www.jBilling.com

jBilling

The Open Source Enterprise Billing System

Telecom Guide

Copyright

Author

Emiliano Conde and others

Revision number and software version

This revision is 2.0, based on **jBilling** 2.1.0

Table of Contents

Chapter 1
Mediation & Rating

About jBilling 'telco'

Welcome to **jBilling** telco! Many believe that open source will play and increasingly important role in the future of enterprise software. Billing will not be an exception, not even for very complex and demanding industries like telecommunications. **jBilling** is the result of the collective effort of dedicated professionals, volunteers and companies that need an enterprise billing solution.

A billing system is at the heart of many organizations. It is a mission critical system. Its importance is such that many companies opt to develop their own simply to be able to retain full control of the billing solution. An open source billing solution is very attractive in this scenario: by having the source code, a company can have full control of their billing solution. It can develop plug-ins to meet its own business needs, its developers can acquire the know-how of every aspect of the billing system. The company can achieve full control of their billing system without writing one from scratch.

Until the release of **jBilling** 1.1.0, **jBilling** did not have enough flexibility to tackle complex business rules related to bundling, item relationships, taxes and pricing. Additionally, for telecom billing a whole module was missing: mediation.

The recent integration with a Business Rules Management System (BRMS), a rules engine and the development of the mediation module put **jBilling** in the league of billing systems needed for the telecommunications industry.

Two years after that initial 'telco' release (1.1.0), a new release of **jBilling** pushed the telco capabilities much further. **jBilling** 2.1.0 increased performance radically, to the point of several hundred of records a second. It also included real-time mediation features and comprehensive management of CDRs error cases, along with other improvements.

In only a few years, **jBilling** has evolved very rapidly to the point that is now competing with proprietary billing systems with licensing cost in the hundreds of thousands or even millions of dollars.

In this document we will focus on the mediation component and describe scenarios that are typical for telecom billing. **jBilling** will continue its fast evolution, adding more features and becoming more robust. For that, we need your help. Participate in the **jBilling** community by posting on the forums, reporting bugs, submitting plug-ins or code.

Who should read this?

The audience of this document is those **jBilling** users that need to set-up a mediation process. This usually translates to telecom companies, but this is not exclusively for them. There are other industries that also have this type of requirement and do not provide phone services.

The mediation module can process events of any type, not just traditional phone calls: instant messaging, downloads, data, VOIP, IPTV, etc. If there are events generated by an external system that should be fed into the billing system, jBilling's mediation module can take care of them.

Requirements

To fully understand this document, you need to have some previous knowledge of **jBilling**. The concept of a customer, an order, a payment, and invoice, etc... needs to be clear. These are explained in the *jBilling User Guide* document.

The mediation module makes extensive use of **jBilling**'s plug-in architecture and the integration with the Business Rules Management System. These are covered in the *jBilling extension guide* document, it is recommended you read and have that document handy when going through this document.

You can download the latest versions of these two documents for free at the documentation section of the **jBilling** web site.

This document at times explains how to extend or customize the behavior of the mediation module. To take advantage of this, you will need to understand the basics of software engineering and the Java Enterprise Edition platform.

The need for mediation

A phone call is an event that takes place outside the billing system. From a billing perspective, a phone call is actually a purchase of a service by a customer. This event is known to the telephony system and registered as a record in some storage device.

A phone call is just an example, there are many other types of events that happen outside the billing system and yet should be sooner or later included in an invoice for the customer to pay. Downloading content, for example, or sending an instant message from a cell phone, receiving an email, etc... Any event where a customer is involved making usage of services need to be communicated to the billing system, otherwise nobody is paying for those services!

Because of the nature of these events, the information that the system records could be in very different formats. Each telephone switch vendor has its own format, and that is only for phone calls (voice). All the other services that involve data, messages, etc can be delivered by their own systems that follow very different and proprietary formats. On top of all that, new types of services are being launched all the time.

There are two main approaches to make the billing system aware of these events. There is the batch method, where the mediation process periodically goes over these records and submit the relevant information of each event to the billing system. The other method is to do it all in real-time: as soon as an event happens, the even is sent to jBilling for immediate processing, which includes rating and updating the customer's account.

In any case, the mediation component has to be very flexible to allow the processing of new types of event records with only changes to configuration files.

Flexibility is key, but there is another important challenge: performance. The events that the mediation process needs to go through happen very often. Each phone call is an event! A company offering phone services could easily have millions of these events every day, and they all have to be managed by the mediation process.

jBilling addresses all of the above: batch mediation, real-time mediation or a combination of both. It is very flexible and fast It is not an isolated module that communicates with the billing engine; instead, it is built-in to the core of **jBilling** to improve both the flexibility and performance of the overall billing solution.

Rating events

The system that first knows about an event and records the event does not have any knowledge on how to charge for the event. A telephony switch will know which phone number the phone call originated, the number that was called, when that happened, how long the call was,etc..

All this is very useful, but stops short of the key question: **how much did the phone call cost?** This then is related to other questions like: who is the customer? is the customer subscribed to some kind of plan or bundle that make the phone call free?

The same applies to any other type of event. A web server will be able to tell the IP address that downloaded a music file, but it is unaware of the price the customer should pay for this file.

Rating is the process of giving a price to an event.

The process of assigning a price to an event is what we call rating. **jBilling** has very advanced built-in rating capabilities based on rules managed by a Business Rules Management System. All the data contained in the event is available for rating. For example, the price of a phone call will greatly depend on the day, time, length, origin, destination, etc. These are data fields contained in the event that are made available by the mediation process to the rating engine of **jBilling**.

Mediation and Rating with jBilling

Let's now go over a high level description of how mediation and rating work with **jBilling**. Most of the remaining of this document will be a detailed descriptions of the modules and processes mentioned in this overview. You can use this initial overview as a road-map to all the pieces that come together to provide mediation and rating in **jBilling**.

There are two different ways to get an event processed (mediated). One is to have in a file or DB for batch processing, and another one is to do it in real-time by calling an API method when an event happens.

Let's start with a diagram of a batch scenario:

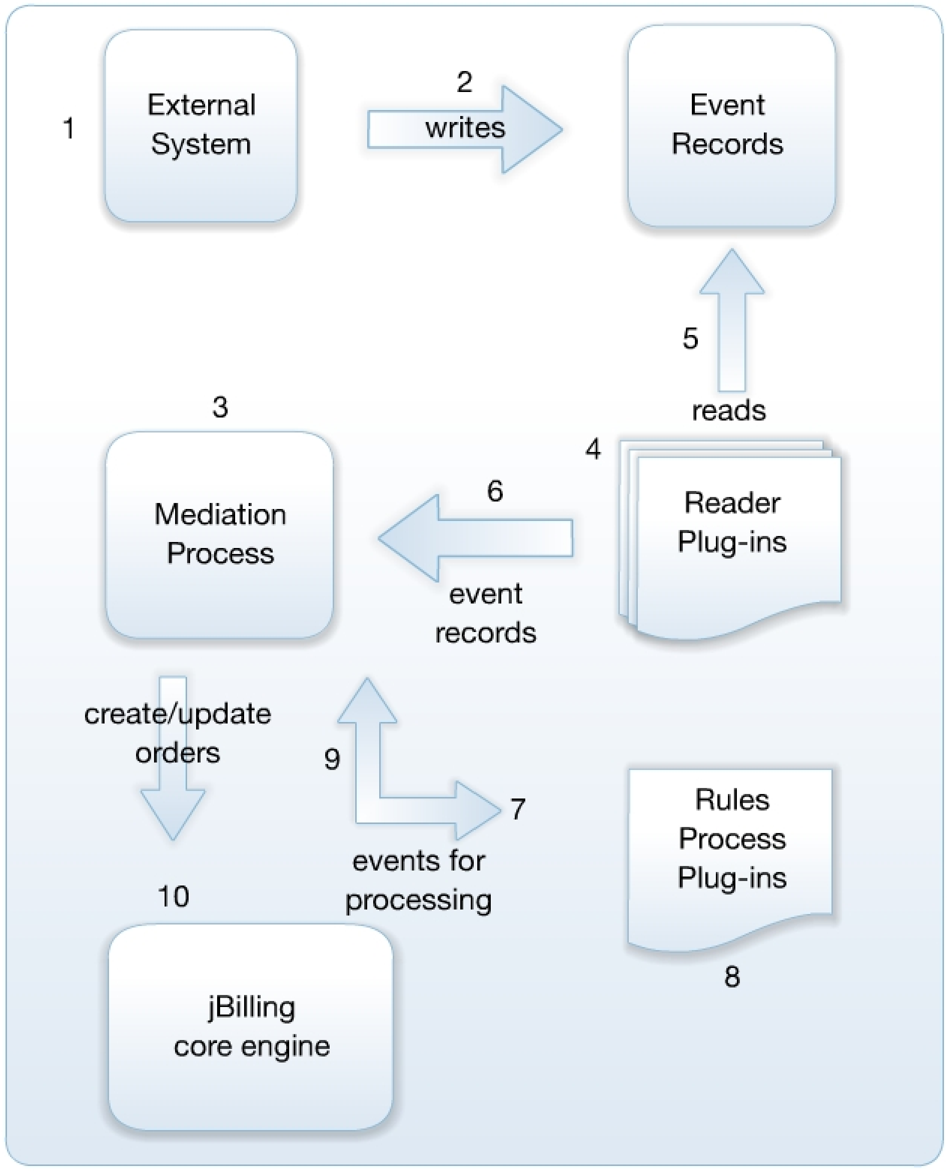


Illustration 1: Overview of jBilling's mediation flow(batch)

1. The external system provides a service to a customer. This is typically a telephone switch for phone calls, but it could be any other system, like a web server serving content to internet users.
1. The external system saves an event record with the details of the service provided to the customer. The storage is usually a text file, but it could also be a relational database (RDBMS) or other types of storage. For the telephone switch example, an event happens every time a phone call is attempted. The event records are then typically called 'Call Detailed Records' (CDR). The event will have all the data later needed to rate and bill the customer.
2. The internal **jBilling** scheduler starts the mediation process. This happens periodically, typically daily. The mediation process will find the configured plug-ins that have the logic on how to read the events and what to do with them. This

component is just an 'orchestrator', it does not know how to do the actual processing. For that, it fully relies on plug-ins.

3. The mediation process will read the configuration records. Each configuration record points to a reader plug-in. The process then calls each of these reader plug-ins.
4. A reader plug-in knows how to read events: their location, the format, how to divide a record in meaningful data fields. Still, the scope is limited to reading. The reader does not know the meaning of these data fields.
5. The data of a group of records is now returned as Java objects to the mediation process.
6. The mediation process calls the processor plug-in with the data of an event record.
7. The processor plug-in can decide what to do with the record. The standard implementation is a rules-based plug-in. This means that you will be using BRMS to process the event. This step gives meaning to the data. Here is where a field in a record is transformed into units, or a date, or a way to identify who, in the billing system, should be paying for this event.
8. The process returns to the mediation process read-to-use billing data about the events: what item was sold, when the event took place, and the customer involved in the event.
9. The mediation process can now make a simple call to the **jBilling** core to create or update a purchase order. This last step gets the billing system updated with the activity the customer did when interacting with the external system in the first step.

Let's now review this same diagram, but in a real-time scenario:

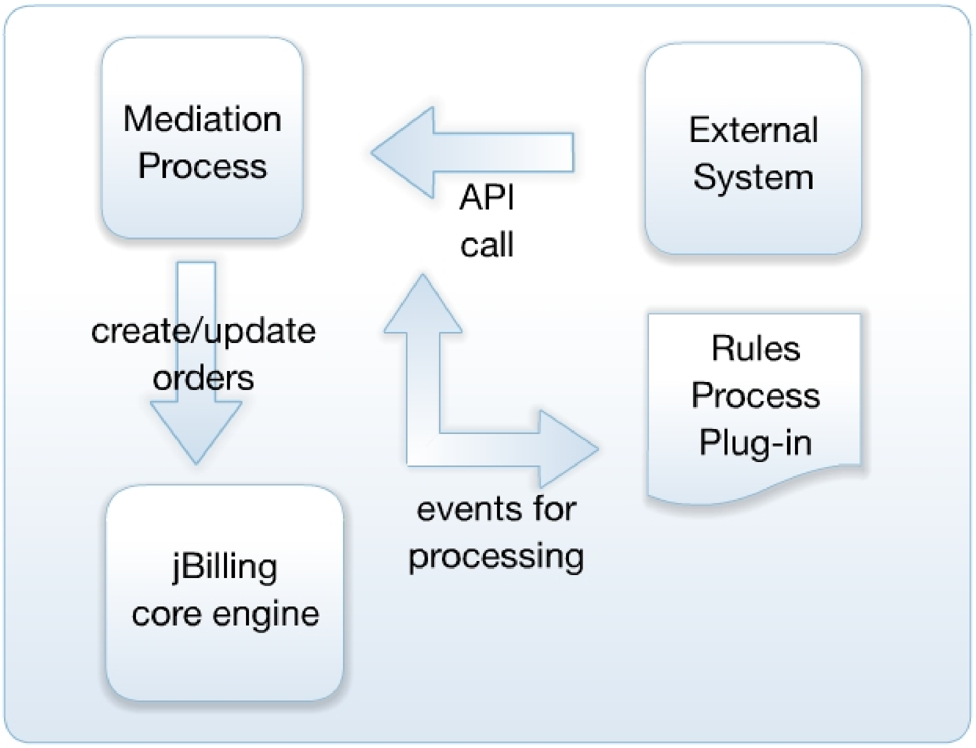


Illustration 2: Real-time mediation data flow

In a real-time scenario, the records are not stored. Instead, they are sent as they happen to the billing system. The above illustration shows basically the same flow as the batch diagram, but in this real-time case the readers are gone.

The readers have only one concern, and it is to read records from there the external system saved them. If that is not needed, then we don't need readers. The external system will be calling jBilling every time a new event happens (or is about to happen, as we'll see later).

This real-time call is done using the standard **jBilling** API (usually using SOAP as a protocol). There are specific methods in the API to allow the processing of events by the mediation process. It is worth noting that the mediation process itself, the module, is unaware of how the records are delivered to it. It does not know if the records are coming from a file that was read by a reader, or they came through the API in real-time.

This is important, because it means that the configuration of the mediation process remains unchanged regardless of the delivery mechanism or source of the CDRs. This will then allow for setups where both batch and real-time processing are in place.

Updating orders

The preview section ends with step 10, when the mediation process updates or creates an order to reflect the event into the customer's account. A fair question at this point is what order is being updated? when does the mediation process create an order, an when does it just update an existing one? what kind of order will it create?

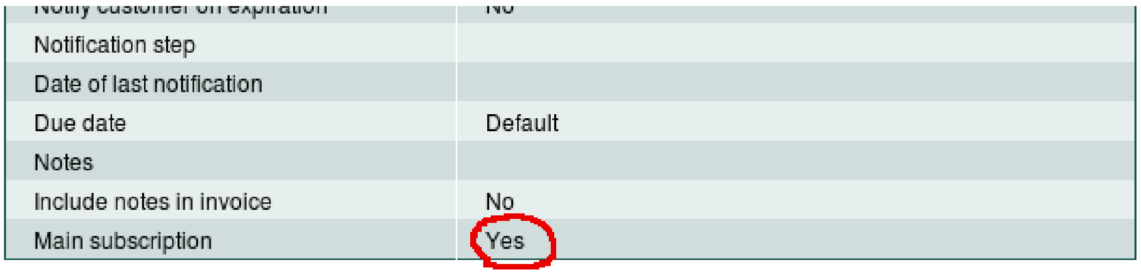
The events being processed by the mediation process are purchases of services from your customers. As such, to translate this to the **jBilling** world, they have to be put into purchase orders.

The mediation process converts external events into purchase orders

The mediation process, being a fully automated process by nature, needs to decide how the events will be placed into orders. *The mediation process will only create or update* ***one time*** *orders*. The reason for this is simple: the events begin processed are naturally one-time purchases.

It would not be practical to create one order for every event: a customer can end up with hundreds or more orders every month. It is better to update an existing one-time purchase order, and just have *one purchase order per billing cycle*.

This brings us to the concept of 'main subscription': a recurring order that sets the length of a customer billing cycle as far as the mediation process is concerned. The main order is recurring: it generates an invoice periodically. The only difference between a 'main subscription' and any other recurring order is a flag the makes the order 'main subscription':



Notify customer on expiration	No
Notification step	
Date of last notification	
Due date	Default
Notes	
Include notes in invoice	No
Main subscription	[✓ Yes]

Illustration 3: The 'main subscription' flag

A customer can only have one recurring order set as 'main subscription'. In this recurring order you will usually add periodic recurring charges, like the 'plan' fee. This is not a requirement, you could have a main subscription with a total amount of zero. The mediation process will look into the main subscription order to decide if a new order needs to be created for the event being processed, or an existing order can be updated. This is basically logic based on dates: the date of the event, and the dates of the main order.

