www.jBilling.com

The Open Source Enterprise Billing System

Extension Guide

Copyright

Author

Emiliano Conde and others

Revision number and software version

This revision is 3.0.0, based on **jBilling** 2.2.0

Table of Contents

Chapter 1
Architecture

An overview of jBilling's design

The jBilling engine

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 try to 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 I 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 shouldn't need to read them all in every case. It all depends on what is that you are planning to do. It is intended to be more of a reference that goes a little bit more in details over the key internals of the system.

After that, you might have some questions, or would like to bounce some of your ideas off someone else. Send an email to the users list of find me in the forums!

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

Architecture Overview

A tiered approach

Overview

The most significant characteristic of jBilling's architecture is the 3 tiers 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.
- Server: It is the holder of the business logic and the only tier that talks with two other tiers: client and database.
- Database: A RDBMS engine that holds all the data. It only holds data, there are no store procedures or any other kind of code, let alone business logic here. Only the server tier gets to access the database.

I picked very standard names to describe these tiers, but this could lead to confusion so I better expand on this. By client I don't mean 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 that 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, I put the components

responsible of how the data is accessed in the database tier, although they are deployed in the application server.

Client tier

The main factor in this tier is Struts (http://struts.apache.org/). This is an implementation of the model-view-controller design pattern for web-based applications written in Java. So yes, our users access the system using a web browser, which connects to a web server where our JSPs are deployed. All the requests are handled by the Struts controller, then forwarded to Actions that eventually call the server to get something done.

Struts also help us with internationalization (i18n), validation, page layout (tiles) among other things. We've got to be grateful to the Jakarta folks!

The client tier, or GUI, is old and limited in functionality. Most of the effort in the past years went to improve the server and expose that functionality through an API. In is very common to deploy jBilling as a server only component, integrated with other systems that provide their own GUIs to interact with jBilling.

Choosing Struts 1 in 2002 was a good idea, it was the latest and greatest. And open source. Since then, AJAX has arrived and endless number of MVC implementations and frameworks (perhaps too many). Doing a complete rewrite of jBilling's GUI, using a modern framework is already in the design stages.

Server tier

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 initially, jBilling started as an EJB application that run 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

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.

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 pretty powerful and open source. It is quite easy to run jBilling in any other engine that supports ANSI SQL.

There are a few ways to access the database: through Hibernate persisted classes, Hibernate HQL queries, Hibernate criteria queries, and through JDBC calls. Hibernate is the preferred way to access the database. I don't see any need to call JDBC directly. Still, 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. In many cases, I had to manually write SQL to overcome this problem.

The preferred query method is Hibernate criteria, but only when the query is not very complex. Then, use HQL. I know that there is an endless debate on this point. This is a good example where there's no point debating... it is just a preference, like coffee or chocolate milk.

Business Rules Plug-ins

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 belong is also a key factor: a phone company is more likely to bill its customer like its competition does, not like a golf club charges its members. But still, business rules can change a lot from company to company just because they prefer to do it differently. Add to that the fact that all these rules change constantly.

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 are 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 design as Java interfaces. jBilling only knows about these interfaces, it doesn't know about the actual implementations. So when it tries to get on-line authorization for a credit card payment, it doesn't 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 any case, you are extending jBilling to fit your needs without modifying jBilling itself.

Rules engine integration

Plug-ins are great, but you don't want to have to write Java code every time a new promotion, discount or plan changes. There is another level of business rules that need to be expressed in a different way.

Writing fine grained business rules in Java is just silly. Worse is to do it in XML. There are already languages and frameworks to deal with this. We picked Drools as the only one that fits both the maturity/functionality and the open-source bills.

Thus, many major modules in jBilling like the mediation or rating rely heavily on external business rules to know how to do things. In other cases, you have many implementations of the same plug-in type, some based on external rules and some that are just plain Java.

It is important to note that jBilling itself, the core, does not use rules and holds no dependencies with Drools. All the interaction with the rules engine is done through plug-ins.

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 there are definitely grouped by their roles. You can tell what kind of type a class just by its name. By going over the major groups I believe you'll have an idea of how the system was design. Then, I'll present a sequence diagram to illustrate an example.

Actions

This client-tier classes extend Struts 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: *name*`Action`

Example: `com.sapienter.jBilling.client.payment.MaintainAction`

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 will 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: *name*`BL`

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

Let's take a look to a simplified version of this class. Note the association to `PaymentDTO` that represents one persisted bean, thus, one row in the database.

Let's see an example in a context. The user clicks on a payment to see its content. After the click, you know the ID of the payment. So you can do just 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 an through 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: It 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, this way implementing the session facade design pattern. They shouldn't do much more than forwarding the calls, although some times 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 who starts a transaction for us. 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 exiting 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.

Having been an Oracle DBA in a previous life (got certified and everything), I keep a close eye on how the database is being accessed to avoid performance problems down the road.

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

A common complain is the name of these classes. Why DTO? Specially considering that DTOs is a naming pattern that is used for other purposes. It is a poor choice of names, I admit. In my defense I can say that they are transferring data (from the DB to the application) and I do like having some kind of name for a class that, if I modify, I will be modifying the database. I like better PaymentDTO than just Payment.

Class name: *name*`DTO`

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

Processing flow

Let 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 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 Struts Action class, in this case `MaintainAction`. 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 call 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 Action ends by forwarding the user to the payment view page. This JSP knows how to display a payment based on the DTO bean present in the session.

Chapter 2

Reports templates

Extracting real-time data

Introduction

Behind every jBilling 3.0 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 do need to be familiar with the database tables of the jBilling schema and SQL in order to write a new report. Last but not least, you need to be familiar with Jasper Report JRXML files and preferably, the Jasper iReport design tool.

What is a report?

As mentioned before, a report is a Jasper Report file with a query that jBilling loads and runs against the database to produce a readable report. In-order 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, report name and the file name 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 sub-folder 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 sub-folder.

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

Lets 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 identifer 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 file name 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 will be 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 sub-folder 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]”/>
      </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 (i18n)

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 youll 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 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]”/>
      </g:applyLayout>
</div>
```

Add the report name to the messages.properties bundle:

```
total_payments=Total Payments
```

Chapter 3

Business Rules Plug-ins

The key to extending jBilling

Why 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: industry, geographical location, etc. still companies decide to work differently.

jBilling tries to face 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 can be 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 server multiple companies, and each company doesn't 'know' about the other ones. We can have one instance running for a company in Italy and another one in Canada. When the billing process runs, the right plug-in is plugged to calculate the taxes, which are very different between Italy and Canada.

The business rules plug-in architecture

There are many places that need their business logic encapsulated as a plug-in. Each of this 'areas' are represented as a 'plug-in category'. Then each category maps to an 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.

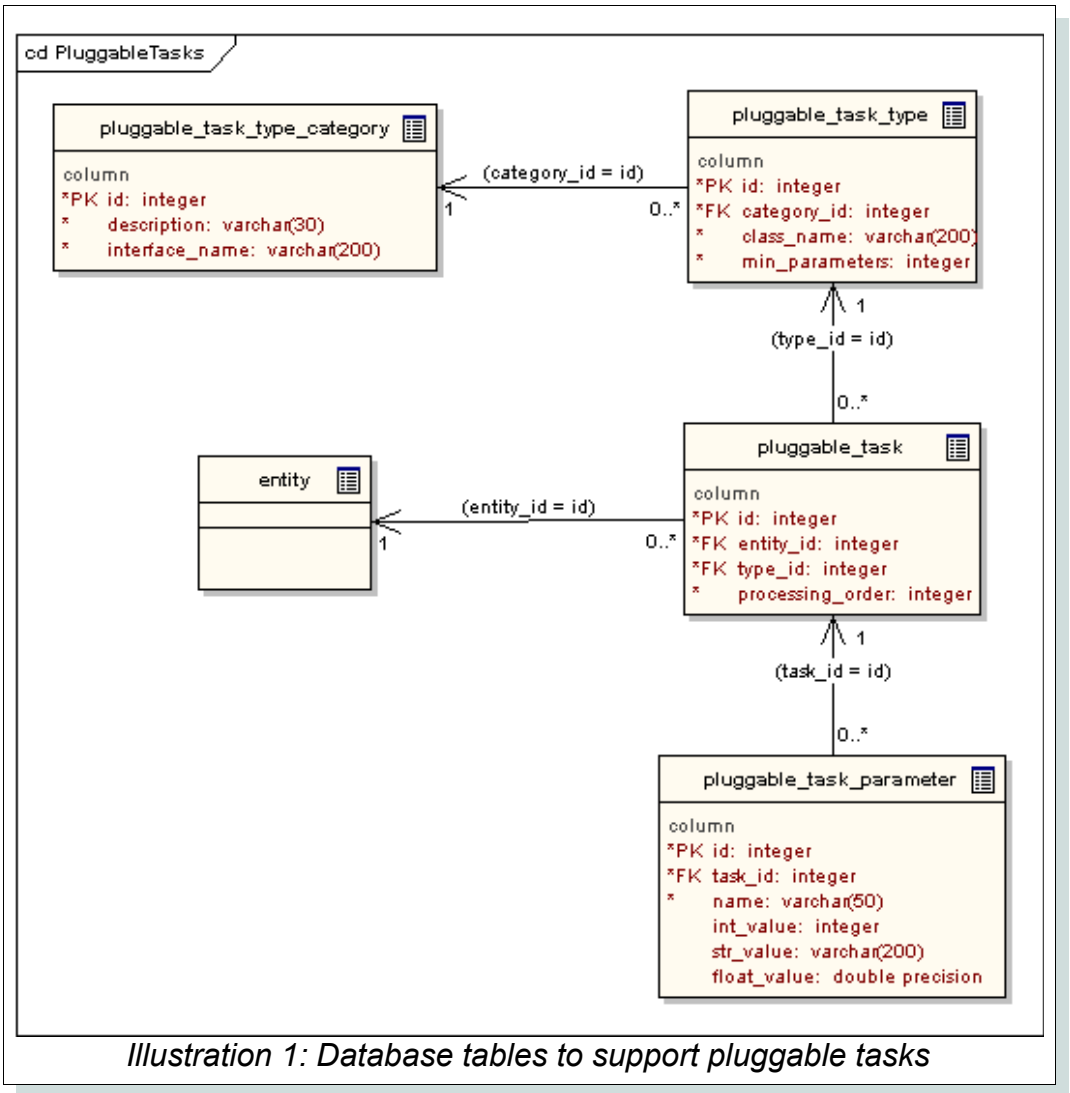


Illustration 1: Database tables to support pluggable tasks

Plug-ins are named in the code 'pluggable tasks'. Let's take a look to how the configuration data of the plug-in engine is represented as tables:

From the previous diagram, we can make some statements:

- 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.

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

- 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 like the user name and password to use the payment gateway goes. With it, we can give company red its own credentials to use the big gateway, and the same thing goes for blue.

How does it work

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 on, 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 the same can be scheduled as a Quartz job. At the time of system startup, these plug-ins or Pluggable Tasks that are already configured, are pulled from the database and scheduled as a Quartz Job depending on each plug-in's parameter values.

Therefore, today, the jBilling system has **three varieties of plug-ins**. There are also called as 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

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 like 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 the jBilling system that extend from 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 designate 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 a "**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 Plugin 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 `BillingProcessTask` is also an example of **`AbstractBackwardSimpleScheduledTask.`**

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 was 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	Does 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	22
Name	Scheduled Tasks
Interface	com.sapienter.jbilling.server.process.task.IScheduledTask
Description	Plug-ins of this type are scheduled as Quartz Job 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 a AbstractCronTask or an AbstractSimpleScheduledTask.

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 get 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	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'd be by using the default filter.

Category	3
Name	Dummy 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	Returns always false, which makes **jBilling** to never carry over an invoice into another newer invoice.

Category	4
Name	Invoice due date

Class	com.sapienter.jBilling.server.pluggableTask.CalculateDueDate
Description	A very simple implementation that sets the due date of the invoice. The due date is calculated by just 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	Invoice composition task with pro-rating (day as fraction)
Class	com.sapienter.jBilling.server.process.task.DailyProRateCompositionTask
Description	When creating an invoice from an order, this plug-in will pro-rate any fraction of a period taking a day as the smallest billable unit.

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'd be by using the default task.

Category	5
Name	Order periods calculator with pro-rating
Class	com.sapienter.jBilling.server.process.task.ProRateOrderPeriodTask
Description	This plug-in takes into consideration the field 'cycle starts' of orders to calculate fractional order periods.

Category	6
Name	Payment process for the Intraanuity 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 & 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*” chapter from 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	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, given priority to credit card. In other words, it will return the credit car 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 exactly the same as the standard payment information task, the only difference is 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	9
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	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, considering its present status and a state machine.

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	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	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	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	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	22
Name	Mediation Process Task
Class	com.sapienter.jbilling.server.mediation.task.MediationProcessTask
Description	A scheduled task to execute the Mediation Process.

Category	22
Name	Billing Process Task
Class	com.sapienter.jbilling.server.billing.task.BillingProcessTask
Description	A scheduled task to execute the Billing Process.

Creating your own plug-ins

When the default types do not meet your requirements, you will need to 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.

Eventually, 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 earlier, your new plug-in will be implementing an existing **jBilling** interface, or extending one of the existing types.

*Take advantage of the fact that **jBilling** is open source, all the source code is there for you to study!*

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 'System', 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.

*In is a 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:
    i. `AbstractCronTask`
    ii. `AbstractSimpleScheduledTask`
    iii. `AbstractBackwardSimpleScheduledTask`
2. Provide customized implementation to the methods
    i. Method getTaskName() - This method returns a string to identify the task by a name
    ii. 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
    iii. Method getTrigger() - Depending on the type of the parameters used for execution, the Quarts 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 seconds
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.
    i. 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
    ii. 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

Integrating with payment gateways

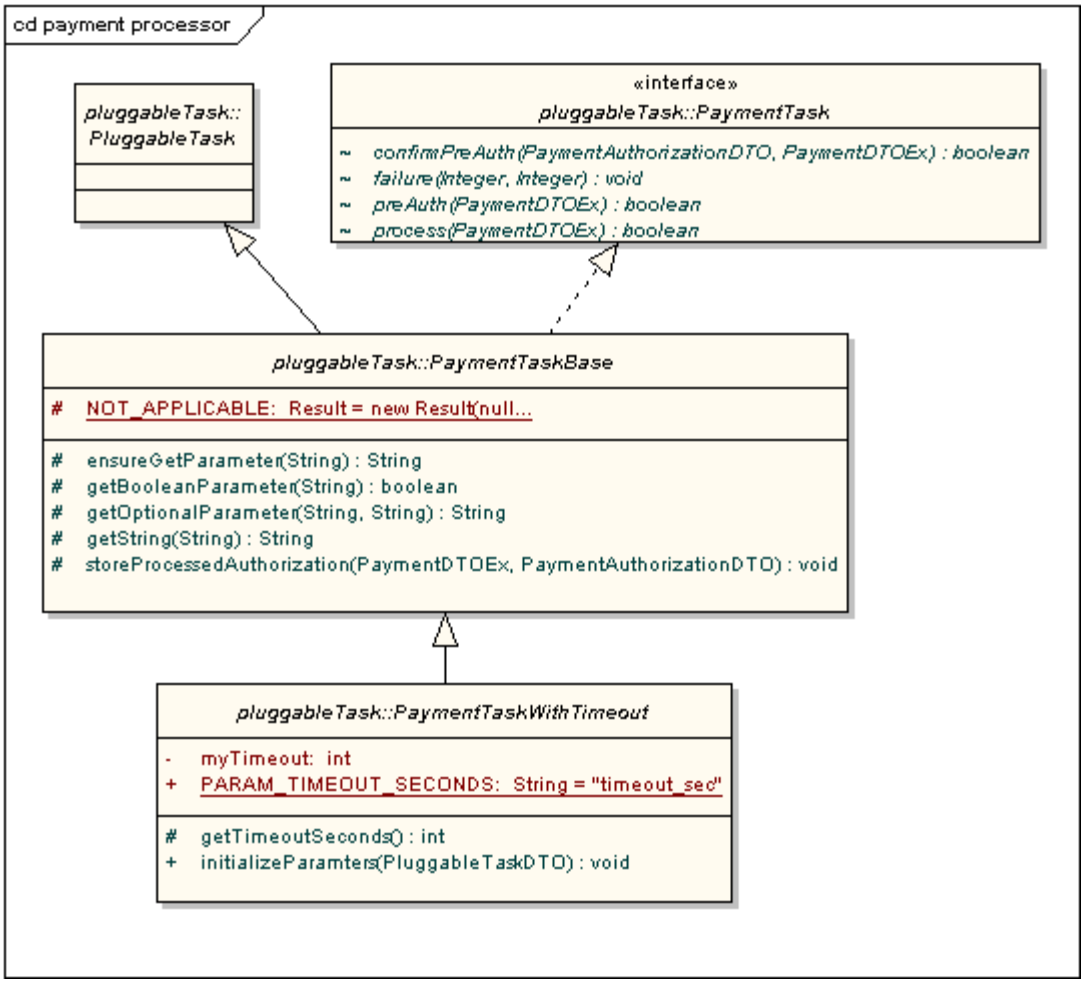
Introduction

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 fail-over 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 be extending `PluggableTask` and implementing 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) classes 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 else 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.

The main method for you to code is `process`, which takes a `PaymentDTOEx` object as a parameter. This way, the processor is not necessary 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, for result of success of failure, the return is false. If the communication with the payment processor fails (server down, timeout, etc), return true.

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 `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 can not 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 time out should be a parameter of the plug-in (see next point).

- Use parameters: Do not hard code parameters like the user name 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 id updated. Make a call to `setResutId` with the constants `Constants.RESULT_OK, Constants.RESULT_FAIL, 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.setResultId( ... ); // 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 finish 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 'System' → 'Plug-ins'. You will need to remove any plug-in that belongs to category 6. By default, that would be the '`PaymentFakeTask`'.

Then, create a new plug-in entry and select you class from the drop-down menu (it shows up now because of the previous step). Click on Submit so the changes are saved. You most probably will 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 just keep following 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 to **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 short 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 as the flag 'Use for automated payments'. It does filter 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 wide spread usage of credit cards as payment methods makes this plug-in category an unlikely candidate for custom implementations. Example requirements that would lead to one are: 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.

Starting on **jBilling** 1.0.8, 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 you 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? 2, 5 or more? Gateways can restrict the number of concurrent requests, and most probably will do so. 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 (and even the same) payment 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 gateways: **jBilling** let's you configure your payment processing in a way that you let 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/jbilling-jms.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>
```

