sending request to the gateway that are known failures.

The basic contract to follow, when implementing your own type, is to return a ***PaymentDTOEx*** object with the payment method initialized. You can return ***null*** if the customer does not have any suitable payment method.

The widespread usage of credit cards as payment methods makes this plug-in category an unlikely candidate for custom implementations. Requirements that could lead to one include: customers in different geographical locations should use different payment methods, prioritize one payment method over another one, such as ACH to credit cards if the customer has both, etc.

Asynchronous Payment Processing

Asynchronous payment processing is an advance deployment feature that allows large number of payments to be batch processed. You would only need to use this feature if you have so many payments to process that you need to send more than one at a time to one or more payment gateways.

The typical scenario is that you have a daily billing process with automated payment processing. Some of your customers are being processed every day, and yet getting all the payments done takes too long. Theoretically, there is no problem as long a the payments get done before the next billing process takes place. In our example that is 24 hours.

Yet, that would mean continuous payment processing 24x7 and even that could not be enough if the payment load is too big.

The billing process does not process payments itself. It calculates and generates the invoices and then 'stacks' a payment request. It is now up to another process to pick up these requests and interact with the payment gateways. The payment process runs independently of the billing process, and it does so in a way that can spawn several concurrent processes.

This multi-processing capability allows you to configure jBilling to do multiple payment submissions simultaneously. There are many configuration options that let you tell jBilling exactly how is that you want to interact with your payment gateway(s). Sending more than one

payment at once increases the payment processing throughput of jBilling enormously. Just having two payments sent simultaneously literally means getting your payment processing done in half the time.

Still, your payment gateway has to support this. Can your payment gateway receive more than one request from you at a time? That is something you will need to find out. And if so, how many? Two? Five? Gateways can restrict the number of concurrent requests, and most probably do. Otherwise, they can be flooded with requests from just a few companies in a short period of time.

Another option for simultaneous payment processing is to have more than one account with different payment gateways, or even the same gateway. You can then submit only one payment request at a time for each of your accounts, but since you have more than one, you effectively process more than one payment simultaneously.

The previous two options are not exclusive of each other. You can also send many requests to many payment gateway accounts: jBilling allows you configure your payment processing in a way that you lets you scale up without limits.

In this section, we will go over these options, and also review a plug-in category that provides the ultimate flexibility for asynchronous payment processing.

Configuration

The payment processing is implemented in jBilling through the usage of Spring *message driven pojos* (MDP). Each bean is an independent payment processor, by default jBilling comes with just one bean configured. This bean will start processing payments from the queue as soon as the billing process queues one payment request.

To add more beans, you will need to modify some configuration files. The key file to look at is jbilling/conf/spring/resources.xml, in particular the following section:

```
  <bean id="processPaymentMDB"
class="com.sapienter.jbilling.server.payment.event.ProcessPaymentMDB"/>

    <!-- Mapping of MDBs to queues/topics they listen to -->

    <!-- Queue Listeners -->

    <jms:listener-container connection-factory="jmsConnectionFactory">

        <jms:listener ref="processPaymentMDB"
destination="queue.jbilling.processors"/>

        <!-- <jms:listener ref="processPaymentMDB"
destination="jbilling.processors.queue" selector="entityId = 1" /> -->

    </jms:listener-container>
```

You can find all the details on how to configure MDP and JMS in general in the Spring documentation.

Here we can say that you can easily add more beans to process payments. By doing this, you will have many beans to process payments at the same time, but this alone might not be enough. You might want to configure a particular bean to process some type of payment only. This can be helpful for the cases mentioned before where a payment gateway restricts the number of simultaneous requests it will accept from a single account.

The scope of payment processing for a bean is narrowed in the selector tag. Here you can enter a SQL style statement that will be applied as a filter to the message the bean will take for processing. By default, jBilling only exposes one field, entityId (which you can see an example commented out in the original file). A request will never be processed by more than one bean, but many beans can be processing (different) requests at the same time.

Imagine that the payment requests are in a queue. Each of them has a series of information fields needed for the payment to happen: the amount, user ID, invoice ID, etc. Each payment processing bean will start taking these requests from the queue to get them processed. If there are conditions stated in the selector section of the bean, that bean will only take those requests from the queue that satisfy the conditions.

If your jBilling installation is serving many companies, you can use the entityId field to assign one or more beans to each company. Then each company can have its own payment gateway account.

If you are assigning a payment processor to each customer by using the router payment processor plug-in, you can use the field 'processor' as well. You'd do this because that field is made available by the 'router asynchronous parameters' plug-in (see category 12).

In most situations, simply having two or three beans will solve your volume problems. If your payment gateway accepts processing that number of payments simultaneously, you configure this scenario very easily: add the new beans with just a name change and keep the message-selector empty.

If your scenario is more complex, and the beans need to evaluate more fields to pick the right payment to process, you will need to develop your own asynchronous parameters plug-in type.

Adding New Parameters for Asynchronous Processing

Any field present in the selector section needs to be added by a plug-in that implements the category ID 12 (with the sole exception of entityId, as mentioned earlier).

Take a look to ***RouterAsyncParameters*** class. This works with the router payment tasks to add the processor field. As you can see, this type of plug-ins will simply receive a JMS message as a parameter. Your task then is to add more fields to this message.

Any field added here can be used as a filter in the SQL style statement present in the selector for each bean.

When you create your own implementation of ***IAsyncPaymentParameters***, you might need to also have your own payment processor plug-in similar to the router processor. This is true if your filter criteria requires a different processor to be used for a specific MDP.