There will be only one one-time order with charges coming from events per period. This is, if the main order is a monthly order, then there will be only one one-time order with events created by the mediation process per month.

The mediation process never updates the main subscription order

An order created by the mediation process will have some clarification text in the 'notes' field of the order. It will simply say that the order was created by the mediation process. Beside this, there isn't anything different between this order an one that you create, and nothing prevents you from editing the orders created by the mediation process: it is just not a good practice. Although it should not bother the mediation process, it is just less confusing if you create your own orders and leave the ones created by the mediation process untouched.

Let's see an example: a customer has a recurring order, monthly, set as main subscription. This order started on January 15 and has generated two invoices, one for the period 1/15 to 2/14 and another one for the period 2/15 to 3/14.

When the mediation process runs there are three events for this customer:

1. 3/17 for 10$
2. 3/24 for 5$
3. 4/15 for 1$

For event 1, the mediation process creates a new one-time purchase order with an order line of 10$. The actual item, quantity and price for the order line are part of the mediation rules and will be covered in detail in later chapters.

For event 2, the mediation process finds that there is already a one-time purchase order that covers the period 3/15 – 4/14. It then just updates this order (the one created when the previous event was processed). Depending on other factors, this can be just an update on the quantity of an existing order line, or a new order line.

For event 3, the mediation process can not find a one time order for the period that starts on 4/15, so it creates a new one.

The way the mediation process identifies a one-time order for a specific period of time is by checking the 'active since' field. The first order created for events 1 and 2 will have an active since of 3/15, while the second one will be 4/15. The mediation process knows that this orders cover only one month because that is the recurring period of the main subscription order.

In a typical configuration, when the billing process runs to produce an invoice for the period starting on 3/15, it will find two applicable orders: the main subscription order and the one-time order with the events from the mediation process. It then generates a single invoice for the customer with all the compiled charges; the customer does not want to know anything about mediation processes and orders, just a simple invoice with the right charges will do.

The billing process in this case will ignore the other one-time order because its active since is too far in the future, that order will be picked up the following month. However, the first one-time order will not be picked up the next month because, being a one-time order, its status is set to 'finished' after being applied to an invoice.

Chapter 2

The Mediation Process

Overview

The mediation process is the component that orchestrates and coordinates all the steps needed to complete the mediation. By itself, it doesn't know much on what to process and what to do with the events. For that, it checks in the **jBilling** plug-in configuration to find out the Java classes that can take care of those tasks.

The mediation process is implemented as a stateless session bean, the class name is `MediationSessionBean`. The internal **jBilling** scheduler will call this class to tell it that it is time for the mediation process to take place. The class is acting as a 'facade' to the external world to allow an easy entry point where to trigger the mediation process.

Deployment

The mediation module is embedded with **jBilling**. You don't need to do any special installation specific to the mediation module. Installing a server with **jBilling** is all you need to have the mediation module installed.

This does not mean that there aren't any deployment options. You can run your mediation process in the same physical server as the **jBilling** server, or you can run it on its own in a different machine. This second option is usually recommended but for the least demanding scenarios, the mediation process requires a good deal of processing power.

As mentioned before, a **jBilling** server can do everything or just some billing functions. To activate or deactivate a role, we need to edit the configuration file '`jbilling.properties`', located in the directory `jbilling/conf`.

The following is an example where all the roles are active, this server will perform all the billing functions:

```
# if the daily batch includes running the billing process

process.run_billing=true

# if the daily batch includes running the ageing process

process.run_ageing=true

# if the daily batch includes running the partner process

process.run_partner=true

# if the daily batch includes running the order expiration
notification process

process.run_order_expire=true

# if the daily batch includes running the invoice reminder
notificationprocess

process.run_invoice_reminder=true

# if the daily batch includes running the overdue penalties process

process.run_penalty=true

# if the daily batch includes running the list statistic collection
process

process.run_list=true
```

```
# if the daily batch includes running the credit card expiration
notification process

process.run_cc_expire=true
```

`# if the daily batch includes running the mediation process`

`process.run_mediation=true`

Now, if we want to deploy a mediation-only server, we only need turn all the above properties to 'false', except the '`process.run_mediation'` property that should remain 'true'.

We could even have multiple servers doing mediation. At the time, this requires distributing the event files to each **jBilling** server doing mediation:

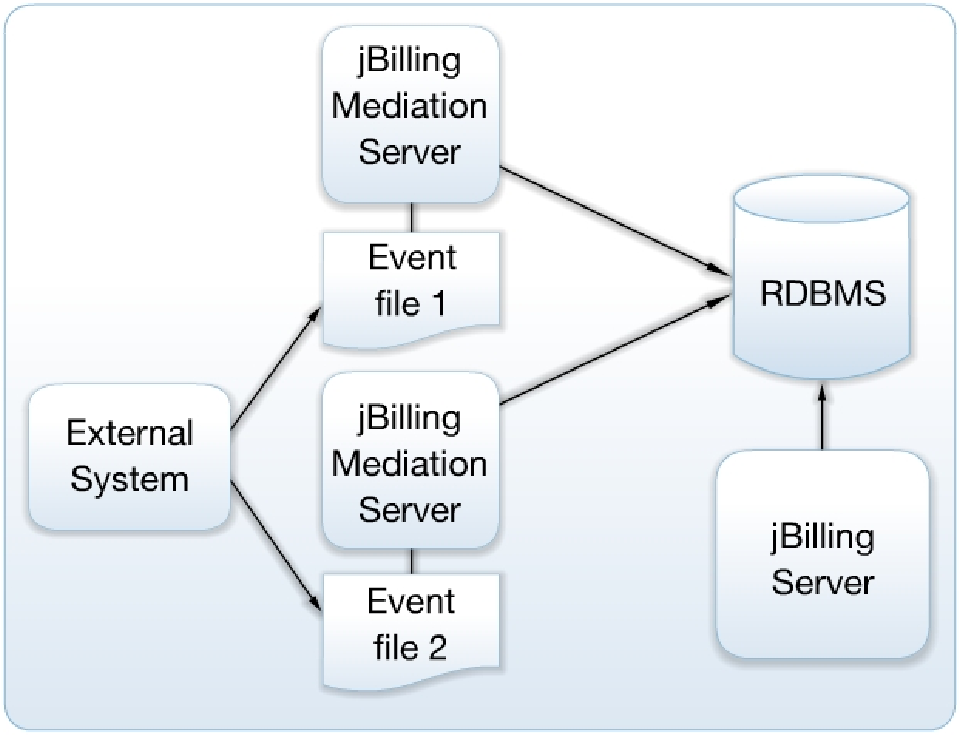


Illustration 4: A deployment with two mediation servers

In the above illustration, we have the external system transferring the event files to two different mediation servers. Those servers are fully dedicated to process mediation events. They have deployed locally all that is needed to process those events, so they can interact directly with the database server where all the billing information is stored.

To complete the example, there is also present another **jBilling** server that is doing everything but mediation: serving request to web clients, serving web services, running the billing process, etc.

The above scenario is certainly possible, but it isn't a likely one. Even in very demanding scenarios, a single server dedicated to processing events is enough to deliver the throughput required. Since version 2.1, jBilling's high-performance for CDR processing has made multi-servers mediation deployments unnecessary.

Performance

A common concern when thinking about mediation is performance. After all, the amount of events to process in a daily basis can be huge, even for medium sized companies. In a typical billing system, the mediation process is done in three very defined phases: mediation, rating and account updates.

jBilling can fully process hundreds of records per second.

The first one is mostly a 'data processing' process: filter out records that are not needed, combined others, and in general leave the records in a format that is easy to rate. Then it comes the rating that assigned the prices to each record. Finally, the billing system is called to update the customer's accounts with the new charges.

Some systems make claims on performance related to processing records, when they only do one of these three steps: they only rate, or the only mediate. In **jBilling**, all these steps are still there, but they are done in one single process. You can expect a performance of hundreds of records per second with all three phases included in the processing.

Even better is the fact that all this is done without hard-coding your mediation and rating rules logic in Java or C language, nor with headache-producing XML file configurations. Instead, **jBilling** uses a rules-based processing method, embedding the JBoss rules (drools) rules engine to represent the business rules needed for mediation as exactly that: business rules to be processed by a rules engine.

Batch size

The key to **jBilling's** remarkable speed while using a rules-engine is the way it processes the events: it does not take one event and sends it to the rules engine for mediation, then rating, then update the customer account. It does all those three thing in one call to the rules engine.

Furthermore, it takes a *group* of events and send them for processing to the rules engine. The mediation module will ask the reader plug-in for a group of records to process, and it will pass this group to the mediation plug-in (typically the rules based one).

The value of how many records to process at a time is then part of the reader's parameters. The standard reader plug-ins that come packaged with jBilling take the parameter `batch_size` to define this value. The default is 100.

The higher this number, the more memory you will need for the mediation process and the more the rules engine will have to do to orchestrate the many events that are at any given time in the memory context. On the other hand, the smaller this number the more times the rules engine is called, which represents some overhead.

Thus, the ideal number is not as simple as 'the higher the better'. A batch size of 100 (the default) is usually good. In many cases the best performance came from values between 10 and 30.

There are many factors that will influence the ideal batch size: the format of your records (how many fields, data types, etc) and the type and number of rules are the most important. The easiest way to find you ideal batch size is to simply experiment with a few different values. Do this only after you have completed all your rules and your configuration is fully tested, otherwise you might be taking readings on a moving target.

Last but not least, do keep in mind that by default, the Tomcat installation that comes with the standard **jBilling** has configured a maximum usage of RAM of 256 MB. This is very little memory for the mediation process. Depending on your batch size, you will have to increase this. Take a look to the `catalina.sh` or `catalina.bat` files. There you can find the JVM parameters that include the maximum memory.

Buffer size

For file-based readers, there is also the size of the buffer used when reading data from the CDR file. These readers take the parameter `buffer_size`. The default is 8192 and the value is in characters. When the reader has to access the file with records in the hard drive, it will spend some time just requesting access to that file to the operating system.

This overhead is specially high if, to read the number of records that the batch size needs, the reader has to go do many IO interactions with the hard drive. The ideal buffer size then depends on the batch size and the size of each record.

This parameter is used by **jBilling** only as a parameter to the Java object BufferedReader.

Rules optimization

The main advice about rules optimization is to avoid doing it. Make sure that you have already tried the previous performance optimizations before starting optimizing rules. In most cases, even the worst written rules (from a performance perspective) perform pretty well.

Having said that, there are some typical cases that can speed up the process significantly:

Avoid 'regex' and favor '==' as operators

A common implementation of rate cards involves a decision table with one line per rate and a 'prefix' to match the phone number called and the assign a price. This is very easy to implement and usually has an acceptable performance. Here is an example from a real case:

```
name == "dst", strValue matches "^$param.*", resultId == $resultId
```

Yet, the rules engine does not like the 'matches' operator very much. With it, it can't optimize how the rules 'net' is formed in memory and it has to do a brute force comparison.

Since these rate cards usually have thousands of rows (which translate to thousands of rules), you can see a significant increase in performance for the price of a more complicated decision table.

The trick is to replace the rule with the 'matches' operator for many rules with the '==' operator, each of these new rules working on one digit of the prefix that you need to match. The result is a more a decision table with many more columns, but they can be well arranged so they are not too hard to deal with.

The rules engine will now be able to do node sharing and node indexing of these thousands of rules.

Write rules using the same order

The order of conditions within a rule and even the order of constraints within a single condition affects the order of within the net of rules that the rules engine creates. This in turn will allow the rules engine to share nodes. The more nodes a rules share with other rules, the better.

Let's see a simple example. Take a look to the following two rules:

```
rule 'user setter from username'

when

    $company : CompanyDTO( )

    $result : MediationResult(step ==
MediationResult.STEP_1_START, userId == null)

    $field : PricingField( name == "userfield", strValue matches
"^[a-Z].*" resultId == $result.id)

then

    modify( $result ) {

        setUserId( getUserIdFromUsername($field.getStrValue(),
$company.getId()) );

    }

end


rule 'user setter from id'

when

    $result : MediationResult(userId == null, step ==
MediationResult.STEP_1_START)

    $field : PricingField( name == "userfield", strValue matches
"^[0-9].*" resultId == $result.id)

    $company : CompanyDTO( )

then

    modify( $result ) {

        setUserId( getUserId($field.getStrValue(),
$company.getId()) );

    }

end
```

One of the can be re-organized to match the same 'sequence' of rules as the other rule. If we take the second rule and rewrite it like this:

```
rule 'user setter from id'

when

    $company : CompanyDTO( )

    $result : MediationResult(step ==
MediationResult.STEP_1_START, userId == null)
```

```
    $field : PricingField( name == "userfield", strValue matches
"^[0-9].*" resultId == $result.id)

then

    modify( $result ) {

        setUserId( getUserId($field.getStrValue(),
$company.getId()) );

    }

end
```

Then the rules engine will be able to share many of the conditions that both rules need. This means a smaller and faster 'net' of rules in the memory context.

Write rules ordering by the most restrictive condition first

Just like with standard procedural languages, if you start a chain of 'ifs' with the one that is most likely to fail, then you will be saving processing time by avoiding doing unnecessary tests of conditions.

Let's take the previous rule. The condition of existence of the object '`CompanyDTO`' if poorly placed. This will actually never fail. It is better to push this condition to the bottom of the rule:

```
rule 'user setter from id'

when

    $result : MediationResult(step ==
MediationResult.STEP_1_START, userId == null)

    $field : PricingField( name == "userfield", strValue matches
"^[0-9].*" resultId == $result.id)

    $company : CompanyDTO( )

then

    modify( $result ) {

        setUserId( getUserId($field.getStrValue(),
$company.getId()) );

    }

end
```

The rest looks fine. It is a good practice to start a mediation rule with a condition about its step. This is quite restrictive and it is easy to remember, which leads to all the rules to start the same way for automatic node-sharing.

Scheduling

OK, we have now a server that is going to be doing mediation. We only had to install **jBilling** and make sure that the '`process.run_mediation'` property is set to 'true'. Now, when is it that the mediation process will take place? And how often will it run?

For this, we also use properties in the '`jbilling.properties`' configuration file. The two properties involved are the same used to run the billing process:

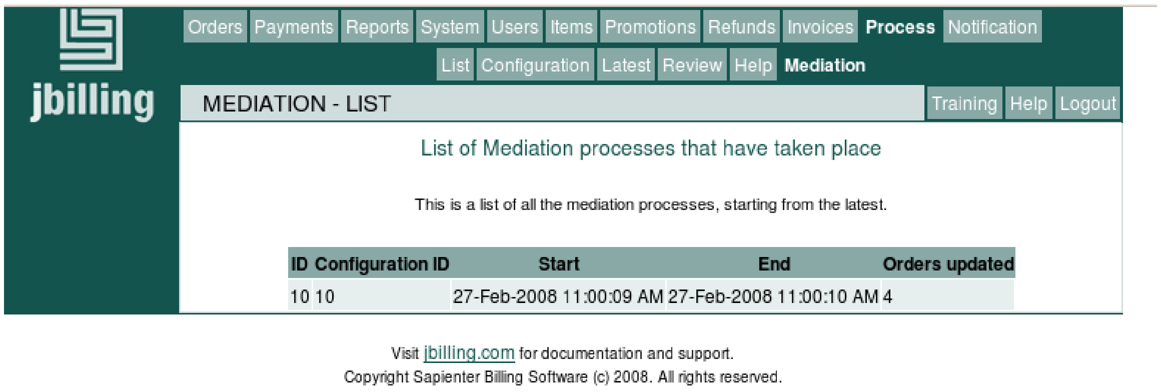
```
# These two properties set the frequency of the jbilling batch process
# The fist property indicates at what time of the day the trigger has to
# happen for the very first time. After this first run you will need X minutes
# (specified by 'process.frequency') to run the trigger again.
# The first property is optional. If it is not present, then the next trigger will happen at
# startup + minutes indicated in 'process.frequency'.
#   time: YYYYMMDD-HHmm (a full date followed by HH is the hours in 24hs format and mm the minutes).
process.time=
#   frequency: the number of minutes between runs
process.frequency=720
```

In fact, these properties dictate when the 'batch' process, in general, will run. What will be included in that batch process is then up to the boolean properties that start with 'process.' referred in the previous section.