This is a configuration file from the Spring framework, you will 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 means 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).

Imagine that the payment requests are in a queue. Each of them have a series of information fields needed for the payment to happen: the amount, user id, invoice id, etc... then 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.

A request will never be processed by more than one bean, but many beans can be processing (different) requests at the same time.

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 to process that number of payment simultaneously, you configure this scenario very easily: add the new beans with just a name change and keeping 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

The billing process is the module that more 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 where 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 key billing questions are answered by independent classes, rather than be 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 can't understand these plug-ins if you don't have a clear picture of the sequence of events happening in the billing process.

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**. The default implementation, `BasicOrderFilterTask`, considers the properties of the order: if it is pre or post paid, the active since and until, when was the last time it generated an invoice. All this in relationship with the date and period that the billing process is running for.

This task might change the status of an order if the situation calls from it. For example, if the active until has be reached and this is the last invoice that the order will generate.

This is not very common to extend or implement your own order filter. An example of an extension is the class `OrderFilterAnticipatedTask`. 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 filer, 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..

The default implementation, `BasicInvoiceFilterTask`, blindly returns 'true' in all cases. Since the filter is only called for invoices that have a balance (they are not paid and balance is grater than zero), this will wok 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`, which does the opposite: blindly returns 'false' in all cases. This is useful when your company never wants an invoice to get carried over to a newer 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.

Invoice composition

Once the billing process has the orders and invoices to include in a new invoice, it just 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 just that, and the default implementation adds the date ranges if the line is coming from a recurring order.

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 period

This type of plug-ins are involved in the billing process once the order has been already confirmed as one that will be generating and 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.*

Let's see an 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 filer 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.

The output of this plug-in category is two dates. It is assumed that the order in question has to generate an invoice. The logic in this category of plug-ins is valuable 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.

The default implementation is `BasicOrderPeriodTask`, and it is rarely needed to extend or modify it.

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 very much possible to use a complete new implementation and take the default type only as an example.

Chapter 6
Notification plug-ins

Notifications

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

When an 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 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 can represent a key area of your overall system, it allows you to process payments in real-time. If a payment gateway is down, your business can suffer. Since you can not process payments, you might not be able to offer your services to new customer or sell on-line.

jBilling is sitting in 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 gateways is down.

When a payment fails, the system will take a look to your plug-in configuration and 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: the gateway is not responding (unavailable), or 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 an 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 plug-ins

Interest and penalties

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 let it take action on them. That means that the actual logic to calculate an apply interests or penalties is encapsulated into a plug-in.

The default type is `BasicPenaltyTask`. This task will take one parameter: 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 interests.

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 simply see an additional line in her 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 taking a look to the default. All in all, 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

Introduction

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 (let's call this 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, even add some penalty fees. The payment module will simply store the result of the payment, that is its concern. Then, it lets the even processing module know about a new event: a payment failed event. That is all, the payment module can then ignore any ramifications of what to do when a payment fails.

There will be many other modules that have subscribed to this event: may be the notification module, to send an email, may be many others. It will be relatively easy to add a new subscriber to the event later on, or remove a subscriber if needed. We have turned this into a configuration task, rather than a development task.

Plug-ins for internal events

What it's actually 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 simply sequence diagram that shows how your code can get called for any event happening in jBilling:

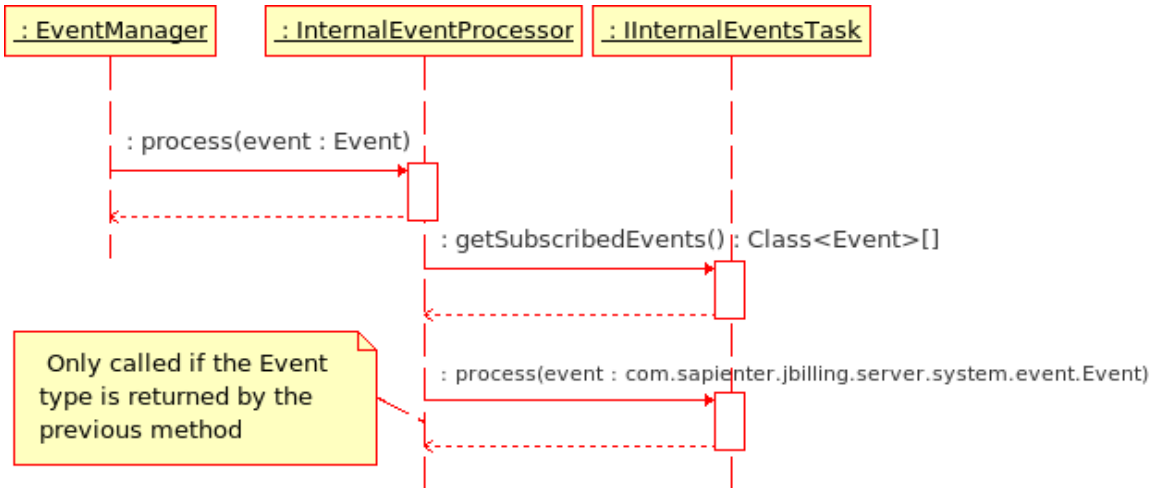


Illustration 2: The sequence of calls that get your plug-in called

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 is 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 an query if the plug-in is interested on the event that is being processed. If so, the plug-in is called passing the event as a parameter.

Universal events-to-rules plug-in

If you wish to run business rules in response to events, you can use the existing `InternalEventsRulesTask` plug-in for simple cases, instead of creating your own rules-based plug-in from scratch. See chapter 8 for more information on rules and the `InternalEventsRulesTask`.