Chapter 5
Billing Process Plug-ins

CHAPTER 5: Billing Process Plug-ins

The billing process is the module that heavily uses plug-ins. This comes as no surprise, since this is the true core of a billing system. Those key points that can be left open for future extension were identified and designed to use business rules plug-ins.

What should an invoice contain? Which orders should generate invoices? How the various totals should be calculated? These and many other key billing questions are answered by independent classes, rather than being hard-coded.

In this section we will cover the plug-in categories related to the billing process and the existing implementations available in the default distribution. It is a good idea at this point to review the basics of jBilling's billing process by going to the user guide and reading the chapter dedicated to it. You must have a clear picture of the sequence of events happening in the billing process, in order to understand these plug-ins.

Order Filter

The order filer is a key component to the billing process. This object is called by the billing process and **decides if an order should generate an invoice or not**.

Interface	**com.sapienter.jbilling.server.pluggableTask.OrderFilterTask**

boolean isApplicable (OrderDTO order, BillingProcessDTO billingProcess)

Description: This method takes order object containing all order details and billing process object which contains all necessary details about the process.

BasicOrderFilterTask implementation of ***isApplicable(...)*** method:

Verifies if the order should be included in a process considering its active range dates. It takes the billing period type in consideration as well.

Argument:

order—*OrderDTO*—the order that is being considered to be included

billingProcess—*BillingProcessDTO*—billing process that is currently running

return—**true** if order should be included in the billing process, **false** otherwise

Class structure:

```
public interface OrderFilterTask {
   boolean isApplicable(OrderDTO order,
           BillingProcessDTO process) throws TaskException;
}
```

The default implementation, ***BasicOrderFilterTask***, considers the following properties of the order:

- Whether it is pre-paid, when the payment is made before the services are delivered, or post-paid, when the payment is made after the services are delivered.
- The active since (order available from) and until date shows end of the order availability (order expiration date)
- When was the last time it generated an invoice.

All this is considered in respect with the date and period that the billing process is running for. If active since date of order is after the billing until date those one time orders are not picked up by the billing process.

It is not very common to extend or implement your own order filter. An example of an extension is the class ***OrderFilterAnticipatedTask***. This class extends ***BasicOrderFilterTask***. This type considers another order property to allow orders to generate invoices for some periods in advance.

Invoice Filter

The invoice filter plays the same role in the billing process as the order's filter, but acting on invoices. The billing process needs to know **if an invoice should be carried over to a new invoice**. This object encapsulates the logic to make that decision.

Interface	**com.sapienter.jbilling.server.pluggableTask.InvoiceFilterTask**
boolean isApplicable (InvoiceDTO invoice, BillingProcessDTO billingProcess)	
Description: This method takes invoice object containing all invoice details and billing process object which contains all necessary details about the process. ***BasicInvoiceFilterTask*** implementation of ***isApplicable(...)*** method: This filter simply verifies that this invoice is being processed after its due date. Return true if the method should be included in the billing process run or false if the invoice should not be included in this billing process run.	
Argument: invoice—*InvoiceDTO*—the invoice in question billingProcess—*BillingProcessDTO*—billing process that is currently running *return*—**true** if the invoice should be included in the billing run, **false** otherwise	
Class structure: `public interface OrderFilterTask {`	

```
    boolean isApplicable(InvoiceDTO order,
            BillingProcessDTO process) throws TaskException;
}
```

BasicInvoiceFilterTask.isApplicable(...) method code:

```
public boolean isApplicable(InvoiceDTO invoice, BillingProcessDTO process)
throws TaskException {
        // we always delegate, even when is not yet overdue
        // this ensures the 'last invoice holds them all' policy
        return true;
}
```

The default implementation, ***BasicInvoiceFilterTask****,* returns *true* in all cases. Since the filter is only called for invoices that have a balance (they are not paid and balance is greater than zero), this will work fine when you want to follow the policy of having the last invoice to represent the total balance of a customer's account.

The other implementation is ***NoInvoiceFilterTask*** does the opposite, returns *false* in all cases. This is useful when your company never wants an invoice to get carried over to a new invoice.

You could write your own implementation if you need additional logic in the decision of carrying over an invoice. For example, if this should be done following some customer preference.

Order Period

This type of plug-in is involved in the billing process once the order has been already confirmed as one that will be generating an invoice. As we've seen earlier, this means that the order filter plug-in has given the green light about this order inclusion in the billing process.

We know then, that this order has to generate an invoice. What the billing process needs to know now is what period of time is that the new invoice will get out of this order for this particular billing process.

For example, there is a monthly order that has generated three invoices for the first three months of the year. When April comes along, and the billing process runs:

- First, the order filter is called. It will determine that this order is to be included in an invoice.
- Second, the order period is called. It will give the starting and ending date for the period to include. In our example that will be April 1st (inclusive) and May 1st (not inclusive) respectively.

In a nutshell, the job of this plug-in is to calculate and return the number of periods that will be included from a particular order into an invoices. The output of these types of plug-ins is two dates and a list of periods. The logic in this category of plug-ins is valid only for recurring orders, those that over their lifespans will generate many invoices. If the order is a one-time purchase,

the plug-in should return ***null*** for both dates and one period in the list with both dates set to null. This is how, later in the billing process, the invoice composition task knows which charges are for a period and which are a one time charge.

Here we should emphasise the importance of the calculated list of periods. This periods are represented by the class ***PeriodOfTime.*** In monthly scenarios we expect to calculate one period for a given order. However in more complicated scenarios, for example backdated orders, we could easily see more than one period calculated for the same order: one period for the missing period, and one for the current period.