In the previous example, the batch process (that will be including mediation) will run every 12 hours, starting 12 hours from the moment the server goes up.

The list of mediation processes

When a mediation process starts, you can verify that this happened by taking a look to the mediation process list. Click on 'Process', then on 'Mediation' to access the list screen:



List of Mediation processes that have taken place

This is a list of all the mediation processes, starting from the latest.

ID	Configuration ID	Start	End	Orders updated
10	10	27-Feb-2008 11:00:09 AM	27-Feb-2008 11:00:10 AM	4

Visit jbilling.com for documentation and support.
Copyright Sapienter Billing Software (c) 2008. All rights reserved.

Illustration 5: The list of mediation processes

When the process is on-going (it has started, but has not finished yet), the 'End' and 'Orders updated' columns will be blank. Let's review each column's meaning:

ID: This is a simple unique identifier of this mediation process.

Configuration ID: Here you can see which mediation configuration the process used for the processing. This is useful because you can have multiple configuration present at the same time. The subject of mediation configurations is covered in the next section.

Start: The time the mediation process started.

End: The time the mediation process ended.

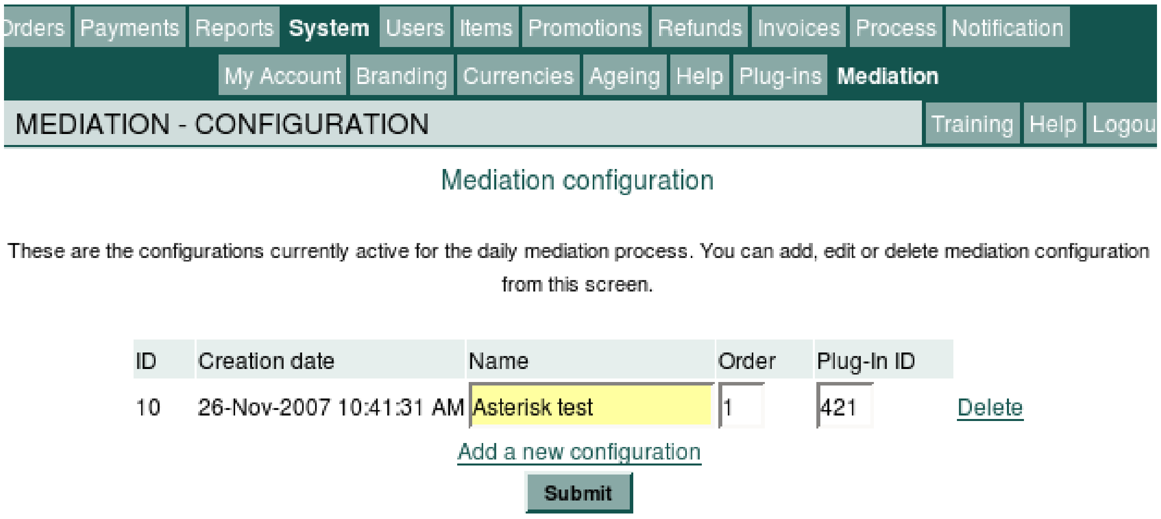
Orders updated: How many orders this process created or updated. It is common that this number is equal to the number of records present in the files processed, but it doesn't have to be. There are many circumstances where records do not translate into an order updated.

Chapter 3
The Mediation Configuration

Once the mediation process is setup, the next step is to add 'mediation configurations'. A 'mediation configuration' tells the mediation process where the event records are and how to read them by pointing to a plug-in that knows how to perform those tasks.

You need to have at least one mediation configuration for the mediation process to do something, if there aren't any configurations present when the mediation process is triggered, then nothing will happen.

To manage the mediation configurations click on 'System' and then on 'Configuration':



MEDIATION - CONFIGURATION

Mediation configuration

These are the configurations currently active for the daily mediation process. You can add, edit or delete mediation configuration from this screen.

ID	Creation date	Name	Order	Plug-In ID	
10	26-Nov-2007 10:41:31 AM	Asterisk test	1	421	Delete

Illustration 6: The configuration screen

To create a new configuration click on the link 'Add a new configuration'. The ID and creation date will be given by the system. You can enter a name to later identify this configuration.

You will need one configuration per 'type' of event. For example, if you need to process voice calls and SMS messages, then you will need two configurations. You can name the first one 'Voice' and the second one 'SMS'. This assumes that those two event types are stored in their own files, with their own formats.

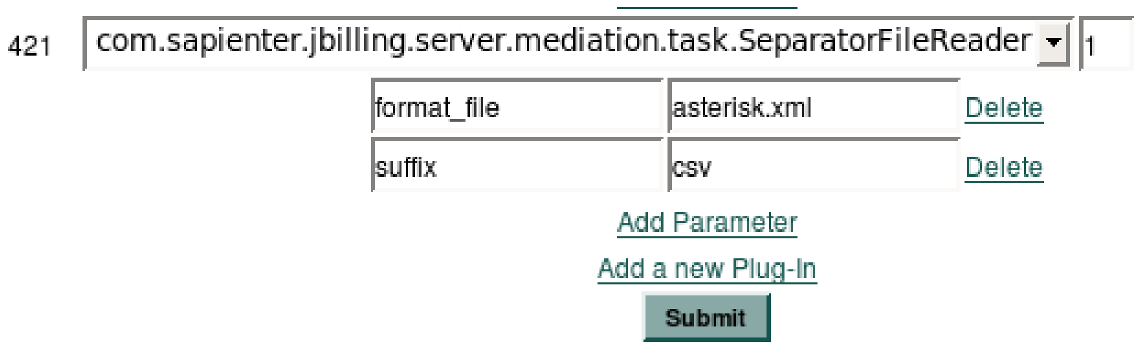
The order number tells the mediation component which configuration to use first, which one is next, etc. The plug-in ID points to a reader plug-in. This plug-in is the component that will do the actual record reading, and it is covered in the next section.

Chapter 4
Readers

The readers plug-ins

A mediation configuration needs the ID of a reader plug-in. This is the way the mediation process actually knows what class is going to be doing the record reading. Because there is a one-to-one relationship between a mediation configuration and a reader plug-in, you will only need several configurations when you have the need to use many different readers. This happens when you have files with different formats or various sources like files and databases all of them with events to process.

You can add a reader plug-in just like any other plug-in from the 'Plug-ins' screen. Click on 'System' then 'Plug-ins':



421 com.sapienter.jbilling.server.mediation.task.SeparatorFileReader 1

format_file	asterisk.xml	Delete
suffix	csv	Delete

Illustration 7: A reader plug-in

Currently, **jBilling** comes with four possible reader plug-ins, one for files with fields separated by a character, another one for files with a fixed record length, a generic JDBC database reader, and a MySQL database reader:

- `SeparatorFileReader`: This plug-in can read text files where the fields are separated by a string, usually a characters like a comma.
- `FixedFileReader`: This plug-in can read text files where the fields occupy exact positions in the record, and the record itself is always the same length. Each field is not separated by any character.
- `JDBCReader`: This plug-in can read records from a database using JDBC. It supports two schemes for marking which records have been read. This first way is by remembering the last id read by the last process, then only reading records with ids greater than this value. The second way is by using a timestamp column to mark the read records.
- `MySQLReader`: This plug-in is an extension of the JDBCReader plug-in and provides some default values and simpler plug-in parameters for reading from MySQL databases.

The file reader plug-ins

The file reader plug-ins hold the information of where the record files are located and what format it is that they follow.

The following parameters are in common to both readers:

`format_file`: This is a required parameter. It is the name of the format file. We will discuss the content and function of the format file in the next section.

`format_directory`: The directory where the format file is located. This is an optional parameter, if not present, the value of the property 'base_dir' is used with an appended 'mediation' directory. The default directory is then '`jbilling/resources/mediation`'.

`directory`: The directory where the event files are located. This is an optional parameter with the same default as the previous one.

`suffix`: This is the last part of the file name to be included for processing (also known as 'file extension'). In the previous illustration we see the 'csv' suffix. This means that only files ending on 'csv' will be taken for processing. You can use 'ALL' as a value to indicate that there should not be any filtering, all files located in the directory should be processed. This is an optional parameter, it defaults to 'ALL'.

`rename`: This is a boolean parameter, possible values are 'true' or 'false'. If 'true', the mediation process will rename a file after it has been fully processed, appending the tag '.done' to its original name. This works well In combination with the 'suffix' parameter to avoid processing the same file twice. This is an optional parameter, it defaults to 'false'.

`dateFormat`: This is the format that the reader expects to find date values in the file. When a field is a date, the reader will parse it so it can be later handled as a date (rather that a string of characters). You will be able to tell the reader which fields are dates in the format file. The date format has to comply with the patterns set by the standard Java class 'SimpleDateFormat', these are described here. This is an optional field, it defaults to '`yyyyMMdd-HHmmss`'.

The `SeparatorFileReader` plug-in has an additional parameter:

`separator`: The character or characters the separate each field. This is an optional parameter, it defaults to comma (',').

`batch_size`: The number of records to read at a time, for group processing. This greatly affects the performance of the process, see the Performance section for more information.

`removeQuote`: If the reader should remove surrounding quotes from each field. This would allow the content of fields to have the separator character as content.

`autoID`: If true, the reader will assign a value to the key field for every record.

`buffer_size`: The number or characters that the reader will use to fill its buffer every time it access a file. This is a performance parameter, see the Performance section for more details.

The JDBC reader plug-ins

To use the JDBC reader plug-ins, make sure the JDBC driver for your database is copied to the `jbilling/lib` directory.

Record marking method

By default, the 'last id read' marking method is used, which saves the last id processed by the last process run in the preference `MEDIATION_JDBC_READER_LAST_ID (47)`. It will only read records with ids greater than this value.

If the `timestamp_column_name` plug-in parameter is set (or the default `jbilling_timestamp` column is found in the table), the timestamp marking method is used, which only reads records that haven't been timestamped and timestamps them after they are read. The advantage of this method is that it allows composite primary keys, but a possible disadvantage is that it must update the table.

Plug-in parameters

The following parameters are in common to both readers:

`batch_size`: The number of records to read at a time, for group processing. This greatly affects the performance of the process, see the Performance section for more information.

`database_name`: The name of the database. Default is `jbilling_cdr`.

`table_name`: The name of the table to read the records from. Default is `cdr`.

`key_column_name`: Allows a single key column to be specified. Default is to automatically use the primary key from the table's metadata. Should this fail, it defaults to `id`.

`username`: The username for connecting to the database. Default is `SA`.

`password`: The password for connecting to the database. Default is an empty string.

`order_by`: Allows columns to be specified for the `ORDER_BY` clause in the SQL that selects the records for processing. This is important for the 'last id read' marking method, which assumes the last id is the highest id. Default is to order by primary key ascending.

`timestamp_column_name`: Name of the column used for timestamping columns. Also indicates to the reader that the timestamp record marking method should be used. Default is to check for a `jbilling_timestamp` column, and if found, the timestamp record marking method is used, otherwise the 'last id' method is used.

`where_append`: Allows appending SQL to the `WHERE` clause of the SQL that selects the records for processing. Mainly used for testing purposes.

`JDBCReader` specific parameters:

`driver`: The JDBC driver string. Defaults to the HSQLDB driver string.

`url`: The JDBC URL string. Providing a value for this parameter causes `database_name` parameter to have no effect. Defaults to an HSQLDB URL for a test database found in the `jbilling/resources/mediation` directory.

`MySQLReader` specific parameters:

`host`: The MySQL server hostname or IP address. Default is `127.0.0.1`

`port`: The MySQL server port. Default is `3306`

Creating you own reader plug-in

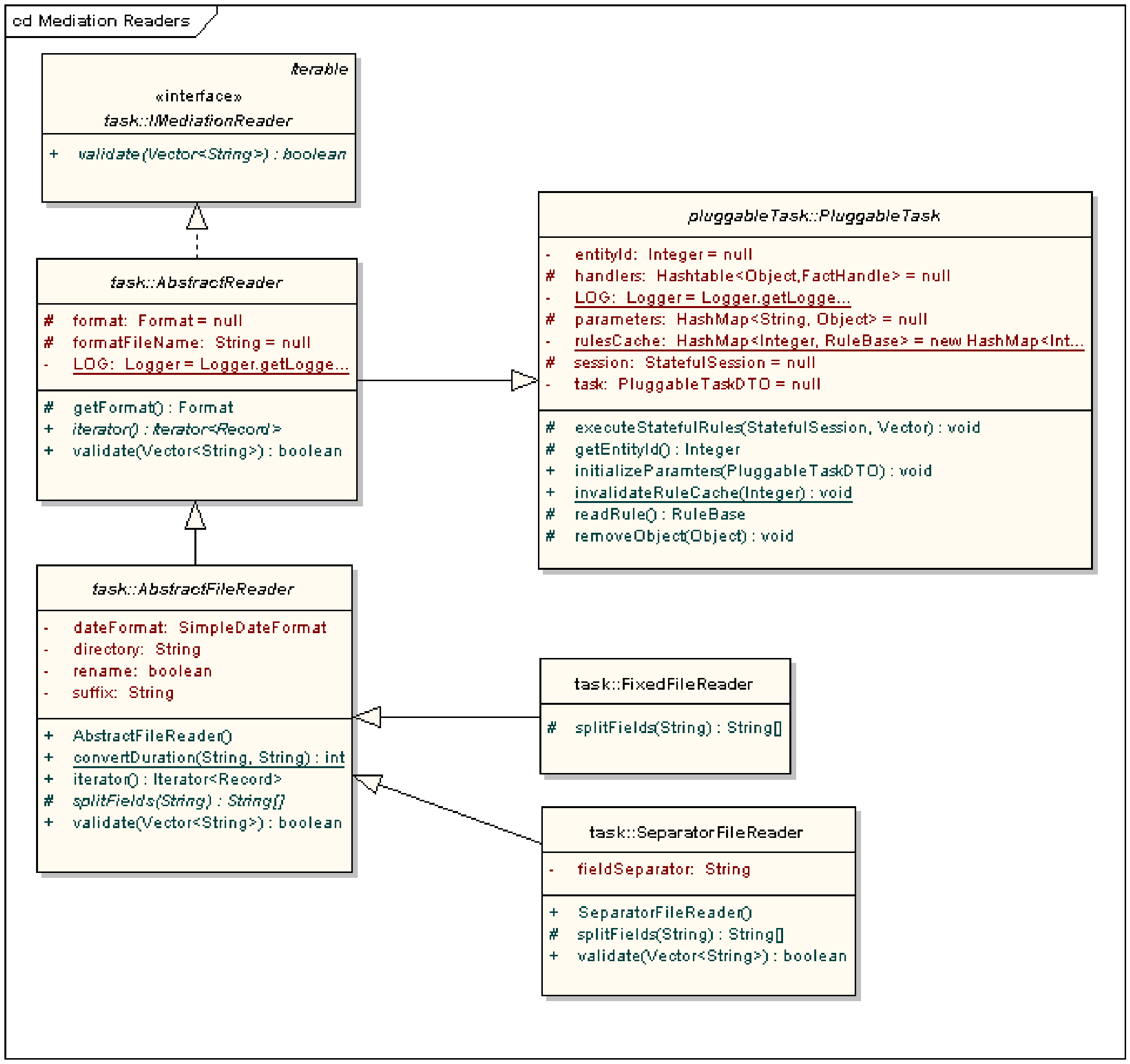
Readers are business rules plug-ins. This is a design decision meant to facilitate creating new readers without having to modify **jBilling**. You can extend some of the existing reader plug-ins or create your own to satisfy requirements that go beyond what the default readers do.

__jBilling's__ business rules plug-in architecture is covered in detail in the 'Extension Guide' document

Examples of situations where you would need you own reader are: your files are not plain text, they do need some additional processing before the records are readable: decryption or decompression for example. Another possible scenario is when your records are not in a file, but in some other archive like a relational database.

Note that none of the previous examples mention the format of the file. If what you need is a reader for a text file for a particular format of the records (number of fields, location of the fields, etc) then you do not need to develop your own reader plug-in. You just need to provide the record format to the existing readers. This is explained in the next chapter.