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:

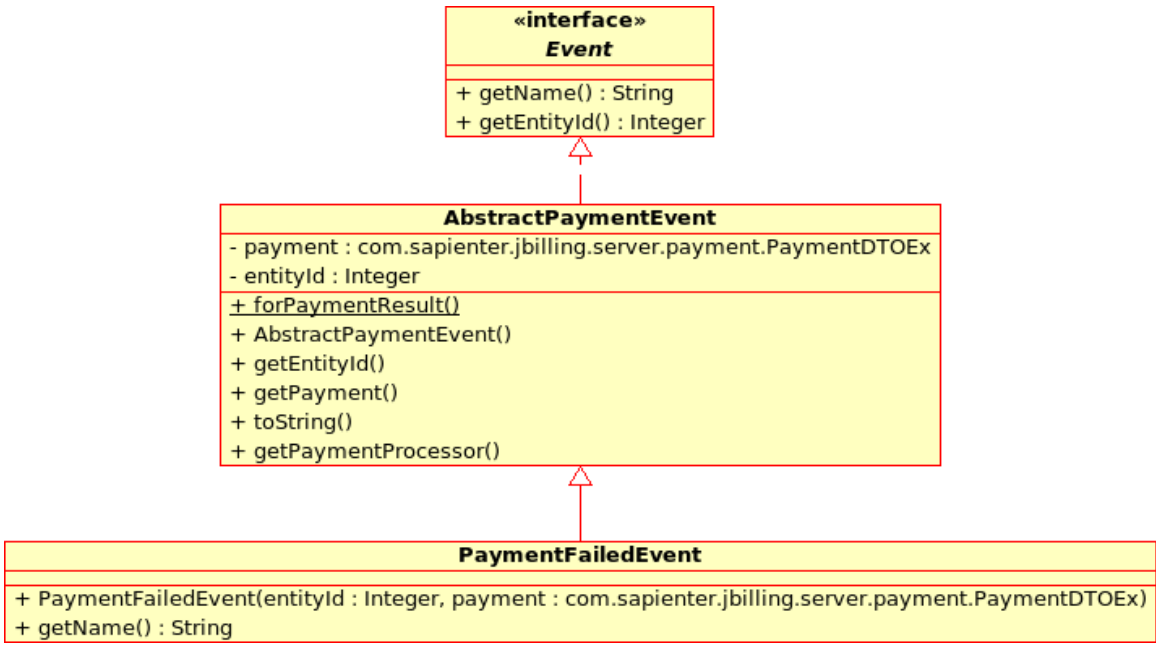
```
public interface Event {

    public String getName();

    public Integer getEntityId();

}
```

As you can see, the interface is mostly a way to group all the events in a single type. Let's see an example 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 is because there is some business requirement that you need to address. If you can do so or not will depend mostly 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 are 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. Also, more events are added continuously. The following list is accurate at the time of this writing, but most probably already 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.

`NewActiveUntilEvent`

Type: Orders

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'.

`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'.

`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.

`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'.

`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.

`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.

`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'.

`NewQuantityEvent`

Type: Order

Trigger: When an order line's 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.

`OrderToInvoiceEvent`

Type: Order

Trigger: When an order is added to an invoice.

Current use:

`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.

`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.

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.

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.

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

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 and 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 and 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("Cant not process event "
                   + event);
        }
    }
}
```

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

We processed 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, we just need to let jBilling now 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 'System' then on 'Plug-ins' to add your new plug-in.

Chapter 9

Rules and BRMS

Your business rules in action

Extending through rules

Introduction

This section is dedicated to extending jBilling by adding new business logic through a rule-based engine and BRMS (business rules management system). It continues on the on-line guide called 'Getting Started – BRMS'. If you have not read that guide yet, do so before continuing. You can find it here.

An important goal of the 1.1.0 release was to add substantial more flexibility to jBilling. Flexibility is key for a billing system, because billing is so tied to the way a company runs its business (business rules).

jBilling integrates with a rules engine to achieve this leap in business rules flexibility. All this chapter is dedicated to 'rules', so it is best if you are familiar with rules, rules engines and BRMS. The accepted 'birth' of rules systems came with the design of the 'RETE algorithm' by Dr. Charles L Forgy in 1974. This is basically a pattern matching algorithm that simplifies writing business logic and executes very efficiently. To learn more about RETE, read this article.

Instead of having hard wired business rules in the core of the billing system, you can neatly write business rules in a more natural language (rather than in programming language like Java). You then store the rules, so it easy to manage them. With some configuration of the right plug-ins, jBilling will go to that rules storage and execute your rules.

Drools

The rules-engine we will be using is Drools, also known as JBoss Rules. This is an open source implementation of the RETE algorithm, but it doesn't stop there. It comes with a full BRMS, you can use a GUI to create edit and manage your rules.

Drool is a feature rich, complex product. We will not try to re-write its documentation here. Instead, we are going to focus on how Drools and jBilling interact with each other. You can find Drools official documentation here.

The most important Drools features, from the jBilling perspective, are:

- Robust, proven implementation of the RETE algorithm.
- BRMS
- Ability to express rules in different ways: technical rules (plain text) and business rules (using a GUI with drop down values) among others.
- Supports the creation of a Domain Specific Language (DSL), so we can write rules using something closer to a natural language, rather than having to deal with objects, attributes and methods.

Rule based plug-ins

All the interaction between jBilling and rules happens through plug-ins. In fact, the base class that all plug-ins have to extend (`PluggableTask`), has been added support for rules so any plug-in can now ask for and process rules.

Four useful 'rules based' plug-ins are:

RulesItemManager: This is a rule-based implementation of a new plug-in type: `IitemPurchaseManager`. The default implementation is `BasicItemManager`. When using this one (the default), jBilling behaves just like it used to before 1.1.0. When the `RulesItemManager` is configured, you are enabling rules for item relationships. More on this plug-in later.

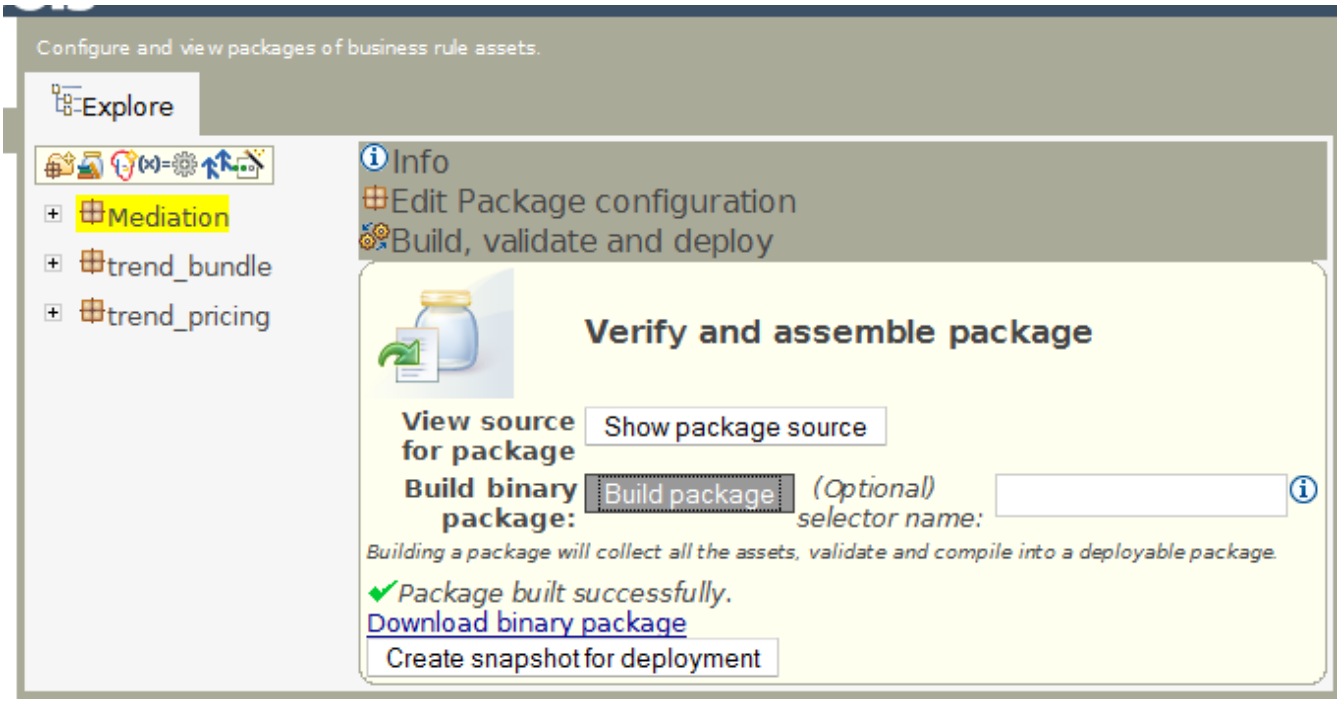
RulesMediationTask: This plug-in belongs to a new type `IMediationProcess` and it is in charge of the mediation process. This is a specific module that is out of scope for this document, it has been documented in the “jBilling for Telcos” document.

RulesPricingTask: This is an optional plug-in, that implements the new type `IPricing`. If present, it enables complex pricing policies based on rules. This class is covered in detail later in this document.

InternalEventsRulesTask: This plug-in allows rules to run in response to a configurable set of internal events. This class is covered in detail later in this document.

Deployment

You will be writing your rules, usually using Drools BRMS GUI. Then Drools will compile them and make them available to jBilling. We call this “rules deployment”. There are a few different options for rules deployment.



The first one is to tell jBilling to ask the BRMS for the rules on real-time. When a jBilling plug-in needs to process the rules, it can use a URL where the BRMS is listening so the binary version of the rules are transferred using the HTTP protocol.

The alternative is to save the binary version of the rules in a directory, and tell jBilling about this directory so it simply reads the rules from there.

Any of the plug-in classes listed earlier, and indeed, any plug-in that uses rules in jBilling, will take as plug-in parameters a series of values that will determine how jBilling expects the rules to be deployed. In fact, jBilling will be using Drools Rule Agent to find the rules, and will only pass the plug-in parameters to the Rule Agent. This means that the documentation about the Rule Agent applies to these parameters. At the time of this writing, section 9.4.4.1 of the Drools documentation goes over these parameters. Let's take a look to some of them of special importance:

- `file`: This is a space-separated list of files - each file is a binary package as exported by the BRMS. You can have one or many. The name of the file is not important. Each package must be in its own file. Please note that if the path has a space in it, you will need to put double quotes around it (as the space is used to separate different items, and it will not work otherwise). Generally spaces in a path name are best to avoid.
- `dir`: This is similar to file, except that instead of specifying a list of files you specify a directory, and it will pick up all the files in there (each one is a package). Each package must be in its own file.
- `url`: This is a space separated list of URLs to the BRMS which is exposing the packages (see below for more details).

Default value

If the plug-in has not parameters present, then it will use a default value. This is the value of the property '`base_dir`" of the jBilling.properties file, plus the directory 'rules' appended. For example, for the following entry in jBilling.properties:

```
base_dir=/usr/jBilling
```

The following parameter will be passed to the Rule Agent for the rules location:

```
dir=/usr/jBilling/rules
```

File deployment

After building the package (by clicking on 'Build package'), you can download a binary file with the package rules compiled. You only have to place this file in a directory that is visible to the Rules Agent, using the 'file' or 'dir' plug-in parameters.

This is usually a good option for a production deployment. It is faster and more robust than a URL deployment.

URL deployment

On the other hand, it is work to download and put the file in the right directory every time you make a modification to the package. Specially if you are writing new rules, or making changes. For a development deployment, it is better to use a URL deployment.

The BRMS exposes the binary of a package on the following URL:

```
http://<server>/drools-jbrms/org.drools.brms.JBRMS/package/<packageName>/<packageVersion>
```

You can use “LATEST” for the package version and the latest version will be taken. This comes very handy for a development environment: when you change any rules, you only need to compile the package. jBilling will use this new version the next time it needs rules for that plug-in.

Rules cache

Certain operations can involve many plug-ins, and some of these can be rule-based. Creating an order from the API could be one of these cases. Rule-based plug-ins can be involved in the item management and in determining the price of each item in the order. If every time a rule-based plug-in is called the rules are loaded, this can have a severe impact on performance.

There is a property in jBilling.properties that allows you to turn on and off a cache of rules:

```
cache_rules=true
```

For a production deployment, the default 'true' is typically correct. For a development deployment, you probably want this to be 'false', then it is easy to make changes to rules and see the changes immediately.

Note that the cache does not know (or check), if the rules have changed and the cache should be refreshed. It reads the rules the first time and keeps them in memory thereafter. The only way to invalidate the cache from the GUI is to make a change to the configuration of a plug-in. When, for example, a parameter of a rule-based plug-in changes, the cache for that plug-in is invalidated.

Creating new rules

Anatomy of a rule

A rule is a simple condition with a consequence. Since we are using Drools, we will use its own rules language. An example rule is:

when

customer is in Canada

then

add GST tax

There is a condition after the keyword 'when', and a consequence if the condition is met after the keyword 'then'. This is all very clear, but Drools does not have any idea about jBilling, its data model or how to, for example 'add GST tax' which is clearly an operation that jBilling would have to do.

This brings us to two key elements of a rule:

- The data model
- Helper services

The data model is needed to write the 'when' condition. The concept of a 'customer' and its address is foreign to Drools, so it has to be added at one point by jBilling. Any data available to write rules conditions needs to be explicitly exposed to Drools by jBilling.

Then there is the helper services, in the example 'add GST tax'. When the condition is satisfied, Drools will call this helper service and it is the service that will take care of the actual addition of this tax. Of course, this service is just part of jBilling.

You can see that Drools and jBilling need to be well integrated for the rules to be useful. Drools provides the ability to write, store manage and execute the rules, while jBilling will provide the data model and helper services.

This all happens in a jBilling plug-in: all the data is added to the Drools 'working memory' and the Drools is told to execute the rules. jBilling does not know about the rules themselves: how many there are, where they are, etc.

As part of a plug-in, a typical rule-based implementation will have an inner class dedicated only to act as a helper class to enable jBilling 'actions' in the rules. So the 'add GST tax' will translate to a method in an inner class of a plug-in that takes care of calling the right methods of other core classes.

Let's take a look to the previous rule, written as a 'technical rule':

```
when
      ContactDTOEx( countryCode == "CA" )
then
      order.addItem(11)
```

Now we can see something more familiar to Java code. `ContactDTOEx` is a jBilling class, and `countryCode` is one of its fields. The 'then' is just calling the method 'addItem' of the 'order' object with one parameter.

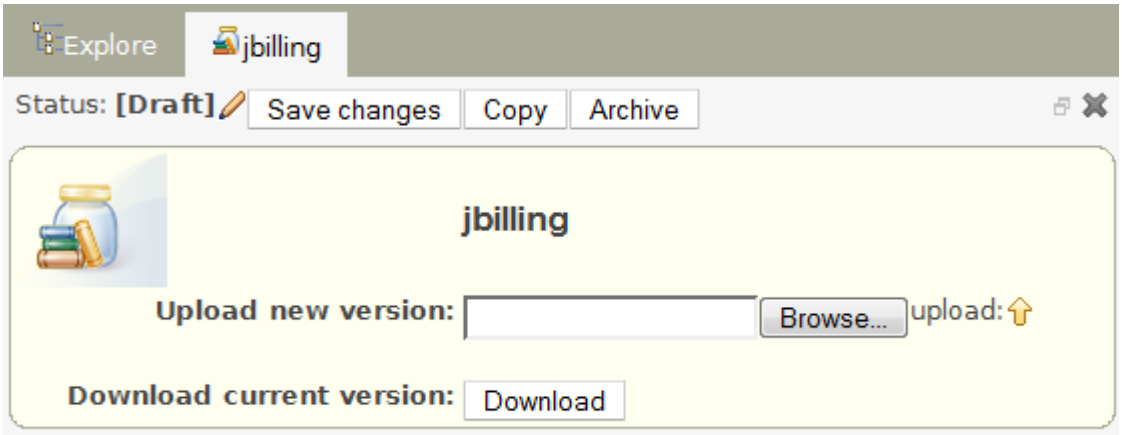
This rule would work for the `RulesItemManager` plug-in. You can see the following lines of code in this plug-in that take care of making the contact information of the customer available to Drools:

```
ContactBL contact = new ContactBL();
contact.set(userId);
rulesMemoryContext.add(contact.getDTO());
```

Later in the code, we will find the inner class '`OrderManager`'. An instance of it with the name 'order' is added as a global element for Drools. Now we can call a jBilling object directly from a rule!

When working with the BRMS you need to let it know about all these objects. This is important for it to make validations such as a typo on a class name. All you need to do is to add a model to a package. You will upload the file 'jBilling.jar' for that model. This file

has all the jBilling objects. This is a one time operation needed when you create a package from scratch:



Rules for business users

Rules can externalize business logic so it is easy to change without having to actually change jBilling. Editing, compiling and deploying rules is much easier than any changes to Java code in jBilling. Rules are meant to be deployed on-the-fly in production without downtime. Attempting the same with Java code is difficult to say the least.

All this is very good, but, wouldn't it be great if those creating and editing rules don't have to be programmers? Business users, like marketing folks and product managers should be able to work with rules without calling the IT deparment.

Business users can not deal with '`ContactDTOEx`' and the like, they need something closer to English. Drools has a good alternative with domain specific language (DSL). With this, we can provide an English alternative that then gets translated behind the scenes to the Java equivalent.

Remember that we started with our example rule with the condition "customer is in Canada", that later was written as "`ContactDTOEx( countryCode == "CA" )`" We can create a DSL sentence that will help here:

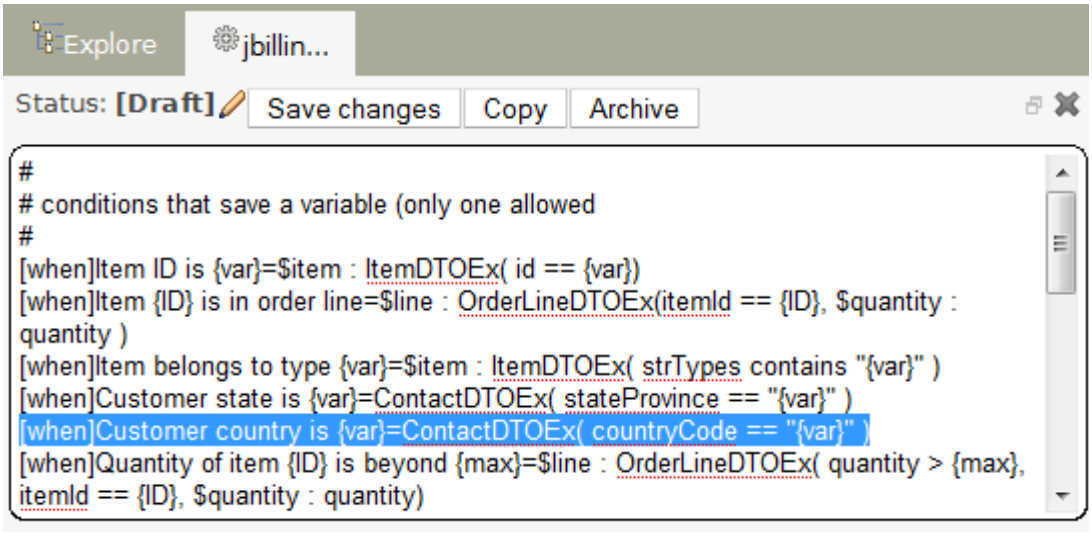
```
[when]Customer is in Canada=ContactDTOEx( countryCode == "CA" )
```

Now business users will be able to use the GUI of the BRMS and simply select this sentence from a drop down menu. Still, we don't want to have one of these sentences for each possible country. We could end up with thousands of options in the drop down menu! It is better to use variables:

```
[when]Customer country is {var}=ContactDTOEx( countryCode == "{var}" )
```

Which brings another problem. The user will need to know that CA is the code for Canada. To fix this, you can setup an enumeration, which is another feature of Drools BRMS. Take a look to Drools documentation for details.

jBilling comes with a simple DSL for each of the three packages. They are meant as examples, rather than a final version:



Steps for rules adoption

We know what the goal is: to provide our business users with a good DSL that covers most of their needs. To get there, we need to follow these steps:

1. Provide a plug-in that includes all the necessary data model into the working memory, as well as the helper services.
2. Write some technical rules that cover all the use cases and scenarios that our business rules will have to face.
3. Produce a DSL based on the technical rules that were tested in the previous step.

These three steps will be performed by developers familiar with Java, jBilling and rules. Business users will need to be involved to provide requirements and validate the results in an iterative fashion.

Let's take the following requirement as an example:

"*The system should automatically include taxes based on the customer's country of residence".*

Step 1: Plug-in We know we need two things here, one is the address of the customer, an another one is the ability to add new items to an order. If this wasn't already part of the plug-in `RulesItemManager`, you could extend it and add this functionality to it.

Hopefully your needs are covered by the existing plug-ins. Otherwise, creating a new one should not be of great difficulty for an experience Java programmer. `RulesItemManager` for example, is only 250 lines of quite understandable code.

Step 2: Technical rules You need to write an example technical rule that meets the requirement. It does not matter how 'ugly' the rule looks, with text that is totally cryptic to a business user.

Your technical rules should not be very verbose. If you find yourself writing many lines of Java code for a condition, or for a consequence, then you need to work on taking that code out of your rules. You could use functions to help you in some cases. Functions is another of Drools features. There are cases where functions are the best way to go, to do conversions for example. If your parameters are in seconds but you need to convert to minutes and round that to the next integer, you could put that in a function.

Yet, you should rely on functions with caution. It is not a good idea to have the same code duplicated in many functions across package rules, or to duplicate the code in plug-ins. As a general rule, the best way to go is to put that code as a helper method on your plug-in, typically in the helper inner class that provides services to the rules engine.

You can find the technical rule for the example requirement earlier in this chapter.

Step 3: DSL Once you have your technical rule working well, you can provide a natural language version of it through DSL. This is probably the easiest of the three steps, and the one that makes the business users the happiest. In many cases, it is as simple as writing two or three lines of text:

```
[when]Customer country is {var}=ContactDTOEx( countryCode == "{var}" )
[then]Add item {var} to order=order.addItem({var});
```

Try to keep your DSL in synch with what your business users really need. Having too many sentences when only a few get used will defeat the purpose of the DSL.

Item relationship management

Overview

When an item is added to an order, the category under the interface '`IItemPurchaseManager`' is executed. This means that you can execute rules when an order is created and items are added to it, the same applies to the modification of an order.

What kind of operations could you do with this category? When an item is being included in an order, you get control of what is going to happen with rules. You can do nothing, and let the item go into the order as normal. You can also prevent the item to get included, switch it with another one, or include other items at the same time. These are only a few options.

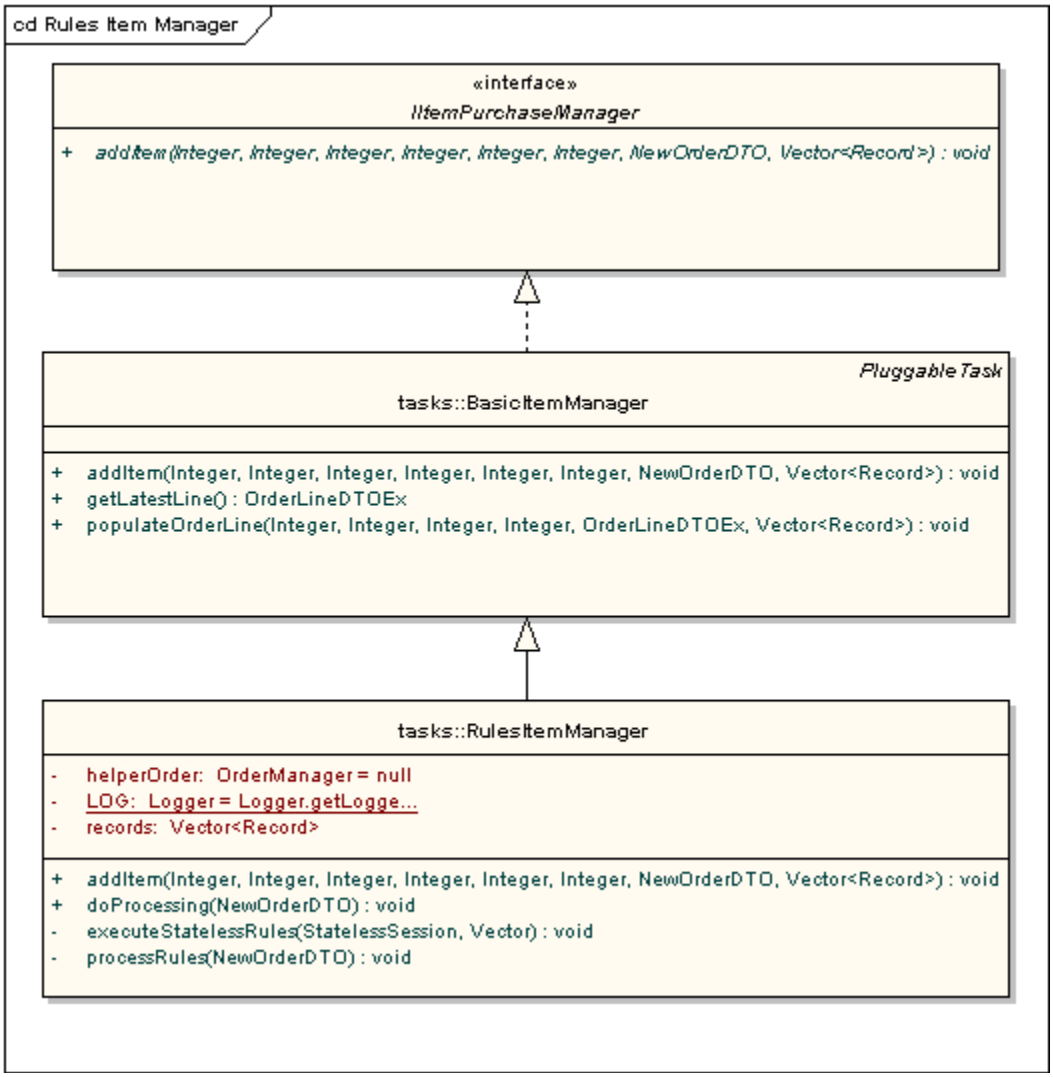
This category applies to *how items relate to each other*, as well as the behavior of items in general. In theory, you could do practically anything you want by providing your own plug-in. To stay within the expected scope of a plug-in of this type, try to only affect the

way items get into the order. Imagine if your plug-in starts sending emails when a particular condition happens, it'd be very confusion to figure out why the system is sending emails at that point in time.

The default implementation for this category is `BasicItemManager`. This is not a rule-based plug-in, and its functionality is limited to calculating the price of the order line based on the quantity and price of the item. If the item is already in the order line, it will update the quantity rather than include another line with the same item.

The interesting plug-in is `RulesItemManager`, because it is rules based. Let's take a closer look to it:

RulesItemManager



This is the standard rule-based implementation of this category. It is important to note that it is actually extending `BasicItemManager`. This means that behavior from this plug-

in, like increasing the quantity instead of having the same item in two order lines, also apply.

Take a look to the interface, `IItemPurchaseManager`. You can see how simple it is, it only has one method to add an item, that's all. Its basic responsibility is to add the item to the order by creating (or updating) an order line

As mentioned before, any rule-based plug-in will have to provide two things: the data model and helper services. Let's see how this plug-in handles that:

Data Model

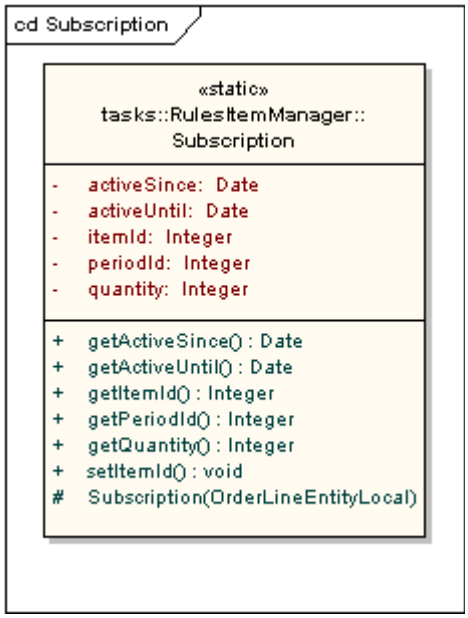
Order Lines: All the order lines present in the order are included in the working memory. This means one instance of the object `OrderLineDTOEx` per order line. This is helpful for conditions like: “only include this item if it is not already there”, or to put limits “this promotional item can sell only 10 per customer”.

User record: The record of this user, including the ID, user name, etc. This is an instance of `UserDTOEx`, you can also see it as the data present in the table '`base_user`'. You can add conditions to specific customers: “Add item '10% discount' to customer Acme'.

Primary contact: This is the address of the customer, represented by the object `ContactDTOEx`. You can write rules that affect customers depending on their address: country, state, zip code, etc.

Subscriptions: The previous classes involved in the data model belonged to the standard jBilling domain model. They are the same classes you will find when using the API, and they are very close to the related database tables structure.

For subscriptions, we created a new 'convenient' class, only for the purpose of facilitating the writing of rules. Let's take a look to this class:



This is an inner class of the plug-in. It is a clear example of what is called 'flattening the model'. The data that is exposed to the rules engine needs to be 'flat', rather than adding

a network of objects for evaluation in the working memory. This makes for clear rules and takes full advantage of the great speed of the RETE algorithm.

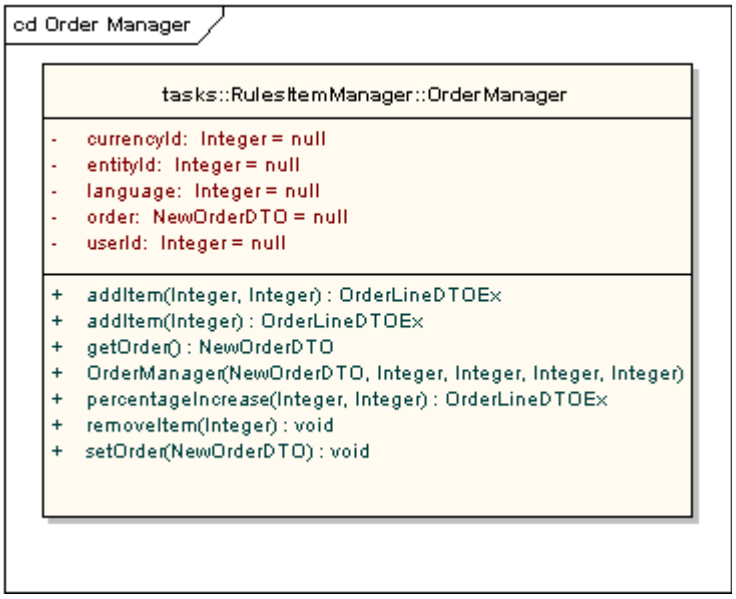
You will have an instance of Subscription *per order line, for each order that is recurring and active:*

- Only order that are active are included. This only refers to the status of the order, it does not depend on the 'active since/until' of the order. Those dates are added as part of the Subscription object.
- Orders with a one-time period are not included.

With these object available, you can write rules like “only add this item, if the customer is subscribed to this other item'.

Helper Services

Helper services for this plug-in are grouped by an inner class, `OrderManager`:



The name of the global that is actually an instance of this class is '`order`'. The important methods in this class are:

`addItem`: This simply adds and item to the order. You can specify the quantity, otherwise it defaults to zero.

`percentageIncrase`: This method takes two parameters: the first is the item to add to the order. The second one is also an item ID. The percentage price of this second item will be taken to modify the amount of the order line. For example, you sell the item 'Subscription A' with a price of 50$, and you have another item that is 'Special Discount 10%' with an percentage price of -10. If you call this method you will be given the 10% discount on the price of the item 'Subscription A'.

`removeItem`: Call this method to remove an item from the order.

Example

The following rule will replace item 16 by time 14 only if the customer is not subscribed to item 12