Interface	**com.sapienter.jbilling.server.pluggableTask.OrderPeriodTask**
Date calculateStart(OrderDTO order)	
Description: Calculates the date that the invoice about to be generated is going to start cover from this order. This IS NOT the invoice date, since an invoice is composed by (potentially) several orders and other invoices. This method takes **OrderDTO** object (order) as a parameter	
Argument: order—*OrderDTO*—object with all order details	
Date calculateEnd(OrderDTO order, Date processDate, int maxPeriods, Date periodStart)	
Description: This method takes an order and calculates the end date that is going to be covered considering the starting date and the dates of this process. This method takes ***OrderDTO*** (order), *process date*, *max periods* and *period start* as a parameters. If the order has no periods within the billing process dates, no invoice will be generated for that order. The parameter *maxPeriods* applies **only** for the recurring orders. This parameter specifies how many periods of the purchase order would be included in the invoice. *periodStart* is connected to the start period of the billing process and orders to be included in the process.	
Argument: order—*OrderDTO*—object with all order details processDate—*Date*—billing process date maxPeriods—*int*—how many periods of the purchase order would be included in the invoice periodStart—*Date*—start period of the billing process and orders to be included in the process	
List<PeriodOfTime> getPeriods()	

Description: Returns a list of the calculated periods that should be included in the invoice for the given order.

Class structure:

```
public interface OrderPeriodTask {
   Date calculateStart (OrderDTO order) throws TaskException;

   Date calculateEnd (OrderDTO order, Date processDate, int maxPeriods,
           Date periodStart) throws TaskException;

   public List<PeriodOfTime> getPeriods();
}
```

plug-ins that implement "*OrderPeriodTask*"
com.sapienter.jbilling.server.pluggableTask.BasicOrderPeriodTask

Description: Calculates the start and end period to be included in an invoice. This is done by taking the billing process time span, the order period, the active since/until, etc**.** The billing process will include one period by default, unless it is manually changed. If there are backdated orders the system will include them in the next billing run.

Invoice Composition

Once the billing process has the orders and invoices to include in a new invoice, it has to create it. The main job here is to **create the invoice lines**, which have a description, price, quantity and total.

The invoice composition plug-in will do that, and the default implementation adds the date ranges if the line is coming from a recurring order.

Invoice composition (***InvoiceCompositionTask***) is a quite complex plug-in which requires additional information for its structure. Below is the explanation of the ***InvoiceCompositionTask*** and the use of its methods and arguments for the plug-in to work properly along with the default implementation plug-ins.

Interface	com.sapienter.jbilling.server.pluggableTask.InvoiceCompositionTask
void apply (NewInvoiceContext invoice, Integer userId)	

Description: Creating invoice lines for orders and invoices that are processed by the billing

process.

Argument:

invoice—NewInvoiceContext—complex object that delivers information about orders/periods included, about the current state of the invoice that we are creating etc.

userId—Integer—the user/customer for which we are generating invoice currently

Class: ***NewInvoiceContext***

`List<OrderContext> ordersContext`—primarily an input parameter that provides information about orders/periods that are to be included in the newly created invoice. The objects here are created based on the work on previous plug-ins such as ***OrderFilter*** and ***OrderPeriod***. For more information check the description about ***OrderContext*** class bellow.

`Set<InvoiceDTO> invoices`—primarily an input parameter that provides information about overdue invoices that can be included as carried invoices into the newly created invoice. The content of this set also depends on the result of the ***InvoiceFilterTask***.

`List<InvoiceLineDTO> resultLines`—This is primarily an output parameter. You are meant to store all the create invoices lines as part of this plug-in into this list. Upon completion of the plug-in the server will be using this list to create the actual invoice and store it in the database. Also you can use this list to reference already created invoice line further in the process.

`Integer entityId`—owning company

`Date billingDate`—billing process date

`TimePeriod dueDatePeriod`—period of time to pay the invoice before it becomes overdue

`Boolean dateIsRecurring`—true if the order date is recurring (monthly, weekly, semi-annually, etc.), else it is one time order.

Class: ***OrderContext—***provides contextual information about each order to be included in the invoice. This would include the order in question, periods to be included as calculated in ***OrderPeriodTask*** ….

`OrderDTO order`—order object that contains order details needed to construct an invoice

`List<PeriodOfTime> periods`—list of time periods to be included in the invoices for the order in question. These are calculated in the OrderPeriod task, so the actual number and content of this depends on which OrderPeriod task is configured in the system

`BigDecimal totalContribution`—could be used to count the total contribution that individual order is making towards the newly generated invoice. Example, an order can contribute for multiple periods into an invoice or it can include discounts. Knowing the exact total of the order contribution to an invoice could prove as useful information, especially in reseller scenarios where one need to know exactly how much the end customer was charged.

Class: ***OrderLineCtx (extends OrderLineDTO)***—provides contextual information about each order line from a specific order that needs to be included in the invoices as invoice lines. In most simple scenarios this object would be equal to the order line itself, but sometimes there can be multiple changes that are coming from different order changes, which are applied to the same order lines. In the latter case, a single order line that has more than one order change can result in more than one OrderLineCtx calculated for it, and because of this it can result in more that one invoice line. To summarize, the order line contexts object can be calculated either from the order line directly or from the order changes that are applied against that order line.

`Integer typeId`—the type of the order line i.e simple item, discount, tax …

`Integer itemId`—product associated with this order line

`Date date`—meant to represent the next billable date (from order line or order change)

Class: ***OrderChangeCtx (extends OrderChangeDTO)***—provides contextual information about each order change that is applied on specific order line. This information is mostly used during the calculation of the ***OrderLineCtx***. Note that order change provide more insights into how the order lines were created and also allow for much more control into how the invoice will look in case of more complicated billing scenarios.