If you do have to develop you own reader plug-in, doing it is pretty simple, but like any plug-in development, it does require Java programming. The plug-in category is 15, for the interface '`MediationReader`'.


cd Mediation Readers
Iterable
«interface»
task::IMediationReader
+ validate(Vector<String>) : boolean
pluggableTask::PluggableTask
- entityId: Integer = null
# handlers: Hashtable<Object,FactHandle> = null
- LOG: Logger = Logger.getLogge...
# parameters: HashMap<String, Object> = null
- rulesCache: HashMap<Integer, RuleBase> = new HashMap<Int...
# session: StatefulSession = null
- task: PluggableTaskDTO = null
# executeStatefulRules(StatefulSession, Vector) : void
# getEntityId() : Integer
+ initializeParamters(PluggableTaskDTO) : void
+ invalidateRuleCache(Integer) : void
# readRule() : RuleBase
# removeObject(Object) : void
task::AbstractReader
# format: Format = null
# formatFileName: String = null
- LOG: Logger = Logger.getLogge...
# getFormat() : Format
+ iterator() : Iterator<Record>
+ validate(Vector<String>) : boolean
task::AbstractFileReader
- dateFormat: SimpleDateFormat
- directory: String
- rename: boolean
- suffix: String
+ AbstractFileReader()
+ convertDuration(String, String) : int
+ iterator() : Iterator<Record>
# splitFields(String) : String[]
+ validate(Vector<String>) : boolean
task::FixedFileReader
# splitFields(String) : String[]
task::SeparatorFileReader
- fieldSeparator: String
+ SeparatorFileReader()
# splitFields(String) : String[]
+ validate(Vector<String>) : boolean
Illustration 8: The mediation readers plug-in classes

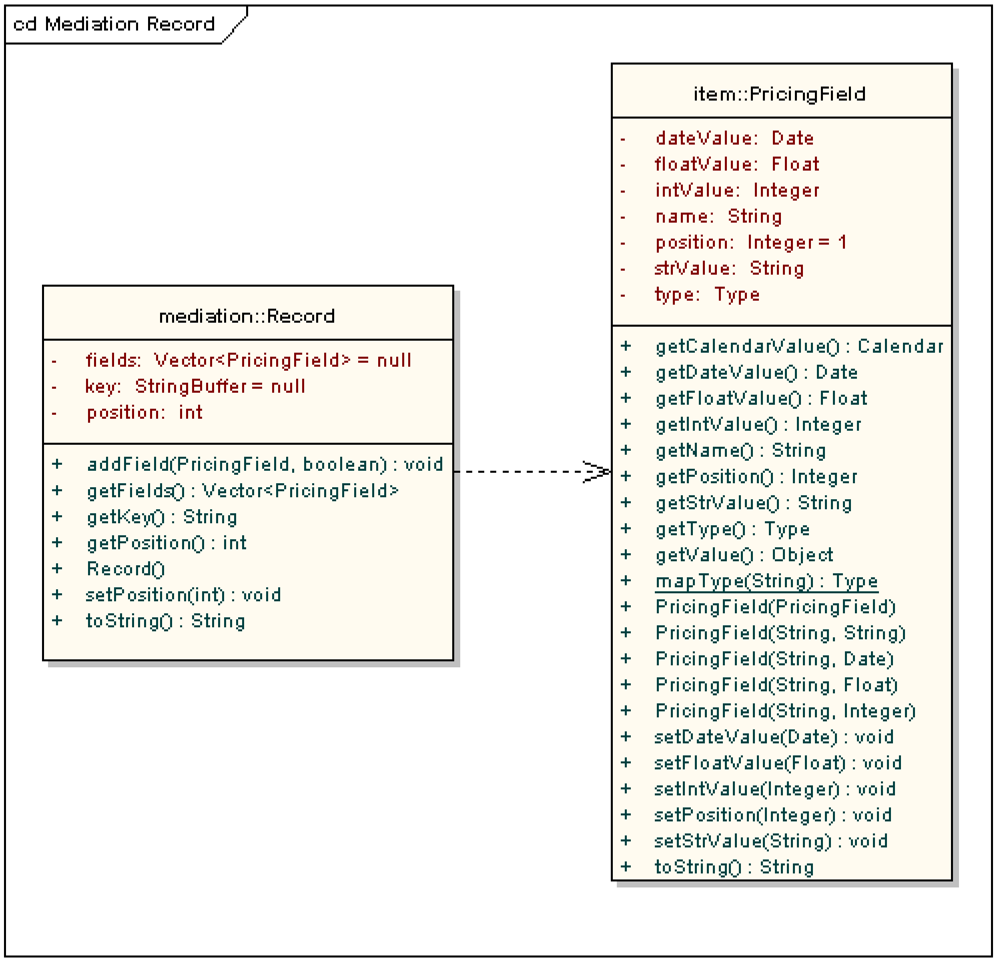
Reader plug-in responsibilities

Like any business rule plug-in, a reader has to implement a Java interface provided by **jBilling** and extend the class '`PluggableTask`'. For this category, there are two methods that your plug-in needs to implement: record reading and validation.

Record reading comes from the fact that the interface `IMediationReader` actually extends the standard Java interface `Iterable`. Thus, you need to provide an implementation of the method 'iterator' which returns an object `Iterator`. There isn't much to be said about this object that is not already well documented in the standard Java documentation. This object represents an iterator, like a cursor that reader records from a source.

The iterator will iterate through objects type `Record`. This is a **jBilling** class, it holds one record with all the fields.

Here is the core of what a reader plug-in has to do: read a record from the events source and create a `Record` object with the information extracted from such source. Each field has to be encapsulated in a `PricingField` object, which is also a very simple object. When you read a field from a record, add it to the record by calling `Record.addField()`.



cd Mediation Record
mediation::Record
- fields: Vector<PricingField> = null
- key: StringBuffer = null
- position: int
+ addField(PricingField, boolean) : void
+ getFields() : Vector<PricingField>
+ getKey() : String
+ getPosition() : int
+ Record()
+ setPosition(int) : void
+ toString() : String
item::PricingField
- dateValue: Date
- floatValue: Float
- intValue: Integer
- name: String
- position: Integer = 1
- strValue: String
- type: Type
+ getCalendarValue() : Calendar
+ getDateValue() : Date
+ getFloatValue() : Float
+ getIntValue() : Integer
+ getName() : String
+ getPosition() : Integer
+ getStrValue() : String
+ getType() : Type
+ getValue() : Object
+ mapType(String) : Type
+ PricingField(PricingField)
+ PricingField(String, String)
+ PricingField(String, Date)
+ PricingField(String, Float)
+ PricingField(String, Integer)
+ setDateValue(Date) : void
+ setFloatValue(Float) : void
+ setIntValue(Integer) : void
+ setPosition(Integer) : void
+ setStrValue(String) : void
+ toString() : String

Illustration 9: A record object and fields

In fact, it will do this in groups. It will read a group of records and return it as a `List`. This is not a requirement, you can always read only one record, put it in an `ArrayList`, and return that. As a good practice, it is best if your plug-in respects the value of the parameter '`batchSize`'. Any class that extends `AbstractReader` has access to this parameter. It signals to the plug-in the expected number of records it should read in one iteration, its purpose is mostly performance optimization.

Another function your plug-in is responsible is validation. The scope of the validation is typically just the parameters of the plug-in: are all the required parameters present? Of course, you can validate anything you want here. If there are one or more errors, simply add the error messages to the vector you are getting as a parameter for this method and return false.

Code re-use

The existing plug-in readers might not do all you need a reader to do, but they might do some useful things already. Before starting your own reader from scratch (directly implementing `MediationReader`), it is a good idea to make sure you won't be writing code that is already there.

If you won't be reading a file or from a database: If your records are in some other type of storage that is not plain text files, you might want to still use the format service provided by the class `AbstractReader`. This means that you can extend this class and add the actual reading capabilities to to it. You will be able to call 'getFormat' and get a format object created from the format XML file. This will make your reader plug-in much more flexible that if you hard-code the format into the reader.

If you will be reading a file: The class `AbstractFileReader` reads records from a plain-text file, leaving the parsing of a record (dividing the record on many different fields) unimplemented. If all you need is a different way to break a record into many fields, you can extend this class and simply implement the method `splitFields`.

Take a look to the plug-in `SeparatorFileReader`. It is very simple, because all the work is done by `AbstractFileReader`. Breaking the record into many fields is very easy in this implementation.

If you will be reading from a database: `JDBCReader` provides a number of extension points. For example, overriding the `getQueryString()` method allows a custom SQL query to be used for selecting records to process. Overriding the `recordRead()` and/or `recordProcessed()` methods could allow a different record marking methods to be implemented.

Chapter 5

The record format

XML format description

The reader plug-ins provided with the standard **jBilling** distribution can read files in any format. This is possible because the description of the format is in an XML file, rather than being hard-coded into the plug-in. Instead of having one plug-in per switch, we can have one plug-in for all the switches that write events in a file with the fields separated by a character. The actual format of the records will be passed to the plug-in as a parameter, the same applies to the characters that acts as field separator.

The format file starts and end with a <format> tag. Inside this block, you can only have <field> sections, where each field is described. The valid tags that you can use within a 'field' block are:

name: The name of the field. This should be unique (two fields should not have the same name), and will be the way to identify the field later what using mediation and pricing rules.

type: The data type of this field. Valid types are 'string', 'integer', 'float' and 'date'. For date formats, the pattern specified as a plug-in parameter will be used to convert the initial string to a Date object. See the 'dateFormat' parameter described in the 'Readers' chapter.

startPosition: This is only valid for fixed-length formats. It is the starting position of a field, starting with '1'.

length: This is only valid for fixed-length formats. It is the number of characters this field takes starting from 'startPosition'.

durationFormat: When a field represents a unit of time (seconds, minutes or hours), you might want to handle this value in seconds. **jBilling** can convert the field to seconds if you specify the format: use H for hours, M for minutes and S for seconds. Example: 013059, with format HHMMSS will be 1 hour 30 minutes and 59 seconds. The value of the field will be 5459 seconds. Use integer as type for this field.

isKey: If this tag is present (without content), it means that this field is part of the record's key. The record key is composed of all the fields that have been 'flagged' with this tag. **jBilling** will check before processing a record if the key has been already processed. This will prevent processing the same record more than once. It will also group all the records that are in sequence and share the same key, treating them as a single record.

The record key

The key of a record is one or more of its fields. This key has to be unique: two records can not have the same key. You have to define which field(s) make the key using the 'isKey' tag of the format file.

jBilling uses this key for several tasks. The first is to group records. Grouping records can result in many lines in the events file to be treated as one single record. If two or more consecutive lines in the files share the same key, they will be grouped into one record. Let's assume a file has the first field as a key, the rest of the fields are data:

```
00001,33,50,20080205-101010

00002,33,1550,20080207-101010

00002,33,1400,20080215-101010

00003,33,2000,20080225-101010
```

In the above example, there are three records and four lines. The second record is made out of two lines because of the grouping. Note that for grouping to happen, the lines with the same key have to be together, one after the other.

The other important function of the record key is to check for duplicates. **jBilling** will store in a table the keys of all the records already processed. After reading a record but before processing it, **jBilling** will check against the table of keys to see if the current record's key is present. If so, it will skip the record because it is considered a duplicate.

Let's see again the last example with just a simple change in the line's order:

```
00001,33,50,20080205-101010

00002,33,1550,20080207-101010

00003,33,2000,20080225-101010

00002,33,1400,20080215-101010
```

Now the results will be quite different: there are also three records, but they are all made from one line only. The fourth line will be ignored, considered a duplicate. It has the same key as the second line, and it is not placed right after it. The fact that there is another line separating the two with the same id makes all the difference.

Purging the keys table

As mentioned before, there is the database table `MEDIATION_RECORD` in the **jBilling** schema that holds every key processed:

```
MEDIATION_RECORD

      ID_KEY             VARCHAR     100

      START_DATETIME     TIMESTAMP

      END_DATETIME       TIMESTAMP

      MEDIATION_PROCESS_ID           INTEGER

      MEDIATION_RECORD   OPTLOCK
```

This table can grow too big, holding data that is too old to be useful. It is a good practice to establish a policy on when to delete records from this table. It is up to you to setup the script and the method to run it periodically. The SQL statement to delete the records will look like:

```
DELETE FROM MEDIATION_RECORD WHERE START_DATETIME < today – 30 days;
```

The above pseudo-SQL script would delete all the records that are older than 30 days. How old is too old is for you to decide.

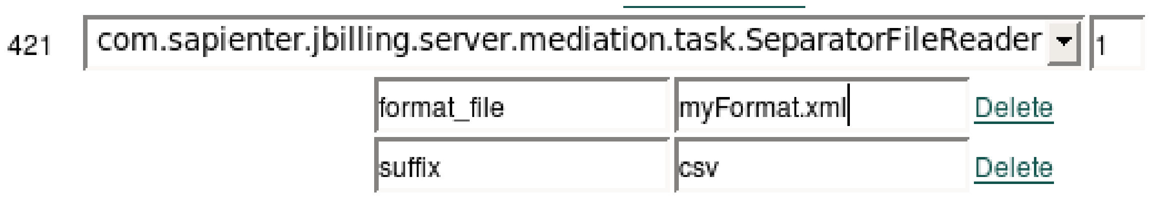
Example

The following is a simple example, involving records with only four fields. These records are usage records for bandwidth, we have in each event the total gigabyte consumption. Let's start with a format that is comma separated:

```
<format>

    <field>

      <name>record_id</name>

      <type>string</type>
```

```
          <isKey/>
        </field>
        <field>
          <name>user_id</name>
          <type>integer</type>
        </field>
        <field>
          <name>total_gb</name>
          <type>integer</type>
        </field>
        <field>
          <name>use_date</name>
          <type>date</type>
        </field>
    </format>
```

Let's assume that this file name is myFormat.xml and its location is the default: `jbilling/resources/mediation`. This file will be describing the format of an event file that has the fields separated by ','. The reader plug-in configuration will look like this:



421 com.sapienter.jbilling.server.mediation.task.SeparatorFileReader 1

format_file myFormat.xml Delete

suffix csv Delete

Illustration 10: The format file plug-in parameter

A file with data that follows the example format will look like this:

```
00001,33,50,20080205-101010
00002,33,1550,20080207-101010
00003,33,1400,20080215-101010
00004,33,2000,20080225-101010
```

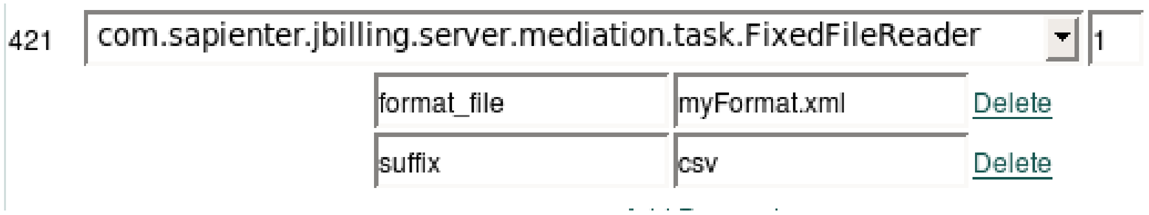
The above records show the usage of a total of 5000 GB from user 33. The name of this file would have to end with 'csv' to be picked up by the reader, since this is the suffix we added as a parameter earlier.

Let's now do the same exercise, but considering that the record files have fixed-length fields. The XML format file would look like this:

```
<format>
    <field>
      <name>record_id</name>
      <type>string</type>
```

```
          <startPosition>1</startPosition>
          <length>5</length>
          <isKey/>
        </field>
        <field>
          <name>user_id</name>
          <type>integer</type>
          <startPosition>6</startPosition>
          <length>2</length>
        </field>
        <field>
          <name>total_gb</name>
          <type>integer</type>
          <startPosition>8</startPosition>
          <length>4</length>
        </field>
        <field>
          <name>use_date</name>
          <type>date</type>
          <startPosition>12</startPosition>
          <length>15</length>
        </field>
    </format>
```

We had to specify for each field where does the field starts and how many characters it takes. The plug-in configuration uses the fixed-length class:



421 com.sapienter.jbilling.server.mediation.task.FixedFileReader 1

format_file	myFormat.xml	Delete
suffix	csv	Delete

Illustration 11: Using a fixed-length plug-in

The same records we saw before will look different in the text files. The data is the same, only the format changes:

```
0000133005020080205-101010
0000233155020080207-101010
0000333140020080215-101010
0000433200020080225-101010
```

You are right if you think that the suffix of the file should be other than 'csv'. For clarity, this is correct, CSV indicates comma separated values. The plug-in does not know this, so the example will work anyway.

Chapter 6
Processing with rules

Overview

At one point in the mediation processing (see step 8 in the initial overview diagram), the record is ready to be processed. The reader provided the records from the event storage, now something needs to be done with these records, we need to give meaning to all the data coming as fields in each record.

The Mediation Process plug-in based on the interface `IMediationProcess`. The default implementation of this plug-in is rules-based. This means that the actual logic of what the plug-in does is externalized to business rules managed by the Drools rules engine. To understand this section you will have to have a good understanding of rules processing.