```
rule "switch example"
   when
      $line : OrderLineDTOEx(itemId == 16, $quantity : quantity )
      not Subscription( itemId == 12)
   then
      order.removeItem(16);
      order.addItem(14, $quantity);
end
```

Pricing

Overview

Pricing, some times called 'rating', happens when the system needs to give a price to an item. This could be very straight forward, like the price of a book. If that is the case, the basic that you get from jBilling's GUI would be enough. You could even give a special price for a customer, or a partner. Now, if you are going to have more complex conditions, like a bundle of books, or quantity discount, then you need to use a rule-based plug-in for pricing.

The plug in category is for the interface `IPricing`. Unlike many other plug-in categories, this one is optional. If the system finds a plug-in of this category present in the configuration, it will use it. If not, it will simply take the simple pricing of the item, just like it did before the release of version 1.1.0.

Before starting using a rule-based plug-in for pricing, it is important to make sure you really need it. Sometimes, your needs can be better met with an 'item management' plug-in (see the previous section). The following is an example use-case: Trend is launching its service in Florida. There are going to be many parties and speeches, but also a 10% discount for banners sold to customers in Florida for the next month.

There are two ways to tackle this:

- Use the same item 'banners', and through a pricing rule, give it a special price to all customers with a Florida address.
- Create a new item 'banners – Florida promotion', with a default price 10% lower than the standard price. Use an item management rule to switch the standard item banner for this new item when the customer buying is from Florida.

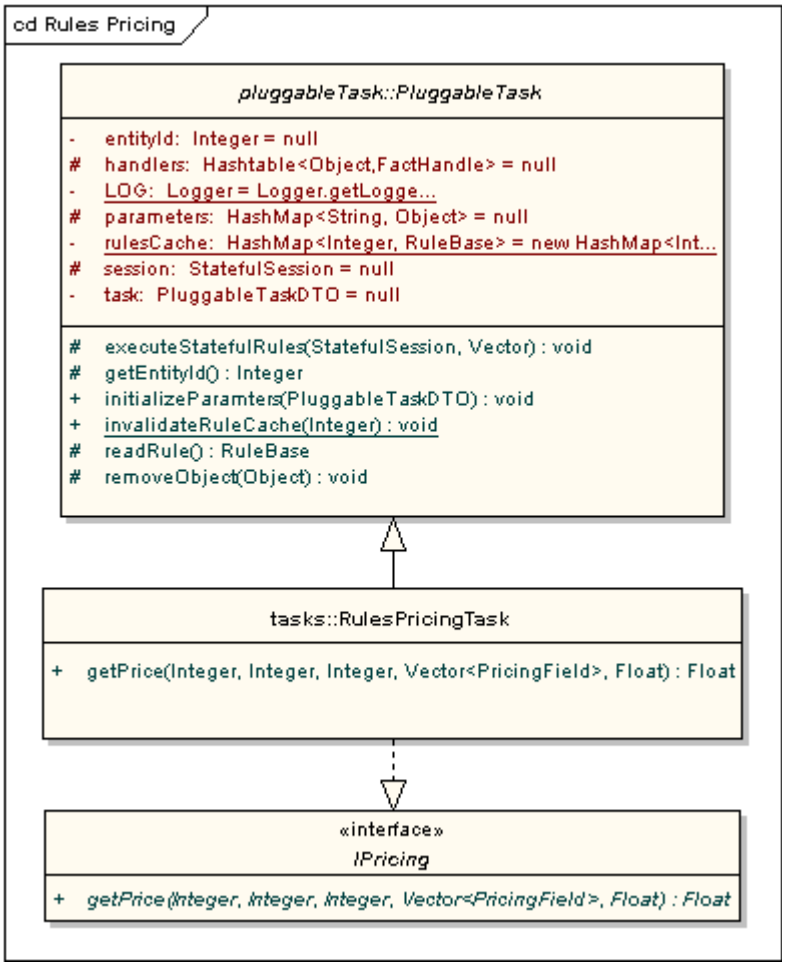
In both cases you achieve the same thing, which is to transparently give a special price to those meeting the conditions. In the first case you save on the number of items, but creating a report on how well the promotion worked will be more difficult because the same item was sold to all customers. In the second case you end up with more items,

which can be confusing if sales are done my human agents, but it will be easier to track down how much Trend sold under the promotion.

The choice becomes clear when the factors that affect the pricing are not related to the customer's account, like the address for example. When external factors related to the event that generated the sale are in play, then pricing rules become more useful. The typical example is a phone call. The price of a call will depend a lot on factors like, where the phone call originated and what the destination was.

We are getting into the territory of the mediation module, which is out of scope for this document. Let's focus on the pricing default rule-based plug-in:

RulesPricingTask



The interface that represents this plug-in is very simple, with just one method to return the price of an item. The parameters are quite self explanatory: item ID, user ID and so on. The one parameter that needs special attention is the array of objects. These objects will take the 'external factors' mentioned earlier. The plug-in `RulesPricingTask` does not have any logic based on this array, all it does with those objects is to put them in the working memory so they are available for writing 'when' conditions on your rules.

A good example for the usage of `PricingField` parameters is the mediation component. It will take all the fields from the record is processing and pass it all the way to the pricing plug-in. This component its documented in the 'jBilling Telco' document.

Data Model

Pricing fields: This object represents an external value. By default, it is used only by the mediation process. Thus, the details of this class have been documented along with the mediation module.

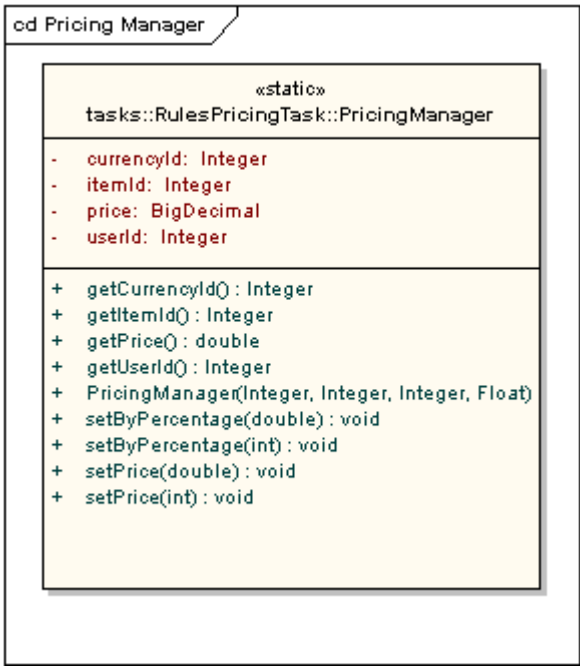
User record: The record of this user, including the ID, user name, etc. This is an instance of `UserDTOEx`, you can also see it as the data present in the table '`base_user`'. You can add conditions to specific customers: “Set price of 90$ for item 10 only to customer Acme”.

Primary contact: This is the address of the customer, represented by the object `ContactDTOEx`. You can write rules that affect the pricing of items based on the customer's address: country, state, zip code, etc.

Additional parameters: An instance of the class `PricingManager` is also present in the working memory. This class will help you with four fields for the 'when' side of your rules: currency, item, default price and user id. See the details of this class in the next section.

Helper Services

Helper services allow consequences for your rules after the 'then' keyword. For this plug-in they are provided by the inner class `PricingManager`:



The name of the global is “manager'. For example, this would set a price:

```
when
```

```
        ....
    then
        manager.setPrice(10);
```

It is not surprise that what you can do with this class is to set a price. There are two ways to do this:

- Flat price: This is just a number to assign the price of the item. For convenience, it is provided in two types, one taking and integer and another one taking a double as a parameter. The method is '`setPrice`'.
- Percentage: This will take the default price as a base, then add the percentage specified as a parameter to this method. If the default price of an item is 5, and you call '`manager.setPercentage(50)`', the result will be a price of 7.5. Once again, the same method is provided taking a double or an integer as a parameter, just to simplify the code in your rules.

Example

The following rule reads: give a special price of 9 cents on item 14 to any customer that belongs to an organization that starts with 'Acme':

```
rule "Acme deal"
   when
      PricingManager( itemId == "14" )
      ContactDTOEx( organizationName matches "Acme.*" )
   then
      manager.setPrice(0.09);
end
```

Universal events-to-rules plug-in

If you want to have rules run in response to internal events, rather than take the time to write your own rules-based internal events listener plug-in, the `InternalEventsRulesTask` plug-in may be sufficient for your needs.

Unlike other rules-based plug-ins, it doesn't provide any helper services for the rules to use. However, it can still be used for manipulating orders and invoices when the events it subscribes to occur. For example, an order line can be removed just before an order is applied to an invoice.

Rules are configured and deployed like any other rules-based plug-in. The events it subscribes to are configured in the following XML file, found in the server `conf` directory: `jBilling-internal-events-rules-tasks.xml`.

Event Subscription Configuration

Our example rule will be used to remove an order line from an order just before it is invoiced. To accomplish this, the `OrderToInvoiceEvent` is the event the plug-in will listen to. Below is an example configuration (`<beans>` tag attributes omitted for clarity):

```
<beans ...>

  <!-- List of internal events that a task subscribes to. -->
  <util:list id="invoiceEvents">
    <value>com.sapienter.jBilling.server.order.event.OrderToInvoiceEven
t</value>
  </util:list>

  <!-- Map linking pluggable task ids to an event list defined above.
-->
  <util:map id="internalEventsRulesTaskConfig">
    <entry key="540" value-ref="invoiceEvents"/>
  </util:map>

</beans>
```

First, a list given the id `invoiceEvents` is created containing the events the plug-in is to subscribe to. It contains one value, the `OrderToInvoiceEvent`. Multiple lists can be defined for multiple plug-ins

Second, a map of pluggable task ids → event list ids is defined. Each plug-in configuration has one entry. Event lists can be reused for multiple plug-in configurations. Here, a pluggable task with the id of `540` is configured to subscribe to the event list defined above it. The plug-in's id is taken from the “System” → “Plug-ins” GUI configuration screen.

Rules

The `InternalEventsRulesTask` plug-in inserts the received event object, plus the publicly accessible objects the event contains, into the rules working memory.

In our example case, we can expect the `OrderToInvoiceEvent` to be inserted, as well as the `OrderDTO` it holds. In the simple example rule below, any items with id 1 will be deleted when an order's create date is earlier than 1st July, 2009. This could be useful for removing discounts from new invoices when a promotion ends, for example.

```
rule 'Modify order'
when
        OrderToInvoiceEvent()
        order : OrderDTO(createDate < "01-Jul-2009")
then
        // delete order lines with item id 1
        for (OrderLineDTO line : order.getLines()) {
            if (line.getItemId().equals(new Integer(1))) {
                line.setDeleted(1);
            }
        }
end
```