Interface Class**: *InvoiceCompositionTask***

```
public interface InvoiceCompositionTask {
   public void apply (NewInvoiceContext invoice, Integer userId) throws
TaskException;
}
```

plug-ins that implement "*InvoiceCompositionTask*"

com.sapienter.jbilling.server.pluggableTask.CalculateDueDate

Description: This implementation sets the due date of the invoice. The due date is calculated by just adding the period of time to the invoice date. It uses *NewInvoiceContext* object to get *dueDatePeriod* and to set the due date of the invoice.

CalculateDueDate.apply(...) method code*:*

```
public void apply (NewInvoiceContext invoice, Integer userId) throws TaskException {
...
try {
```

```
    // add the period of time
    if (CalendarUtils.isSemiMonthlyPeriod(invoice.getDueDatePeriod().getUnitId())) {
        calendar.setTime(CalendarUtils.addSemiMonthyPeriod(calendar.getTime()));
    } else {
         calendar.add(MapPeriodToCalendar.map(invoice.getDueDatePeriod().getUnitId()),
invoice.getDueDatePeriod().getValue());
    }

    // set the due date
    invoice.setDueDate(calendar.getTime());

} catch (Exception e) {
    LOG.error("Unhandled exception calculating due date.", e);
    throw new TaskException(e);
}
}
```

com.sapienter.jbilling.server.pluggableTask.OrderChageBasedCompositionTask

Description: Copy all the lines on the orders and invoices to the new invoice, considering the periods involved for each order, but not the fractions of periods. It will not copy the lines that are taxes. The quantity and total of each line will be multiplied by the amount of periods.

Sometimes you might need some additional information in an invoice: some extra fields coming from the order or even from another system external to jBilling. If that is the case, you can do so as an invoice composition plug-in.

Order Processing: Totals and Taxes

This plug-in category is not called by the billing process, unlike the previous ones in this chapter. However, since orders play such a key role in the generation of invoices, it has been included among them.

The order processing plug-in category is expected to take a raw order straight from the GUI and complete any missing values, such as the order line totals and the grand total for the order.

The default implementation (***BasicLineTotalTask***) will go over the order lines, calculating each total mostly by multiplying quantity times price. It will also consider percentage items, taking first those that are not taxes, and calculating percentage taxes last (so they take into account all the previous items).

This plug-in category is a common source of custom plug-ins. This is usually done by doing a new implementation and chaining the new plug-in type through the configuration (the result is

several plug-ins of the same category, but each with a different processing order).

A good example is the VAT type (***GSTTaxTask***). It will take the order total and add a new line to represent the value added tax. Since it needs the order total, it would have to have a higher processing order than the ***BasicLineTotalTask***. As you can see, you will probably address your requirements by adding more types but keeping the default as the first in the chain.

Still, since this is a good place to add tax related logic and this changes so much from place to place, it is possible to use a completely new implementation and take the default type only as an example.

Chapter 6
Notification Plug-ins

CHAPTER 6: Notification Plug-ins

An important feature of jBilling is that it notifies customers of billing events, such as new invoices, payment results, etc. This is key to help automate the billing cycle. The typical way is to notify by email: it is free and widely accepted.

When a notification is needed, the system will check your plug-in configuration to see how that notification will be done. By default, the type configured will be ***BasicEmailNotificationTask***. This type sends an email to the customer.

There is another implementation, ***PaperInvoiceNotificationTask*** that generates a PDF version of an invoice. This is only used by the billing process for customers that have selected paper as an invoice delivery method.

Implementing new types of this category is relatively simple. The interface ***NotificationTask*** only has one method: deliver. You could create a new type to send notification via telephone with an IVR, or by fax for example.

For the most part, jBilling is unaware of how the notifications are delivered to the end customer. Just implementing a new type and configuring your account to use it would change jBilling's notification method.

Payment Gateway Down Alarm

Your integration with a payment gateway is a key area of your overall system, as it allows you to process payments in real-time. If a payment gateway is down, your business can suffer. Since you cannot process payments, you might not be able to offer your services to new customers or sell online.

jBilling is sitting between your business applications and the payment gateways. It will be most probably the only component interacting with the gateways. Thus, it could tell you if a gateway is down.

When a payment fails, the system will take a look at your plug-in configuration, find the plug-in that can handle the failure, and decide if to send an alarm email to the billing administrator. The default type is ***ProcessorEmailAlarmTask***.

There are two conditions for alarm to go off:

1. The gateway is not responding (unavailable).
2. The gateway has failed a number of payment requests in a row, within a period of time.

The first condition is simple and is is related to a network error. The second is to cover situations where the gateway server does respond, but with an error that states that the gateway is not available. If a gateway is failing all the payment requests, then it is not working as expected.

To avoid having an email for each payment request where a gateway is unavailable (if the gateway server goes down, it can take hours to be up and running again), this alarm can be configured to send only one email over a period of time.

The following parameters are needed to configure the behavior of the alarm; they are present as parameters to the plug-in:

failed_limit: number of payment requests that have to fail before the alarm goes off (see second condition above).

failed_time: amount of seconds where the number of failed requests have to happen.

time_between_alarms: number of seconds that have to pass in between emails reporting an unresponsive gateway.

email_address: This is the address where the alarm emails will be sent. This is an optional parameter. If not present, the email address for the company will be used (as defined when the company was created).

This default implementation is usually enough. You could create new ones to use a different type of notification method instead of emails, or to have different logic on when an alarm should go off.