To learn more about rules and rule-based plug-ins, take a look to the Extension Guide document.

The mediation plug-in makes use of three different types of rules:

1. Mediation rules: make the basic interpretation of the raw data coming from the reader
2. Item management rules: Deals with plan, bundles and rules like: “The first 100 units are free, the next are not”
3. Pricing rules: Assigns a price to those items that can need a different price based on some data present in the CDR. The best example is long distance calls.

Mediation rules

The plug-in takes as input the record object (type `Record`, see the class diagram in the previous chapter). From this input, its is responsible for:

- Returning one or more *order lines* with the items and quantities of what the customer is buying. This is a translation, from event record to actual items and quantities.
- Determine the jBilling *user*. The event represents an action done by a user. In other words, which customer orders need to be updated?
- Determine the *currency*. Since the event is a purchase, jBilling needs to know what is the currency of the purchase.
- Determine the *date* of the event.

As you can see, the responsibilities of this component are the very core of the mediation process. It takes a record without any meaning (just conveniently converted to Java objects by the readers) and from the record data it has to *convert* the record into a purchase that the billing system understands. The plug-in will not update any orders on any of the customer's information, but it will make available all the necessary data for the mediation process to do this.

Item management rules

These rules are typically in a different package. This is, you have group them together in the BRMS or in a separate file. Then you can use this package from the item management plug-in (`RulesItemManager2`) and from the mediation plug-in. This technique is useful when you need the same rules affecting both the mediation process and other areas of the system without having the rule repeated in many packages.

Item management rules affect how the items relate to each other. They can be used to group items, replace an item with another one and other actions that are needed to configure plans and bundles.

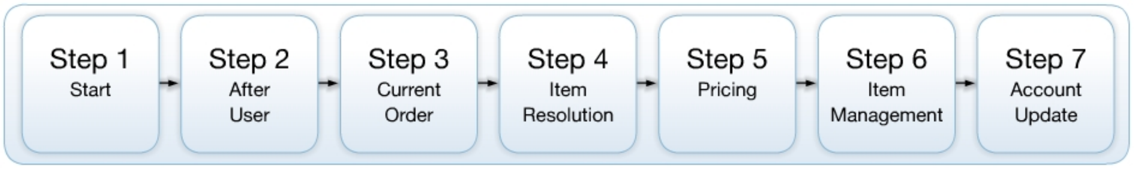
Pricing rules

Like the previous type, this rules are usually in a separate package of rules. Then you can call them from the pricing plug-in `RulesPricingTask2` as well as from the mediation plug-in.

With these rules, you will change the price of an item. This is very important when the default price is not valid for the record being processed.

Steps

For a record to be fully processed, many steps have to take place. Starting with assigning a user to the event, to the update of the user's account. Some steps involve many actions:

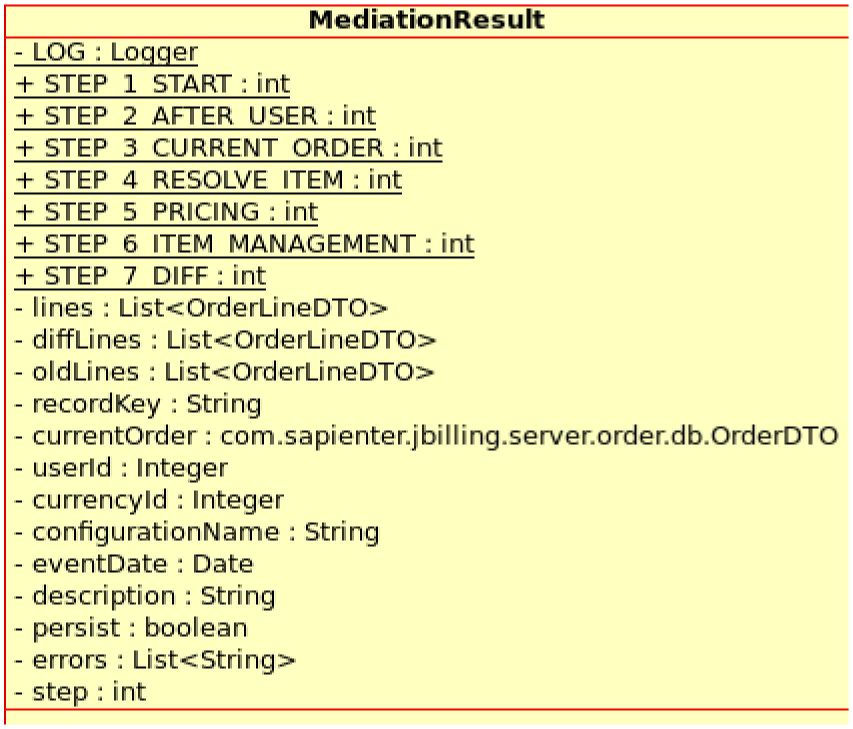


1. **Start**: The process starts by determining the user and date. Additional rules that check for errors in the record belong to this step as well.
2. **After user**: The user is known now. The currency for this event is set.
3. **Current order**: The current order for the user and the date of the event is fetched.
4. **Item resolution**: The event is now mapped to a jBilling item (product). A description of the event is also given to later list the event in the invoice details section.
5. **Pricing**: If the event can not use the default item price, then rules that change this price are triggered in this step.
6. **Item management**: Plan an bundles rules.
7. **Account Update**: The last step takes care of updating the customer's account with the charges resulting from the event.

It is rare that you will need to actively write rules for each step. In most cases you will only use the rules that jBilling provides and add some only for a few steps. Still, it is good to be familiar with all the steps.

The MediationResult object

The way **jBilling** keeps track of the results for each record is through the `MediationResult` object. There will be one instance of this object for each record that needs to be processed in the memory context:



```
MediationResult
- LOG : Logger
+ STEP_1_START : int
+ STEP_2_AFTER_USER : int
+ STEP_3_CURRENT_ORDER : int
+ STEP_4_RESOLVE_ITEM : int
+ STEP_5_PRICING : int
+ STEP_6_ITEM_MANAGEMENT : int
+ STEP_7_DIFF : int
- lines : List<OrderLineDTO>
- diffLines : List<OrderLineDTO>
- oldLines : List<OrderLineDTO>
- recordKey : String
- currentOrder : com.sapienter.jbilling.server.order.db.OrderDTO
- userId : Integer
- currencyId : Integer
- configurationName : String
- eventDate : Date
- description : String
- persist : boolean
- errors : List<String>
- step : int
```

A `MediationResult` object knows then the current step this record is in, and any information that has been collected so far. For example, if a record is in step 2, then we can assume that the `userId` is set.

Most of the rules interact heavily with this object. You don't need to know all its fields, but it is best if you are familiar with most of them.

Step 1: Start

This is the first step. The result record is mostly empty and the goal is to set the user and date.

Input State

The result record is populated only with the data that is known to the system from the very beginning. Every other field is empty: they will be set by the rules.

- `recordKey`: The unique identifier of the record. This was set by the reader plu-in.
- `configurationName`: The name of the configuration mediation. This will be helpful when you have many different configurations: SMS, voice, data, etc..
- `persist`: a boolean flag to indicate if this record is a 'real' record that needs to be persisted or not. This would be false in scenarios where you are calling the API method '`validatePurchase`' which use the mediation module to resolve a record but they should not alter the customer's account. This translates to 'can the customer buy this', rather than 'this customer bought this'. The first case should have persist = false, the second one true.

Output state

- `userId`: set with the ID of the user that the event belongs to

- `eventDate`: set with the date and time that this event took place.

Things to do

Setting the user ID:

You can find the user ID with any of the core classes of jBilling. This function find the user ID from a user name:

```
function Integer getUserIdFromUsername(String username, Integer
      entityId) {
    UserBL user = new UserBL(username, entityId);
    return user.getEntity() != null ? User.getEntity().getUserId() :
null;
}
```

The classes you want to take a look at are UserBL and UserDAS().

Here there is an example rule that uses the above drools function to set the user id based on the CDR field 'userField':

```
rule 'user setter'
no-loop  #because the user might be wrong and the set gets a null
when
    $result : MediationResult(step == MediationResult.STEP_1_START,
userId == null)
    $field : PricingField( name == "userfield", resultId ==
$result.id)
    $company : CompanyDTO( ) # needed to determine a user by its
user name
then
    modify( $result ) {
        setUserId( getUserIdFromUsername($field.getStrValue(),
$company.getId()) );
    }
end
```

Setting the Date:

You need to find out the date of this event. This is typically easy, here it is an example based on the 'start' CDR field:

```
rule 'date setter'
when
    $result : MediationResult(step == MediationResult.STEP_1_START,
eventDate == null)
    $field : PricingField( name == "start", resultId == $result.id)
then
    modify( $result ) {
        setEventDate( $field.getDateValue() );
    }
end
```

Note how we get a data type date out of the field just by calling the right getter.

Remove any undesired record:

If there are conditions that would make a record not processable (an error, for example), it is best to deal with them early in the process. Step 1 is the best place.

The following example will finish the processing of calls that are not answered:

```
rule 'cancel not answered call'
when
    $result : MediationResult(step == MediationResult.STEP_1_START,
done == false )
    $field : PricingField( name == "disposition", resultId ==
$result.id, value != "ANSWERED" )
then
    $result.setDone(true);
    retract($result);
end
```

The most important part is that the record is set as 'done', and then removed (retracted) from the memory context.

Transition rule

Once the user and the date are resolved, we can move on to the next step. It is good to be aware of this type or rules, but most probably you won't need to modify them:

```
rule 'from start to after user'
when
    $result : MediationResult(step == MediationResult.STEP_1_START,
userId != null, eventDate != null, currencyId == null)
    # only one record for a given user at a time
    not( exists( MediationResult( $result.userId == userId, step >
MediationResult.STEP_1_START) ) )
then
    modify( $result ) {
        setStep(MediationResult.STEP_2_AFTER_USER);
    }
end
```

Note that there is specific logic to avoid having two records for the same customer being processed at the same time. This is usually needed to avoid miscalculations in the 'diffs' section (the calculation of how much the customer account was modified by the current record).

Step 2: After User

Now that we know to whom this record belongs and when it was done, we can get to do any other step that requires them. By default, this is only to set the currency.

Input State

- `userId`: set with the ID of the user that the event belongs to
- `eventDate`: set with the date and time that this event took place.

Output state

- `currencyId`: the currency that this event will billed on.

Things to do

Setting the currency

You probably want to use for this event the currency that the customer uses. This is a simple function that does that:

```
function int getDefaultCurrency(Integer userId) {
    return new UserBL(userId).getCurrencyId();
}
```

Here's an example rule setting the currency with the above function:

```
rule 'currency setter'
when
    $result : MediationResult(step ==
      MediationResult.STEP_2_AFTER_USER, currencyId == null)
then
    modify( $result ) {
        setCurrencyId( getDefaultCurrency($result.getUserId()) );
    }
end
```

You could be setting the currency using any other criteria, like the location where the event happened for example. The above is only a simple example, but a very typical implementation.

Transition rule

If all you are doing in this step is to set the currency, this is how the transition rules looks like:

```
rule "from after user to current order"
when
    $result : MediationResult(step ==
      MediationResult.STEP_2_AFTER_USER, currencyId != null,
currentOrder == null)
then
    modify( $result ) {
        setStep(MediationResult.STEP_3_CURRENT_ORDER);
    }
end
```

Step 3: Current Order

Fetching the current order is typically a step you can ignore. You can consider it an 'internal' step.

The mediation process now needs to know which one time order to use when adding the charges that will be coming from this event. Take a look to the section “Updating orders” for more information on this.

Input State

- `currencyId`: the currency that this event will billed on.

Output state

- `currentOrder`: The order that will receive the one time charges coming from this event.

Things to do

Setting the current order

Finding and setting the current order is very simple:

```
rule 'get current order'
when
    $result : MediationResult(step ==
MediationResult.STEP_3_CURRENT_ORDER, currentOrder == null)
then
    modify( $result ) {
        setCurrentOrder( OrderBL.getOrCreateCurrentOrder($result.get
UserId(),
                $result.getEventDate(), $result.getCurrencyId(),
$result.getPersist()) );
    }
end
```

You will rarely have to modify the above rules, but as usual, it is good to be aware of it.

Transition rule

Here the rule that takes care of transitioning from step 3 to step 4:

```
rule "from current order to resolve item"
when
    $result : MediationResult(step ==
      MediationResult.STEP_3_CURRENT_ORDER,  currentOrder != null,
done == false )
then
    modify( $result ) {
        setStep(MediationResult.STEP_4_RESOLVE_ITEM);
    }
end
```

Step 4: Resolve Product (Item)

Resolving the product is the very core of the mediation process. Here it is where you *map a CDR to a billable service*. While the other steps in the mediation process needed little of your work (if at all any), this step will be the one where you provide the actual logic that the system needs to follow.

Input State

- `currentOrder`: The order that will receive the one time charges coming from this event.

Output state

- `lines`: The list needs to get new order lines. Each new order line will have a product (item) and a quantity. This the core of what the mediation process is meant to do: map an event to a product, which is represented as an order line in jBilling.
- `step`: Rather than using a transition rules, the step is typically set in the 'then' section of the rule to step 5.
- New pricing request: This is not a variable to set in the result object. This is a new pricing request that you might need to do by inserting an object in the memory context.
- `description`: an explanation of the charges coming from this event (optional).

Things to do

Add a new order line

The result object has a list of order lines that capture products that the customer is buying because of this event . This means that you need to create an instance of the object `OrderLineDTO`() populated with the information of the purchase. The following is an example (or helper) function:

```
function OrderLineDTO newLine(Integer itemId, BigDecimal quantity) {
    OrderLineDTO line =  new OrderLineDTO();
    line.setItemId(itemId);
    line.setQuantity(quantity);
    line.setDefaults();
    return line;
}
```

An event typically maps to one product, but it does not have to. That is why the '`lines`' variable of the record class is a List.

Let's use the above function in an example rule that uses the 'duration' field from the CDR to product (item) id 2900:

```
rule 'resolve call item and request price'
when
    $result : MediationResult(step ==
      MediationResult.STEP_4_RESOLVE_ITEM)
    $quantity : PricingField( name == "duration",
      resultId == $result.id)
then
    $result.getLines().add(newLine(2900,
      new BigDecimal($quantity.getStrValue())));
    $result.setStep(MediationResult.STEP_5_PRICING);
    update( $result );

    PricingResult request = priceRequest(2800, $result); // because
it will be converted to this...
    insert( request );
end
```

The important part of this rules is that the quantity is coming directly from the value of the 'duration' field of the CDR. The rules assumes that every CDR that made it to this step maps to the same product (2900). It would be very easy to add more conditions based on other fields for conditionally map to other products.

In this example, product 2900 maps to a generic phone call. This item will be later evaluated and swapped to another item depending on the custom begin a subscriber (or not) of a specific plan (more on this later).

Regardless, this rule makes a explicit pricing request for the product 2800. The logic is: if the customer is a subscriber, it will be free so the price is irrelevant. But if she is not a subscriber, then the product will be the 2800 and the pricing needs to be done based on a rate card.

Requesting a price is as simple as adding an instance of `PricingRequest.` More on this in the rating section.

Set the description

It will make it much easier for the customer if an event that translates to a charge is explained. The typical way is by having a section of the invoice with a list of events with the description, date and price.

jBilling does keep track of every event processed, and it can add a description to them. This is later used for the invoice presentation just described.

This is an example:

```
  rule 'resolve call destination'
  when
      $result : MediationResult(step ==
  MediationResult.STEP_4_RESOLVE_ITEM, description == null )
      $phoneCalled : PricingField( name == "dst", resultId ==
  $result.id)
  then
      # set mediation event description to the call destination
      modify( $result ) {
          setDescription("Phone call to " +
              $phoneCalled.getStrValue());
      }
  end
```

Transition rule

In this step, the transition rule does need some thought. One factor is that not all CDRs will be needing a pricing step, in some cases the default price of a product applies. Another factor is that you might want pricing rules to be used not only from the mediation process, but from other areas of the system (like calls to the API when creating an order).

This means that the pricing rules will not use the `MediationResult` object at all. They will run when the `PricingRequest` is detected and largely determine the order using the salience (a drools attribute).

Last but not least, if there is only one rule doing mapping to a product id (like the example above) then it is simple and easy to just set the next step right there. But this is not an option if there are many rules doing mapping.