Chapter 7
Interest/Penalty Plug-ins

CHAPTER 7: Interest/Penalty Plug-ins

As part of the (typically) daily batch process, there is an interest evaluation process. This is an independent process that is meant to run once a day.

This process will take all the invoices that are past their due date and call a plug-in to take action on them. That means that the actual logic to calculate and apply interest or penalties is encapsulated into a plug-in.

The default type is ***BasicPenaltyTask***. The ***BasicPenaltyTask*** is an extension of the ***InternalEventsTask***. This task can be configured with two different parameters. The first one is *item*. The value of the parameter has to be the ID of an item that represents the penalty. The item can have a flat price or a percentage price. A percentage price can be used to calculate interest. The second parameter is called *ageing_step* and it represents the user status ID which the customer needs to enter so the plug-in checks if interest or a penalty needs to be created for that customer.

The plug-in will create a new purchase order for this customer, which will have the specified item and the resulting amount. This new order is a one-time order, and it is meant to be included in the next invoice.

This way, the customer will see an additional line in their invoice with some interest charges. The invoice line will specify the invoice that was not paid on time, and the due date of that invoice.

This type is fairly simple, considering the complexity that charging interest and applying penalties can involve. Therefore, it is not uncommon to create custom types that takes more variables into account for the calculations, such as time, for example.

To implement your own, start by looking at the default. The contract for the category is trivial: you get an invoice ID as a parameter so the plug-in can analyze and take whatever action it wants. Still, the default will show you some considerations to be done, such as how to verify the outstanding balance of an invoice.

Chapter 8
Internal Events

CHAPTER 8: Internal Events

Internally, jBilling has an event processing mechanism. For those familiar with patterns, this is the *Observer* pattern. The basic idea is that when something happens (an event), instead of writing right there all the logic for the consequence of that event, we call another module with the event. It is that module that will take care of calling all those that have subscribed to the event, and take whatever action they want.

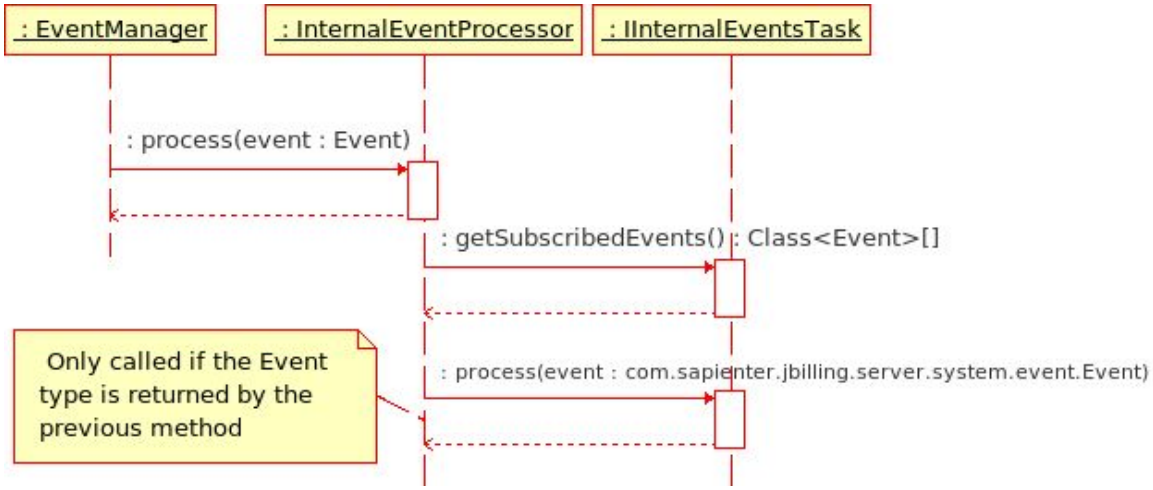
An example is when a payment fails. The payment processing module detects that a payment has failed. There might be a lot of things to do because of this: send an email, update the status of a customer, and even add some penalty fees. The payment module will simply store the result of the payment, that is its only concern. Then, it lets the event processing module know about a new event: a payment failed event. The payment module can then ignore any ramifications of what to do when a payment fails.

There can be many other modules that subscribe to this event, such as the notification module, to send an email. It is relatively easy to add a new subscriber to the event, or remove a subscriber if needed. This is a configuration task, rather than a development task.

Plug-ins for Internal Events

What is important about internal events is that you can subscribe to them to run your own logic. This, of course, is done with plug-ins.

Let's take a look to a simple sequence diagram that shows how your code can get called for any event happening in jBilling:



The main event processor of jBilling will always call a perennial subscriber for all internal events. This subscriber is called the internal event processor; the first thing it does is to look for any plug-ins present for the category 17. This is, any plug-ins that implement the interface ***IInternalEventsTask***.

If there are none, that is fine, these are not required plug-ins. If there are any plug-ins for this category, it will create an instance of each of them and query if the plug-in is interested in the event that is being processed. If so, the plug-in is called, passing the event as a parameter.

Creating Your Own Event Processing Plug-in

There are two steps to create a plug-in that process events:

1. Identify the event you want to process.
2. Write the plug-in

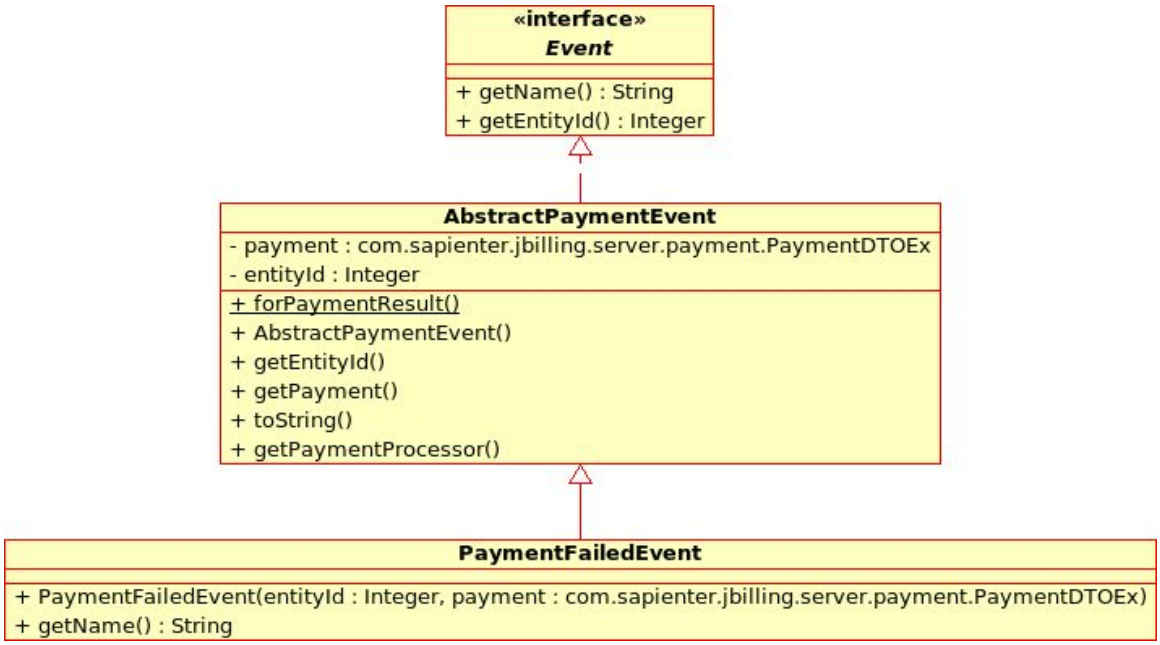
Events

To identify the event that you need to intercept, first you need to know how an event looks like in jBilling.

An event is just a class that carries the data of a real billing event. This class needs to implement a very simple interface:

```
public interface Event {
    public String getName();
    public Integer getEntityId();
}
```

The interface is mostly a way to group all the events in a single type. Here is a sample implementation:



For the most part, all we have here is an implementation of the ***Event*** interface that can hold a payment object (***PaymentDTOEx***). Any subscriber receiving this event knows that a payment failed, and from the payment class can find out all about the payment.

List of Events

When you are considering writing a internal event processor plug-in, it is because there is some business requirement that you need to address. Whether or not you can depends mostly on if there is an internal jBilling event that can help you.

jBilling did not start from scratch with an internal event design. This was added for the 1.0.7 release. Since then, more and more business logic is implemented using events, which means that more events are actually created (and made available to you to write plug-ins). Events were eventually made public by hooking them to plug-ins in version 1.1.0.

The list of events is still fairly short. More events are added with each release. The following list is accurate at the time of this writing, but likely incomplete by new additions. A good way to find out all the jBilling events is by finding all the implementations of the interface ***Event***. Any good IDE will provide this list easily.

ReservationCreatedEvent

Type: Diameter

Trigger: When an amount is reserved for a user in result of a successful Diameter call.

Current use: To be used in determining the dynamic balance of a user.

ReservationReleasedEvent

Type: Diameter

Trigger: When an amount is released for a user in result of a successful Diameter call.

NewInvoiceEvent

Type: Invoice

Trigger: When a new invoice is created for a user

InvoiceDeletedEvent

Type: Invoice

Trigger: When an invoice is deleted

ItemDeletedEvent

Type: Item

Trigger: When an item is deleted in the system.

ItemUpdatedEvent

Type: Item

Trigger: When an item is updated.

NewItemEvent

Type: Item

Trigger: When a new item is created.

NewPlanEvent

Type: Item

Trigger: When a new plan is created.

Current use: To be used to log new plan creation activity in a text file. Also, to communicate any new plan creation to SugarCRM via REST.

PlanDeletedEvent

Type: Item

Trigger: When a plan is deleted.

Current use: To be used to log new plan deletion activity in a text file. Also, to communicate any new plan deletion to SugarCRM via REST.

PlanUpdatedEvent

Type: Item

Trigger: When a plan is updated.

Current use: To be used to log new plan upgradation in a text file. Also, to communicate any new plan upgradation to SugarCRM via REST.

AssetCreatedEvent

Type: Item

Trigger: Event gets fired when an asset is created.

Current use: Event gets fired when an asset is created.

AssetDeletedEvent

Type: Item

Trigger: Event gets fired when an asset gets deleted.

Current use: Event gets fired when an asset is created.

AssetAddedToOrderEvent

Type: Item

Trigger: Event gets fired when an asset is added to an order.

Current use: Event gets fired when an asset is added to an order.

AssetUpdatedEvent

Type: Item

Trigger: Event gets fired when an asset gets updated.

Current use: Event gets fired when an asset gets updated.

NewActiveUntilEvent

Type: Order

Trigger: When an order has been updated and the order's 'active until' was changed.

Current use: To identify if an order is being canceled, and may be apply cancellation fees. Also, to change the subscription status of a customer, from active (recurring order is on-going, then it gets a new active until) to 'pending unsubscription'.

NewOrderEvent

Type: Order

Trigger: When a new order is created.

Current use: To identify when an order is created, and then update the dynamic balance of the user. Also used in ***PooledTarrifPlanTask*** to update the pooled items (quantities).