Let's see a simple transition example:

```
rule "from resolve item to pricing"
when
    $result : MediationResult(step ==
MediationResult.STEP_4_RESOLVE_ITEM, description != null )
    PricingResult( price == null, pricingFieldsResultId ==
$result.id )
then
    modify( $result ) {
        setStep(MediationResult.STEP_5_PRICING);
    }
end
```

Step 5: Pricing

This is an optional step, only needed if the default price for the product (item) needs to be changed depending on the data coming from the CDR.

This step is special, because pricing is done in other parts of jBilling, not just when the mediation happens. And since the object `MediationResult` is only for mediation, the pricing step is usually not dependent on this object.

Pricing is done using the `PricingResult` object. This is described in the 'Rating' section of this guide.

You will need to request a price by adding an instance of `PricingResult` to the memory context (done in the previous step), and then take the price from the object to change the price of the order line that you added in the previous step.

Input State

- Not applicable: `MediationResult` is not involved.

Output state

- Not applicable: `MediationResult` is not involved.

Things to do

Assign the new price

The pricing rules should be running with a higher salience (priority) than the rule that assigns the price for it to work well. This is an example:

```
rule 'price assignment'
```

```
when
    $result : MediationResult(step <
      MediationResult.STEP_6_ITEM_MANAGEMENT)
    $price : PricingResult( pricingFieldsResultId == $result.id,
      price != null )
    $line : OrderLineDTO( itemId == $price.itemId) from
      $result.lines
then
    $line.setPrice( $price.getPrice() );
    update( $result );
end
```

Transition rule

The transition is very simple, and it mostly relies in the salience of the rule:

```
rule "from pricing to item"
salience -10  # has to run after the pricing rules had a chance of
              # setting the price
when
    $result : MediationResult(step ==
      MediationResult.STEP_5_PRICING)
    PricingResult(pricingFieldsResultId == $result.id ) # probably
not needed
then
    modify( $result ) {
        setStep(MediationResult.STEP_6_ITEM_MANAGEMENT);
    }
end
```

Step 6: Item Management

This step is similar to the previous one in that the rules that do item management typically belong to a different rules package because they are also needed for actions like creating an order from the API.

However, the result object does get modified in this step. When step 6 starts, you finally have a product defined, a quantity of what was bought, and a price. So it is time to change the current order accordingly.

After this, you will let the standard item management rules that take care of bundles and plans to do their thing (see the extension guide for more on item management rules).

Input State

- `lines`: with the order lines correctly set: item id, quantity, price.

Output state

- `oldLines`: This should have the order lines of the current order as they were before any changes coming from the event being processed.
- `currentOrder`: The current order should be changed, adding a line or adding quantity to an exiting line with the charges coming from this CDR.

Things to do

Add a line to the current order

This is a function that helps adding a line to the current order. It itself relies on helper methods from jBilling objects:

```
# updates the current order
function void addLine(MediationResult result) {
    result.setOldLines(OrderLineBL.copy(result.getCurrentOrder().get
Lines()));

    if (!result.getLines().isEmpty())
        OrderLineBL.addLine(result.getCurrentOrder(),
            result.getLines().get(0), false);
}
```

And a rule that uses the function:

```
rule 'line creation'
when
    $result : MediationResult(step ==
      MediationResult.STEP_6_ITEM_MANAGEMENT, oldLines == null )
then
    addLine($result); # will add the first line to the current order
and set oldLines
    update($result);

    # to allow item management rules
    insert($result.getCurrentOrder());
end
```

The interesting part of this rule is that it inserts the current order into the memory context. This is to allow item management rules to do their work. They will be expecting a instance of `OrderDTO` to evaluate items, subscriptions, etc. They will alter this instance, swapping items, adding others, etc.

Transition rule

The transition rule needs to have a low salience, to make sure that it runs after all the item management rules (which are typically part of a package outside the mediation rules) have ran already.

```
rule "from item to diffs"
salience -10 # let all the item management rules fire first
when
    $result : MediationResult(step ==
      MediationResult.STEP_6_ITEM_MANAGEMENT, oldLines != null)
then
    modify( $result ) {
        setStep(MediationResult.STEP_7_DIFF);
    }
end
```

Step 7: Account update

The last step is to update the customer's account. At this point is also done a comparison between how the order looked like before the event and after (a 'diff'), and this information is saved as part of the history of the order.

This is a step that you won't typically have to make any changes to the standard jBilling rules. Is one of the so called 'internal rules'.

Input State

- `currentOrder`: complete with all the new charges.

Output state

- `done`: is now set to 'true'.
- `diffLines`: with the result of the comparison between the old current order and the new one.

Things to do

This last rule has a lot of responsibilities:

- Save the current order to the data base
- Set the diff lines by using a helper method from `OrderLineBL`.
- The record is done, so this field is set to true and the record is removed from the memory context along with the current order (for performance and clarity).

Let's see an example:

```
rule 'resolve diff lines'
when
    $result : MediationResult( step == MediationResult.STEP_7_DIFF)
    $company : CompanyDTO()
then
    if ($result.getPersist()) {
        new OrderDAS().save($result.getCurrentOrder());
    }
    $result.setDiffLines(OrderLineBL.diffOrderLines(
      $result.getOldLines(), $result.getCurrentOrder().getLines()));
    $result.setDone(true);
    retract($result);
    retract($result.getCurrentOrder()); #doable because we can count
with one record being process for a give user at a time
    if ($result.getPersist()) {
        new OrderBL().checkOrderLineQuantities(
            $result.getOldLines(),
            $result.getCurrentOrder().getLines(),
            $company.getId(), $result.getCurrentOrder().getId());
    }
end
```

Transforming the data

In many cases, the data coming from the CDR will not be in a consistent format or in a format that let us write rules easily. It can be that before the rules that contain the actual business logic of the mediation process run, some data transformation needs to be done.

Record splitting

It is not hard to find a situation where an event actually means more than one charge (unfortunately for customers). In general, you can deal with this by simply adding many lines (see step 4). This is easy and takes advantage of the built-in support for mapping one CDR to multiple products.

There is another method that can be more elegant from a technical perspective. You can insert a new instance of `MediationResult` into the memory context, along with the `MediationRecord` and the `PricingFields`. This is an advance technique that is out of scope for this guide, but a few pointers cat get you in the right direction.

The mediation rules do not know where the objects that are in the memory context came from. At any time, you can add new data to the memory context and the rules engine will process it. This is what we do for item management, we insert the current order in the memory context so rules that react to an instance or `OrderDTO` get activated. For pricing there is also use of this dynamic insertion into the memory context.

In short, these are the steps you need to follow to create a new CDR for the mediations rules to process:

1. Create a new `Record`
2. Create a new `MediationResult` . You need the '`recordKey`' of this result object to have the key of the record created in step 1.
3. Create new `PricingField` instances (probably many). The '`resultId`' of each of these instances need to be set with the id of the result object form step 2.
4. Add the pricing fields and the result to the memory context.

Record grouping

There is the other way too: many events coming from the external system actually mean only one thing from a billing perspective. If an record does not mean anything at all, you would just remove it as shown in the step 1 section. The problem is when two or more records hold some data that applies to just one billable event.

The mediation module has the capability of having many records in memory at the same time. The number of them depends on the reader. The standard readers will read the number of records given a specific parameter. In most cases, this means that you will need to use a custom plug-in that extends one of the current readers and overrides the batch size logic.

The goal is to be sure that the memory context will have *at the same time* the records that need to be merged. Once this is done, the merging rule is not different than the rules reviewed in the 'Steps' section: you will check for conditions that happen in more than one record and perform an action.

Padding and other field modifications

The data that is coming from the event record might need some tweaks to facilitate writing rules. For example, the phone number that is being called might come with a country code if the phone is long distance. The problem is that if the phone call is a local one, the telephony system doesn't store the event with any country code.

For writing pricing rules, it will be very convenient if you can rely on a fixed-length destination phone number that has all the codes: country, area, etc.

A simple rule can take care of this:

```
when
   $field : PricingField(name=="dst", eval(strValue.length()==10))
then
   $field.setStrValue( "0111" + $field.getStrValue() );
```

This rules translates as: if the 'dst' field (which holds the destination number) is 10 digits long, then add a “0111” string in front of it.

The best place for this type of rule is step 2. This is, after the basics are covered and maybe the record filtered out altogether, but before any actual product logic is done based on the CDR fields.

This is just a simple example, you can check and manipulate any field at will. When writing rules, you have access to all the Java string manipulation methods from the `String` class (we used 'length' in the example).

Extending the mediation module

You might need to add some custom functionality to the default mediation process. This is mostly true when the helper methods offered by the **jBilling** classes are not enough for your requirements.

You do not need add Java code if what you need to do can be done with rules, or even with Drools functions. For example, if you need to convert seconds to minutes and round the result up to the next minute (a typical procedure for wireless phone calls), you can add a function to Drools that does this for you and just call the function from your rules.

On the other hand, there are other scenarios that require plug-in extensions. A common one is when defining the user requires additional logic. Each external system might identify the user in a very different way: sometimes it is IMSI, sometimes is MSISDN or even just an IP address. To map this identifiers to a jBilling user ID, you might need to make a call to an external service, or query a database.

The key to adding new Java code to the mediation module is to keep in mind that there are no restrictions or limitations on how to do this. There isn't a specific class to extend or package where to put your classes.

As you could see from the example rules, it is very easy to call any Java class from a rule. All you need to do then is package your classes in a way that they will be part of the JVM class path, then add the import to the rule's source and use the class as you would do from normal Java code.

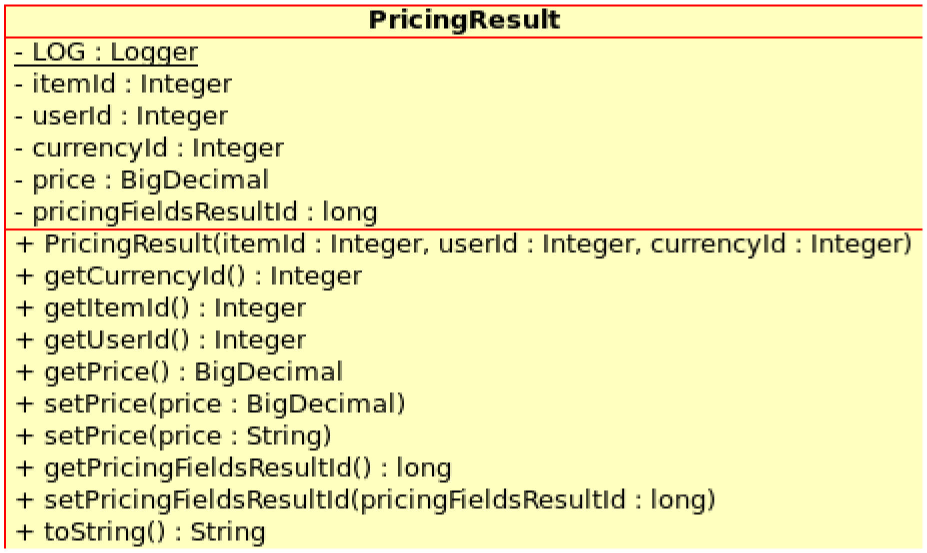
Chapter 7
Rating

Rating events

So far, in the previous chapters and examples, rating has been mostly ignored. The diagram with the mediation process overview on page 12 does not have an explicit 'rating' step. What if you need to process events that need explicit assignment of a price?

The pricing result

In **jBilling**, rating is done with rules. The rules are activated by the presence of an instance of `PricingResult`, and they typically leave the result in this object as well. Let's start by taking a look to this class:


PricingResult
- LOG : Logger
- itemId : Integer
- userId : Integer
- currencyId : Integer
- price : BigDecimal
- pricingFieldsResultId : long
+ PricingResult(itemId : Integer, userId : Integer, currencyId : Integer)
+ getCurrencyId() : Integer
+ getItemId() : Integer
+ getUserId() : Integer
+ getPrice() : BigDecimal
+ setPrice(price : BigDecimal)
+ setPrice(price : String)
+ getPricingFieldsResultId() : long
+ setPricingFieldsResultId(pricingFieldsResultId : long)
+ toString() : String

Here it is an example:

```
rule "long distance plan A price"
    when
        # Long distance call
        $result : PricingResult( itemId == 2800, price == null )
        SubscriptionResult( itemId == 2700, userId == $result.userId,
            subscribed == true ) # Long Distance Plan A
    then
        modify( $result ) {
            # Long Distance Call base price (special prices may still
            # be set via the rating card)
            setPrice("0.20");
        }
    LOG.debug("Pricing set because of plan 2700");
end
```

The rule checks that there is a pricing result object for as specific item that remains unresolved: the price is null. In this example, the condition also adds that the user is subscribed to a specific plan as well.

The 'then' section is very easy: set the price of the result.

Executing pricing rules

Depending where the pricing is needed, the pricing rules will be executed following a different path.

Pricing within the mediation process

When the mediation process runs, to get the pricing rules to run you need to manually insert a new instance of `PricingResult` in the memory context. This is explained in the Steps section.

Pricing outside the mediation process

By using the plug-in `RulesPricingTask2` you can have pricing from other areas of jBilling that are not the mediation process but need a price. For example, the list of items will show up with a price, and this price can be affected by rules if the rules plug-in is part of your configuration.

This plug-in is very simple. It creates an instance of `PricingResult`, initialized with the known information: user id, item id, etc. Then it calls the rules and returns whatever price is in the price field of the result object.

If you need the same pricing rules to be active in both the mediation process as well as the rest of jBilling, then use this plug-in. The rules should be packaged in an independent package that is used by both the mediation plug-in and the pricing plug-in.

Item relationship management plug-in

This plug-in can add, remove and swap items when an order line is added to an order. With this plug-in you can implement tiered pricing, for example. This is when the price of an item changes depending how many units are bought in a given period of time: the first 100 units are included in you monthly plan, the next 100 are price 50 cents, then next 100, 40 cents and so on.

You can do this without the pricing plug-in because *the information that affects the price is not directly in the event.* In this case, the sole factor that affects the price of the event is how many units the customer has bought so far, in this billing cycle. That is not present in any of the event fields. Actually, that is something that the billing system knows.

In this case, you can implement the rating for your mediation process with the help of only the item relationship management plug-in.

Pricing with rules

The mediation process, when updating the order in step 10, makes the call to the **jBilling** core passing along all the fields from the record that originated the order modifications. This then allows the pricing rules to have access to the record fields and act accordingly to set the price of the item that is being added to the customer's order.

When the price of an event depends on many fields from the event, then you need to use the pricing plug-in. The typical example is a phone call: the even record will have fields that affect the final price of the call: area code from where the phone originated, area code called, time of the day, day of the week, etc.

In that case, the item relationship management plug-in will be of little help to set the price: here switching or adding an item of the order doesn't do what we need. We know the item involved: a phone call, now we need to give it a price.

As mentioned earlier, the pricing rules have in its working memory all the `PricingField` objects, just like any rules of the mediation process. We can then write rules that query on those objects:

```
rule '604 -  512'
when
    $src : PricingField( name == "src", strValue matches "^604.*")
    $dst : PricingField( name == "dst", strValue matches "^512.*",
            resultId == $src.resultId)
    $result : PricingResult( itemId == 2800, price == null,
      pricingFieldsResultId == $src.resultId )
then
   modify( $result ) {
       setPrice("7");
   }
end
```

This rule is a typical pricing rule for mediation, let's go through it in detail:

There are three conditions in the rule: The area code of the originating phone number has to be '604', the area code of destination number has to be '512' and the item involved as to be the ID '2800'. If all three conditions are met, the the price will be set to seven.

Let's slice the conditions in all its parts:

    $src : **PricingField**(name == "src", strValue matches "^604.*")

There are two conditions that the field has to meet. In the drools language, that is done by simply using expressions separated by commas inside the name of the call. Record fields are represented in the working memory by `PriceField` objects.

    $src : PricingField(**name == "src"**, strValue matches "^604.*")

In our format file, the field with the source phone number, the number where the call is being made, has the name 'src' and is of type string. This example is actually taken from the asterisk CDR format definition.

    $src : PricingField(name == "src", **strValue matches "^604.*"**)

From this phone number, we are only interested in the area code. That is the first 3 digits of the field. Since this field is a string, we can use the method 'getStrValue' by treating that as a property of the object (following the java beans definition, drools will add the 'get' to the method, that why we only need to type 'strValue' in the rule).

This method will return a `String` object. The operator to do the comparison will do a regular expression. This is how we check that the first three digits have a specific value.

The rest of this rule is straight forward: the second condition is similar to the first one, it just checks that the destination number starts with '512'. The third condition is a requirement to any pricing rule: we need to limit the rule to a particular item, otherwise we would be setting the price of all the items.