NewPriceEvent

Type: Order

Trigger: During the creation of an order, for each order line that has had its price modified.

Current use: To identify when the price of an order line is changed, and subsequently update the dynamic balance of the user.

NewQuantityEvent

Type: Order

Trigger: When an order line quantity is updated in an order, including lines added or deleted.

Current use: For the refund and cancellation fees plug-ins. Also used by provisioning plug-in.

NewStatusEvent

Type: Order

Trigger: When an order has changed status.

Current use: When an order goes to 'finished' status, the customer's subscriber status changes from 'Pending Unsubscription' to 'Unsubscribed'.

OrderAddedOnInvoiceEvent

Type: Order

Trigger: When an order is added to an invoice.

Current use: To identify when an order is added to an invoice, and subsequently update the dynamic balance of the user.

OrderDeletedEvent

Type: Order

Trigger: When an order is deleted.

Current use: To identify when an order is deleted, and subsequently update the dynamic balance of the user.

OrderPreAuthorizedEvent

Type: Order

Trigger: When an order has been successfully pre-authorized.

Current use: To identify when an order has been successfully pre-authorized, and then provision it for partner customers to external system.

OrderToInvoiceEvent

Type: Order

Trigger: Before an order is added to an invoice. Allows for the order to be modified before going on an invoice.

Current use: To identify when an order is about to be added to an invoice, and then apply tax on it if required.

PeriodCancelledEvent

Type: Order

Trigger: When there is a new active until for the order, that is earlier than the previous one.

Current use: There is a plug-in that evaluates this event through rules (see the rules chapter).

The final outcome can be a new order with cancellation fees, as a penalty for an early cancellation of a contract.

ProcessTaxLineOnInvoiceEvent

Type: Order

Trigger: When the billing process is creating an invoice and it is required to process taxes and penalties.

Current use: To identify when an invoice line has been taxed, and subsequently update the dynamic balance of the user.

OrderChangeAppliedEvent

Type: Order

Trigger: This event is triggered AFTER an OrderChange is successfully applied to an order.

Current use: Allows for the order to be modified before saving new changes in order.

PaymentFailedEvent

Type: Payment

Trigger: A payment processing plug-in returns 'failure' as the result of payment request to a payment gateway.

Current use: To take the customer's subscriber status to 'Pending Expiration'.

PaymentProcessorUnavailableEvent

Type: Payment

Trigger: A payment processing plug-in failed to connect to a payment gateway (or a request timed out).

Current use: To evaluate an alarm due to the payment gateway being down.

PaymentSuccessfulEvent

Type: Payment

Trigger: A payment processing plug-in returns 'success' as the result of payment request to a payment gateway.

Current use: To set the customer's subscriber status back to 'active'.

PaymentLinkedToInvoiceEvent

Type: Payment

Trigger: When a payment is linked to an existing invoice.

Current use: To attach a payment to an existing invoice.

PaymentUnlinkedFromInvoiceEvent

Type: Payment

Trigger: When a payment is unlinked from a linked invoice.

Current use: To remove link of an invoice from a payment.

EndProcessPaymentEvent

Type: Billing process

Trigger: All the payment requests for the current billing process have been posted.

Current use: Since the billing process finishes much earlier than the payment processing, it is necessary to signal the end of payment processing to update the 'payments end time' column of the billing process record.

PaymentDeletedEvent

Type: Payment

Trigger: When a payment is deleted.

Current use: To identify when a payment is deleted, and subsequently update the dynamic balance of the user.

ProcessPaymentEvent

Type: Payment

Trigger: The billing process needs a payment to be processed.

Current use: This event is processed by the event manager asynchronously. It allows the detachment of the billing process from a time consuming task that relies on third party system: credit card processing.

AboutToGenerateInvoices

Type: Process

Trigger: When the billing process is about to generate new invoices

Current use: To be used to calculate penalty for overdue invoices.

AgeingProcessCompleteEvent

Type: Process

Trigger: When the ageing process is completed.

AgeingProcessStartEvent

Type: Process

Trigger: When the ageing process is completed.

BeforeInvoiceDeleteEvent

Type: Process

Trigger: Before an invoice is deleted.

Current use: To identify when an invoice is being deleted and send invoice deletion request to suretax.

InvoicesGeneratedEvent

Type: Process

Trigger: After invoices are generated, either by the billing process or by invoice generating API calls.

Current use: To identify negative invoices and then fix them as credit payments using the ***ApplyNegativeInvoiceToPaymentTask***.

NewUserStatusEvent

Type: User

Trigger: When a user's status is changed, either through the aging process or manually.

Current use: Plug-in for blacklisting users that become suspended or higher.

NoNewInvoiceEvent

Type: Billing process.

Trigger: During the billing process, if a user did not get any invoices.

Current use: To handle transitions of the customer's subscription status when it is 'Pending Unsubscription'.

SubscriptionActiveEvent

Type: Provisioning process

Trigger: During the provisioning process when an order's *activeSince* date becomes earlier than or equal to the current date (or null) and has order lines with 'inactive' provisioning statuses.

Also triggered when an order is created with an *activeSince* date earlier than or equal to the current date (or null).

Current use: External provisioning of services.

SubscriptionInactiveEvent

Type: Provisioning process

Trigger: During the provisioning process when an order's *activeUntil* date becomes earlier than or equal to the current date and has order lines with 'active' provisioning statuses.

Current use: External provisioning of services.

AchDeleteEvent

Type: User

Trigger: Delete the existing ACH Payment information for the user

Current use: Delete user's ACH payment details that may be maintained with an external payment gateway so as to avoid it being used for payment from the gateway.

AchUpdateEvent

Type: User

Trigger: Add new ACH Payment information for the user or update an existing one

Current use: Plug-ins of type ***IExternalCreditCardStorage*** that provide integration with 3rd party secure Payment Gateway to store sensitive financial information externally. On this event, user's ACH payment details can be stored externally and obscured from the local database.

AgeingNotificationEvent

Type: User

Trigger: During the ageing notification process.

Current use: Provides a mapping between the ageing step (user status Id) and a custom notification message id

DynamicBalanceChangeEvent

Type: User

Trigger: When the dynamic balance changes of a customer.

Current use: To identify when the dynamic balance of a user changes, and if needed invoke the auto-recharge functionality. Also, to be used to check whether the balance threshold has been reached and if so, send out a notification.

NewContactEvent

Type: User

Trigger: When the contact information is created or updated.

Current use: To identify when the contact information is created or updated, and then invoke a remote web-service when a customer's contact information is changed. This plug-in fetches the current weather from the webservicex.com Global Weather service and updates the customer notes with the result (this was made as an example on how to use/develop plug-ins).

NewCreditCardEvent

Type: User

Trigger: Add new Credit Card details for the user or update existing details

Current use: Plug-ins of type ***IExternalCreditCardStorage*** that provide integration with 3rd party secure Payment Gateway to store sensitive financial information externally. On this event, user's credit card can be stored externally and obscured from the local database.

UsagePoolConsumptionFeeChargingEvent

Type: UsagePool

Trigger: This is a new Event that will be fired when it is defined on the FUP as a consumption action for a certain consumption

Current use: This is currently used for usage pool consumption fee charging.

CustomerPlanSubscriptionEvent

Type: UsagePool

Trigger: When a customer subscribes to a plan.

Current use: This is a new Event object that represents the successful subscription of a customer to a plan.

CustomerPlanUnsubscriptionEvent

Type: UsagePool

Trigger: When a customer unsubscribes a subscribed plan.

Current use: This is a new Event object that represents event of customer unsubscribing from the plan, this is either done by deleting the plan order, or by setting active until date on the plan order.

CustomerUsagePoolConsumptionEvent

Type: UsagePool

Trigger: This is a new Event that will be fired every time there is a change in customer usage pool quantity.

Current use: Used to keep track of the customer's usage pool activities.

UsagePoolConsumptionNotificationEvent

Type: UsagePool

Trigger: This is a new Event that will be fired when it is defined as a consumption action on the FUP for a certain consumption

Current use: Used to perform some custom action when usage of customer reaches a predefined limit.

CommandStatusUpdateEvent

Type: Provisioning

Trigger: When there is change in the status.

Current use: Used to keep a track of each status change.

OrderChangeStatusTransitionEvent

Type: User

Trigger: When there is transition is order change status

Current use: Used to keep track of previous and new order change status.

The above list only tells you the available events and when they are triggered. But, how do you use an event? Keep in mind that an event is only a mean for transporting information—a message to you. It is not meant to do anything on its own. The business logic related to the data in the event should be placed in a plug-in.

Implementing Your Own Plug-in

Like any plug-in in jBilling, a plug-in to process internal events has to extend the abstract class ***PluggableTask*** and implement an interface. In this case, the interface is ***IInternalEventsTask***:

```
public interface IInternalEventsTask {
	public void process(Event event) throws PluggableTaskException;
	public Class<Event>[] getSubscribedEvents();
```

```
}
```

The method *getSubscribedEvents()* should return a array of the events that you want your plug-in to be called for. This is just a way of subscribing to those events. If the event in question is part of this list, then *process()* is called. In other words, *getSubscribedEvents()* will be called for every event, while *process()* only for those events that you actually want to get called.

Example: "Hello Payment"

Let's write a plug-in that writes to the log file every time a payment is processed:

```
public class HelloPaymentTask extends PluggableTask implements
IInternalEventsTask {


    private static final Class<Event> events[] = new Class[]                    {
PaymentFailedEvent.class, PaymentSuccessfulEvent.class };


    private static final Logger LOG =
Logger.getLogger(HelloPaymentTask.class);


    public Class<Event>[] getSubscribedEvents() {
        return events;
    }
    public void process(Event event) throws PluggableTaskException {
        if (event instanceof PaymentFailedEvent) {
            PaymentFailedEvent failed = (PaymentFailedEvent) event;
            LOG.debug("The payment " + failed.getPayment() +
                    " failed");
        } else if (event instanceof PaymentSuccessfulEvent) {
            PaymentSuccessfulEvent success =
                    (PaymentSuccessfulEvent) event;
                  LOG.debug("The payment " + failed.getPayment() +
                          " succeeded");
        } else {
            throw new PluggableTaskException("Cannot process event "
                     + event);
```

```
        }
    }
}
```

Our plug-in is subscribed to two events, one for each payment outcome (we are leaving processor unavailable out since if this event happens then the payment could not be processed at all).

When the process is called, we have to verify for what event we've been called. After that, we'll have an instance of the event with all the information need for any action we want to take. In this case, the key piece of data is the payment object. Our plug-in only prints the payment to the log file.

Configuration

We have the plug-in, and now we need to let jBilling know about it with a couple of configuration steps. First, we need to register the plug-in as a new type:

```
insert into pluggable_task_type values(
  50, 17, 'com.sapienter.jBilling.server.payment.tasks.HelloPaymentTask', 0);
```

Then, from the jBilling GUI, click on 'Configuration,' then on 'Plug-ins,' to add your new plug-in.