Both the second and the third condition have and 'alignment' condition. This is the part of the condition that makes sure that all the pricing fields in the rule belong to the same CDR. This is needed, because there could be many records being evaluated in memory at the same time.

Dealing with large number of pricing rules

The pricing example we just saw get the job done: it set the price for calls between the 604 area code and the 512 area code. Still this was done with a rule that is 10 or so lines of rules code. And we might potentially need one rule per area code combination. We can end up with hundreds or thousands of rules that are almost the same except for a the area code numbers and the price.

There is a solution for this situation: Decision Tables. This is a feature of Drools, it allows you to use an spreadsheet done with Open Office or Excel (or a plain text CVS file) and enter the data for each rule as a row. In Drools decision tables are a way to generate rules driven from the data entered into a spreadsheet.

They are a tool to generate the 'technical' text rule that we saw earlier in this chapter automatically from the data of a spreadsheet. You will define a template with the rule text in common to all the rules, then you only need to enter the source area code, target area code and price for each rule. Each of these values go in their own spreadsheet cell.

There is no point in documenting here how to use decision tables, they are not part of **jBilling**, but part of Drools. There is a chapter dedicated to them in the Drools documentation.

The following example will help you get started faster:

RuleSet	Pricing		
Sequential	TRUE		
Import	com.sapienter.jbilling.server.item.PricingField, com.sapienter.jbilling.server.item.tasks.PricingResult, org.apache.log4j.Logger		
Variables	org.apache.log4j.Logger LOG		
RuleTable Rates			
CONDITION	CONDITION	ACTION	ACTIVATION-GROUP
$result : PricingResult	PricingField		
itemId == $param, price == null, $resultId : pricingFieldsResultId	name == "dst", strValue matches "^$param.*", resultId == $resultId	modify($result) { setPrice("$param"); } LOG.debug("Setting price of item 1 to " + $result.getPrice());	
Call Type (item Id)	**Digits**	**Price**	**Group**
2800	5994165	0.280	1
2800	5215591	0.180	1
2800	5215585	0.180	1

Chapter 8
Mediation errors

The mediation results

When the mediation process is done, the results of what happened with each of the CDRs that were processed are available. There are many outcomes: the record produced some charges to a customer, or maybe it did not, or maybe some errors prevented all the steps from happening.

For those records that present errors, jBilling will save them so you can setup an automated way to get the reprocessed.

But first, it is important to understand the results that jBilling identifies. The following sections list each of the possible results

Done and billable

This is the 'typical' result: a record was successfully processed (all the steps) and it translated into a charge to a customer. For this to happen, the 'done' field of the mediation result object needs to be set with 'true' and the 'lines' field should have some lines in it.

Done and not billable

The record was successfully processed (done is set to true), but there are no lines present. The typical example is when a record represents an event that the customer will not pay for. Like a phone call that was not answered:

```
rule 'cancel not answered call'
when
    $result : MediationResult(step == MediationResult.STEP_1_START,
      done == false )
    $field : PricingField( name == "disposition", resultId ==
      $result.id, value != "ANSWERED" )
then
    $result.setDone(true);
    retract($result);
end
```

You should always mark records as done, even when they are not billable.

Error detected

If a result is left with done equal to false and the mediation process is done, then **jBilling** will consider this an error. You should always have rules that contemplate every outcome of a record and set the 'done' field as true.

The mediation process will try to identify the reason why this record failed, assigning a code (or many) to it. This code then will be saved along with the whole record for later reprocessing (see the next section).

The following table shows the situations that jBilling will check, and the codes for each of them:

MediationRecord field	Error	Code
lines	No items are mapped to this record	JB-NO_LINE
diffLines	The delta between the existing order and the new charges was not resolved	JB-NO_DIFF
currentOrder	The user's current order was not fetched	JB-NO_ORDER
userId	The user was not assigned for this record.	JB-NO_USER
currencyId	The currency of this record was not resolved.	JB-NO_CURRENCY
eventDate	The date of this record was not resolved.	JB-NO_DATE

For example, if a record is coming for a customer that has not been created in jBilling, then the rules that try to assign the user to the record will fail and the 'userId' will be left blank. All the rest of the steps will not take place because the rule that transitions from step 1 to 2 will never take place: the user has not be resolved.

You will find the record in the recycle bin, with probably all the error codes in it except the date. When you see this, you know that the real culprit is the user, because it is the field associated with the earlier step.

Error declared

It is a good practice to try to cover all the possible situations that a record can have with rules. If there is an error that you can detect, then you can let jBilling know using the field '`errors`' from the `MediationResult.` This is a list of strings, so you can add any number of errors, they will be preserved when the errors are saved for reprocessing.

Here is an example that catches negative durations:

```
rule 'check call duration'
when
    $result : MediationResult(step == MediationResult.STEP_1_START, 
done == false )
    $field : PricingField( name == "duration", resultId == 
$result.id, value < 0 )
then
    $result.setDone(true);
    $result.addError("ERR-DURATION");
    retract($result);
end
```

There isn't any limit to the format or number of errors that you can add. They just need to make sense to you. As you saw in the previous section, the errors that **jBilling** resolves use codes that start with 'JB', so your errors should start with something else.

Reprocessing records with errors

jBilling will consider all the records that fall into the error categories (detected or declared) as *not processed*. This is important, because otherwise the filter that jBilling does to avoid processing the same record twice would prevent a record with an error to be processed later.

The normal work-flow would be: the mediation process runs, some records have errors, you review these records and correct what caused the problems, then run the mediation process again only for the records that had errors.

Saving the records with errors

There is a plug-in category that is in charge of saving records with errors. The id of this category is 21. There are two implementations of mediation error handlers:

SaveToFileMediationErrorHandler:

This plug-in will save to a file every record with an error. The format is plain text with each field separated by commas (csv). These are the parameters that modify its behavior:

- directory: the directory where the files will be saved. It defaults to the value of the property 'base_dir' from the jbilling.properties file with the added 'mediation/errors'. Example: if 'base_dir' is “/home/jbilling/” then the error files will bin in “/home/jbilling/mediation/errors”.
- fie_name: The name of the file name to use. It defaults to “mediation-errors”. The system will always add a “csv” extension to the files saved.
- rotate_file_daily: if 'true', then the file name will have the date added to its name in the fomat “_yyyyMMdd”. Example “mediation-errors_20100301.csv”.

SaveToJDBCMediationErrorHandler:

This plug-in will save a record in a database for every record with an error. It will use the name of each pricing field as the column name that the table getting the errors has to have.

These are the parameters that modify its behavior:

- url: This is a mandatory parameter. It is the url for JDBC connection to database, i.e. jdbc:postgresql://localhost:5432/jbilling_test
- driver: The JDBC driver class for connecting to DB, defaults to 'org.postgresql.Driver'
- username: The user name for the database, defaults to 'SA'
- password: The password for the database, defaults to a blank string ("")
- table_name: The name of table name for saving the error records, defaults to 'mediation_errors'
- errors_column: The column name for saving error codes, defaults to 'error_message'

- retry_column: Column name for saving flag of reprocessing, defaults to 'should_retry'.

Fixing the errors

Now that you have the records that produced errors, and that you know why (or at least, a starting point thanks to the error codes), you will need to fix the problem.

The problems can be grouped in two categories: rules and data. The first one happens when some of your rules are wrong, incomplete or just missing. For example a rate card with pricing rules might be missing some destinations, which causes the price to be zero and an error detection rule to fire.

Data rules are those that involve some missing or wrong data. If a customer is missing because she was given a phone line before she has a record in the billing system, then the billing system will complain.

You will need to somehow separate the records that you want to reprocess. If you are using a database, then this can be as simple as setting a boolean column to true.

Reprocessing the records

When it makes sense to reprocess the records because the errors have been fixed, you can just simply run the mediation process again.

If you are using files, you can simply copy the file saved by the error handler plug-in. If your original file was a csv file, then you could get away without have an extra mediation configuration for the reprocessing of mediation records.

In practice, it is best to just have one mediation configuration, with its format and other data, to deal with the reprocessing. This is called the recycle bin. It is more practical because you can get to a quite automated configuration. Let's see an example:

1. The mediation process runs, for all its configurations. The normal one find some errors, which are saved in a database. The recycle bin does not find anything, because the column that flags an error record to be reprocessed defaults to false.

2. An operator reviews the records in the recycle bin, and proceeds to fix some of them. She sets the retry column to 'true'.

3. The mediation process runs again. As in the first step, it will process normal records and some of them might go to the recycling table. When the recycling configuration runs, this time it will find some processable records that are actually fixed records from the first run.

A new configuration to handle the fixed errors is practical because it helps you deal with the logistics of these records: where they are, when to process them, etc. This does not mean that you need new mediation *rules* to deal with these records. The new configuration affects only from where, or how the records get to be read, but not how they are processed.

The mediation module is ignorant of how records are selected from processing, all the logic applies to any record regardless of its origin.

Chapter 9

Real-time Mediation

The difference of real time processing

The typical mediation scenario involves large number of records generated by some external system and then processed in a scheduled, periodical run. This batch processing has many limitations, the most important is that there is some time (usually a few hours or a full day) in between the events happening and the billing system 'being told' of these events.

Batch processing is still very common in the telecommunications industry. It has been in place for a long time and people are comfortable with it. Still, it is a product of the technical limitations of computer systems that today are decades old.

If you need to validate an event *before* letting in happen, or you need to keep a balance updated up to the second, then batch processing will not do it. The classic business case for this is pre-paid services where a customer has a balance and can only use a service as long as the balance remains positive. Every attempt to use the service has to check if there is enough balance.

jBilling can process events both in batch and real-time. Real-time processing does not involve reader plug-ins to go to files or databases for the records. Instead, the API is used to send the events as they happen.

Processing an event in real time

To send an event in real time, your system needs to be able to make a SOAP call to jBilling. There are specific API methods used for dealing with real-time mediation. These are documented in the integration guide. This is a list of the most relevant ones:

rateOrder	Will evaluate an order with pricing rules and item management rules *without* creating the order. Your application can show the order to the customer to approved it, and the create it.
rateOrders	Same as the previous one, but it does many orders in a single call.
getCurrentOrder	Returns the order that the mediation process is using to collect usage for a billing period.
updateCurrentOrder	**Runs the mediation process for a single event, passed as an argument**
validatePurchase	Runs the mediation process for a single event, but it does not update the customer account. Instead, it checks if the customer has enough balance to purchase the charges that the event represents.
validateMultiPurchase	Same as previous, but taking several

	events as input.

The most important method is `updateCurrentOrder`. A call to this method would amount to the mediation process running for one single event. The method will take an array of pricing fields, this is the equivalent of a mediation reader reading one event and returning that to the mediation process.

This method can instead take an array of order lines. If this is the case, then the mediation process does not run. See the integration guide for more details.

Here there is an example of an API call that would mediate in real-time one event:

```
PricingField pf = new PricingField("call-type", “local”);
PricingField duration = new PricingField("duration", 5); // 5 min
PricingField dst = new PricingField("dst", "12345678");
OrderWs order = api.updateCurrentOrder(userId, null,
      new PricingField[] { pf, duration, dst }, new Date(),
      "Call to " + dst.getStrValue());
```

This code starts by creating three pricing fields. This is the data that the mediation process will use to resolve the item, quantity and potentially price.

This data is passed to jBilling calling `updateCurrentOrder`. The jBilling user has to be known to the caller already. The description of the event is also passed.

Note the differences with batch processing. In batch processing, the user is resolved with rules in in step 1 and the description is also done in a rule.

This example assumes that there are rules the will make sense out of the fields 'call-type', 'duration' and 'dst'. These rules don't need to be specific to real-time mediation. The same set of rules could be doing batch and real-time mediation.

After this call, the one-time order belonging to the user affected will have an extra line, or an increased quantity. This updated order is returned to the calling code.

Validating an event before it happens

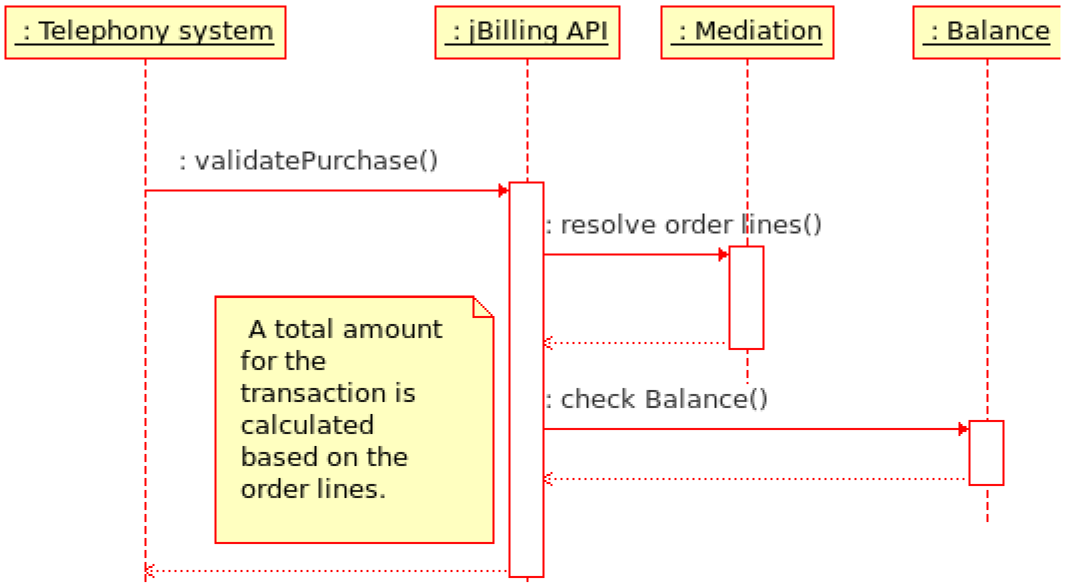
So far, the scenarios for mediation have been that an event has taken place. This is, it is in the past. What if you want to check if an event *should* be allowed to happen?

This is the typical 'pre-paid' scenario: a customer buys a phone and 'loads it' with some initial amount of money. This customer is allowed to use the phone as long as the balance on the account is greater than zero. Before usage (before a call is placed, an sms message sent, etc), the telephony system should call the billing system to validate that this service is available to the user.

A very similar case is the one where the user is not pre-paying, but there is a credit limit. Here the customer is given a credit limit when the customer is first singed-up. The customer can use the services up to that limit. When the limit is reached, the services are withheld.

jBilling will keep track of these balances (pre-paid or credit limit) and help with the validation through the '`validatePurchase`' API method. Please refer to the User's guide (balance type section) and integration guide for more information.

The following is the high level sequence of events:



A call to `validatePurchase` will (usually) result in the mediation process called. This is to resolve the fields that come as raw data to an actual product with a price. The total of this charge will then be compared against the customer's balance (subtracted if it is pre-paid, or added if it is on credit).

The result of this comparison is the passed to the caller. The results include not just a 'true' or 'false' but a quantity of how much of that product the customer could currently buy. This is helpful to enforce limits on services: if a customer has a balance to make a call of 50 seconds, then the call should be terminated after that time.

After this call is done, the balance (and current order) of the customer is unaffected. You need to call `updateCurrentOrder` for the balance to be modified.

This is an example call to `validatePurchase`:

```
PricingField[] pf2 = {
      new PricingField("src", "604"),
      new PricingField("dst", "512"),
      new PricingField("duration", 1),
      new PricingField("userfield", myUser.getUserName()),
      new PricingField("start", new Date()) };
result = api.validatePurchase(myId, null, pf2);
```

The code starts setting a few pricing fields. Then calling the validation method. The second parameter (the item) is left null: we expect the product to be resolved using the mediation rules.

Chapter 10
Example scenario

Our example plan

After going through every step and component of the mediation process, it is time to put everything together to make a phone plan work with **jBilling**. The idea is not so much to get into complex scenarios, but to see all the pieces working together.

Ours is a phone plan offered by a company that uses Asterisk as a telephone switch. The marketing department decided to call this plan “Super 500”, it includes the following:

- 500 anytime minutes
- Free calls for the weekends
- 30 cents per minute, when calling on weekdays and going above the included 500 minutes.
- Long distance calls are allowed but priced separately. The price per minute is dependent on where the call originates and where and the target phone number.
- The monthly fee is 100 dollars.

We will start from a freshly installed **jBilling**. This means that we will not cover installation and deployment. Instead, we will focus on the items, configuration and rules that make this plan come to life.

Initial configuration

We will be using the example company *Trend* for this exercise. By default, this company is already created in **jBilling** as the company number 1. Remember, you can login using the user name 'admin' with password '123qwe'.

Trend's configuration is not ready for the kind of business rules we need to implement in our plan, we will have to add some plug-ins to allow rule-based processing.

Plug-ins

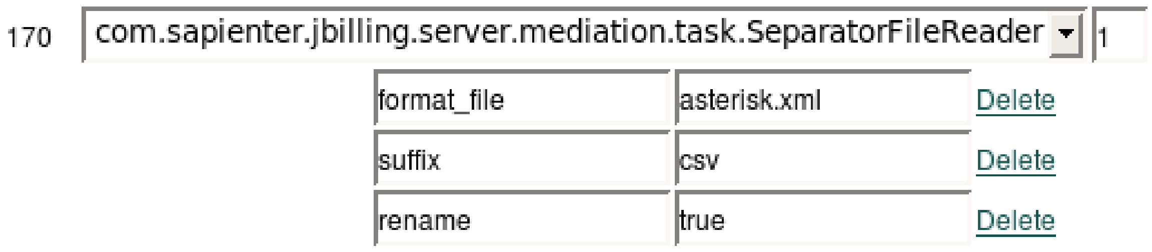
Items management: The plug-in configured for *Trend* for item management is the `RulesItemManager2`. This rules-based plug-in is what we need, so there are no changes to do here.

Pricing: Here *Trend* has also the right plug-in `RulesPricingTask2`. This will allow us to set-up rating rules in our mediation process.

Mediation Process: The `RulesMediationTask` is already present in the configuration, but you need to change the parameters it uses. This plug-in needs to access to not just the mediation rules, but also the item management and pricing rules. In the bundle that you downloaded, the item management package also includes the pricing rules, so you need the mediation process plug-in to have two parameters:

- Mediation rules: name=url, value=http://localhost:8080/drools-guvnor/org.drools.guvnor.Guvnor/package/Mediation/LATEST
- Item management and pricing rules: name=url, value=http://localhost:8080/drools-guvnor/org.drools.guvnor.Guvnor/package/trend_item_management/LATEST

Readers: We are going to need one reader plug-in. *Trend* has none, so we need to add one that is capable of reading the event files created by Asterisk. The type of file is separated by commas, so we will use the `SeparatorFileReader`. We need to specify the format file, **jBilling** comes out-of-the-box with the asterisk file format in the file `asterisk.xml`. Pretty much every default value for the plug-in is fine for this example, we'll just specify to rename the files once they are processed and the suffix of the files to process, since we know they all end with '.csv':



170	com.sapienter.jbilling.server.mediation.task.SeparatorFileReader	1	
	format_file	asterisk.xml	Delete
	suffix	csv	Delete
	rename	true	Delete

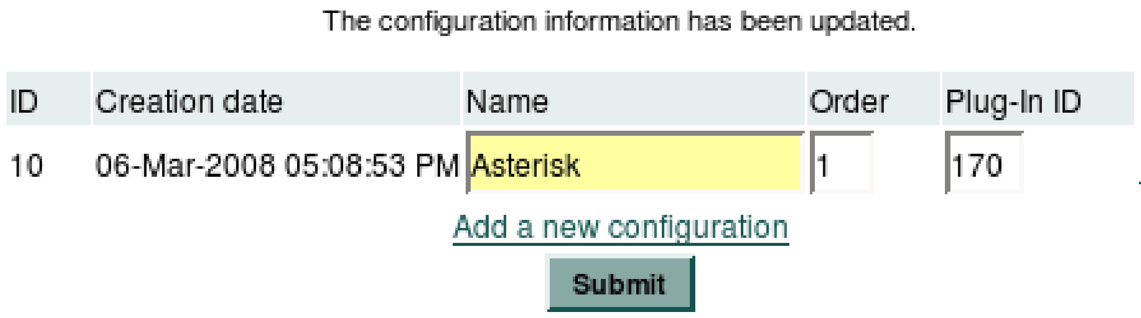
Illustration 12: The reader plug-in with its parameters

It is a good idea at this point to take a look to the format file. Open `jbilling/resources/mediation/asterisk.xml` with a text editor. It is fairly easy to create one of these format files out of the switch's documentation. When writing the plan's business rules, we will be often referring to a particular record field by name. You will need to be familiar with the format file and the CDR documentation from asterisk to understand the rules we'll be writing in later sections.

Mediation configuration

Since we now have a reader plug-in ready to run, we can now create a new mediation configuration. Click on 'System' then 'Mediation' and then on the link 'Add new configuration'.

For name, we will use 'Asterisk', and for the plug-in ID we enter 170, since that is the ID of our reader plug-in (see the previous illustration):



The configuration information has been updated.

ID	Creation date	Name	Order	Plug-In ID
10	06-Mar-2008 05:08:53 PM	Asterisk	1	170

Illustration 13: The example mediation configuration

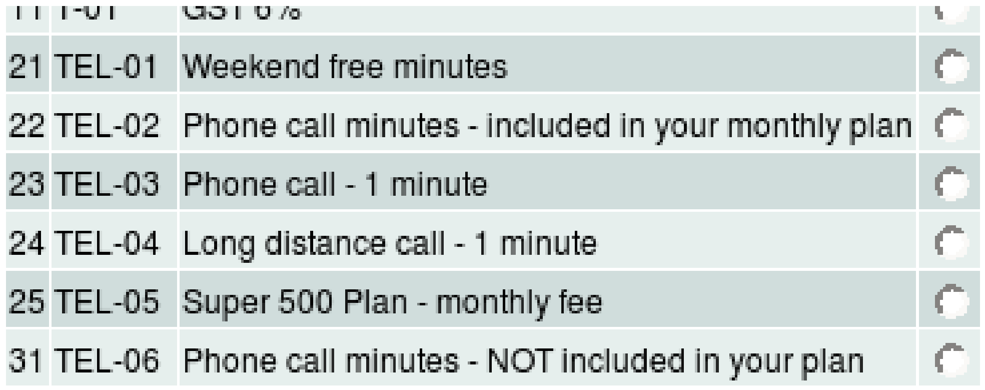
Items and customers

The example company *Trend* does not have any of the telecom items we need for this exercise. They are:

- Weekend phone call minute, priced at 0 dollars.
- Phone call minute included in the plan, priced at 0 dollars.

- Phone call minute not included in plan (above 500 minutes), priced at 0.30
- Standard phone call minute, priced at 0.30 dollars
- Long distance minute, the price is no really important, it will be set with rules.
- The 'Super 500' plan, priced at 100 dollars.

Here are the items, the ID numbers of each of them are important, we will need them when writing rules:



11	T-01	GST 6%	○
21	TEL-01	Weekend free minutes	○
22	TEL-02	Phone call minutes - included in your monthly plan	○
23	TEL-03	Phone call - 1 minute	○
24	TEL-04	Long distance call - 1 minute	○
25	TEL-05	Super 500 Plan - monthly fee	○
31	TEL-06	Phone call minutes - NOT included in your plan	○

For customers, we will just use those that are already loaded for *Trend*. There are three, with numbers 2, 12 and 22.

Expressing the plan with rules

Our phone service plan is pretty simple, and yet when we think how to implement it we discover that it involved many business rules. These business rules we can express very well using the business rules management system (BRMS).

To clarify what we want to achieve, we put together a flow diagram with the main logic points that make our plan:

You can see that the 'type' of rule is color coded: we will need some of the rules to be for the mediation plug-in, most for the item management plug-in and even some pricing rules to handle long distance rating.

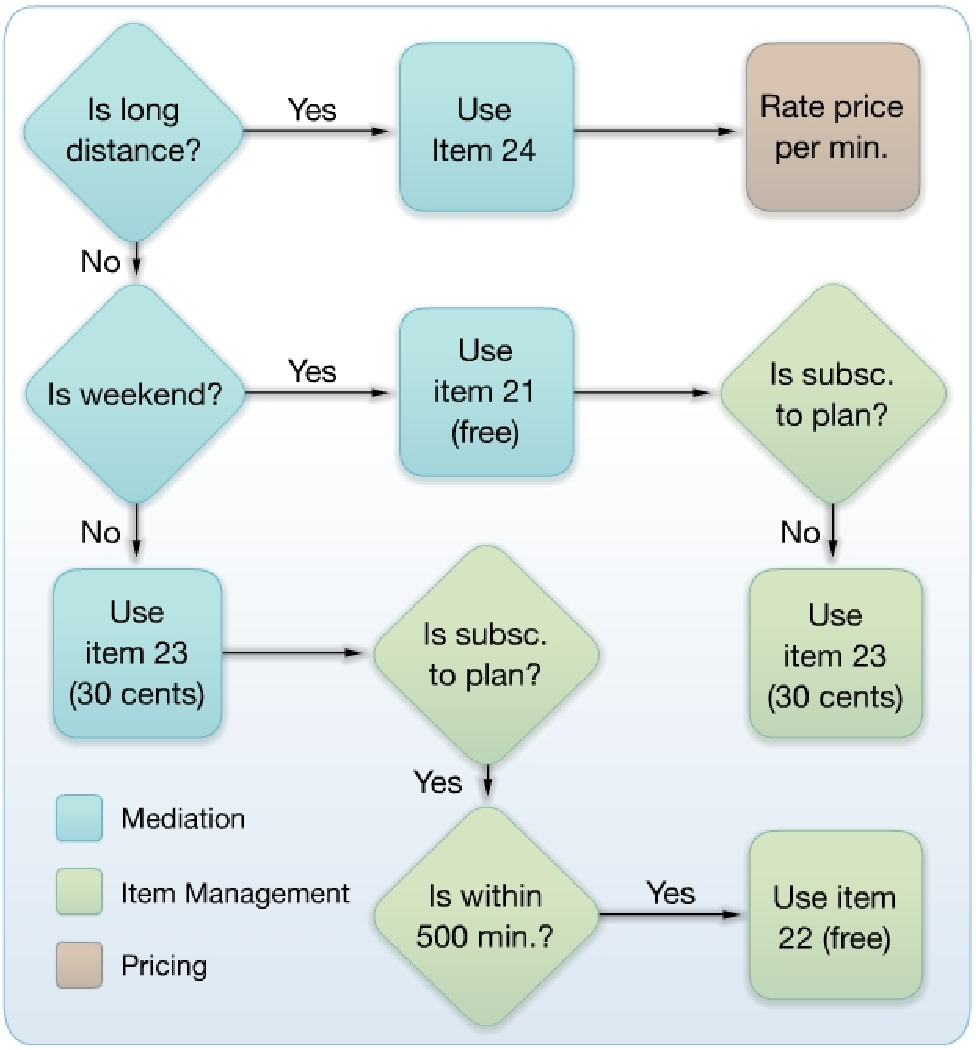


Illustration 14: The logic for the example plan

This kind of diagram helps to understand the extent of the requirements, but it has a drawback. A rule system does not follow a typical logic flow like the one in the diagram, where there is only one conditions or consequence done at any give time, and everything happens in a strict sequence.

Rules work best when it does not matter which rule is executed first or which one last. Imagine that they all get executed *at the same time*. Then, it is easier to maintain and add new rules.

Mediation rules

The mediation plug-in is where everything starts. We are going to be setting a few rules to extract data from the event records and 'translate' their content to **jBilling**. These rules will be useful as an example, but there are some omissions: for example, all the line creation rules should probably check that the phone call was actually answered. The idea is that if you have a good understanding of these examples, you will be able to build on them to setup real production rules on your own.

Defining the user

In this example will assume that the user name comes directly in the record as the field 'userfield':

```
<field>
  <name>userfield</name>
  <type>string</type>
</field>
```

Thus, the mediation rule to define the user looks like this (see the Step 1 section for more information on setting the user, and the content on the help function used here):

```
when
    $result : MediationResult(step == MediationResult.STEP_1_START, 
userId == null)
    $field : PricingField( name == "userfield", resultId == 
$result.id)
    $company : CompanyDTO( ) # needed to determine a user by its 
user name
then
    modify( $result ) {
        setUserId( getUserIdFromUsername($field.getStrValue(), 
$company.getId()) );
    }
```

Defining the event date

Each record as the start date and time, this is the time stamp the phone call started:

```
<field>
  <name>start</name>
  <type>date</type>
</field>
```

The mediation rule that takes this field and sets the date is:

```
when
    $result : MediationResult(step == MediationResult.STEP_1_START, 
eventDate == null)
    $field : PricingField( name == "start", resultId == $result.id)
then
    modify( $result ) {
        setEventDate( $field.getDateValue() );
    }
```

Long distance calls

At the mediation process, we only need to identify if the record belongs to a long distance call or not. If it is, it should add the order line using item 24. That is all, we don't need to worry about the price of the call at this point, but we do need to request the price so the pricing rules do their job.

```
salience 10
when
    $result : MediationResult(step ==
            MediationResult.STEP_4_RESOLVE_ITEM)
    $field : PricingField( name == "duration", resultId ==
$result.id)
    PricingField( srcAreaCode : strValue, name == "src",
           resultId == $result.id)
    PricingField( dstAreaCode : strValue, name == "dst",
           resultId == $result.id)
    eval( !srcAreaCode.substring(0,3).equals(
           dstAreaCode.substring(0,3)) )
then
    $result.getLines().add(newLine(24, new
            BigDecimal($field.getStrValue())));
    $result.setStep(MediationResult.STEP_5_PRICING);
    update( $result );
    PricingResult request = priceRequest(24, $result); // because
it will be converted to this...
    insert( request );
```

The code in this rule takes the values of the 'src' and 'dst' fields and compares the first three digits. If the are *not* equal (notice the '!' character to negate), then the line with item 24 is added, with the quantity coming from the value of the 'duration' field.

We are explicitly giving this rule a high priority (salience equals to 10), so it runs the first among the line creation rules. Then, we are setting the step to 5 (pricing). All this is needed, otherwise the other order creation lines might also create an order line: for example, if there is a long distance call on a weekend.

This long distance rule is *exclusive* of all other line creation rules: if the call is long distance, then use the long distance item and *do not execute any other product resolution rules.* This is easily achieved by the direct setting of the step of the result object to 5.

Weekend calls

If the call happens on a weekend (and it is not long distance), we need to use item 21:

```
salience 5
when
    $result : MediationResult(step ==
                 MediationResult.STEP_4_RESOLVE_ITEM)
         PricingField( name == "start",
               eval( calendarValue.get(Calendar.DAY_OF_WEEK) ==
                                       Calendar.SATURDAY )  ||
               eval( calendarValue.get(Calendar.DAY_OF_WEEK) ==
                                       Calendar.SUNDAY ),
                resultId == $result.id)
         $field : PricingField( name == "duration",
                resultId == $result.id)
     then
         $result.getLines().add(newLine(21, new
                 BigDecimal($field.getStrValue())));
        $result.setStep(MediationResult.STEP_5_PRICING);
        insert( new SubscriptionResult($result.getUserId(), 25) );
         update( $result );
```

We do not have to worry about this call being long distance: that was taken care of by the previous rule when it set the step to 5.

In this rule, we need to check if the call happened on a weekend. For that, the convenience method 'getCalendarValue' of the class `PricingField` comes very handy. Checking for the day of the week is a simple call to the Java standard class `Calendar`.

Just like in the previous rules, we assign a priority to this rule. It is lower than the long distance rule (we want all long distances calls to be treated as such regardless of anything else), but still higher than a normal rule. By doing this, and explicitly setting the step of the result to 5, we avoid having to write a complex rule for normal calls.

Note that there isn't a pricing request in this case: weekend calls have much easier pricing, they don't need a rate card.

Wait a minute, isn't this giving *everyone* free weekend calls? That is not what we want, free weekend calls are only for those that have subscribed to our 'Super 500' plan. Since we know that the status of a subscription is needed, we are inserting now in the memory context an object that will carry this information. Then, item management rules can act on this and swap items as needed.

Therefore, we are going to take care of not giving this free call in the next package of rules that do item management. See the diagram in the last illustration.

Standard calls

If a call is not done on a weekend and it is not long distance, it is just a plain normal phone call that should use item 23:

```
     when
         $result : MediationResult(step ==
                MediationResult.STEP_4_RESOLVE_ITEM)
         $quantity : PricingField( name == "duration",
               resultId == $result.id)
```

```
    then
        $result.getLines().add(newLine(23,
               new BigDecimal($quantity.getStrValue())));
        $result.setStep(MediationResult.STEP_5_PRICING);
        update( $result );
```

We don't need to check in this rule that the call is on weekdays or that is a local call. If the 'duration' field is still in the working memory, it means that the call is standard.

The rule is very simple then, just use item 23. This, of course, is not taking care of making the call free for customers that are subscribed to our plan and within the 500 free minutes. That will be done with item management rules.

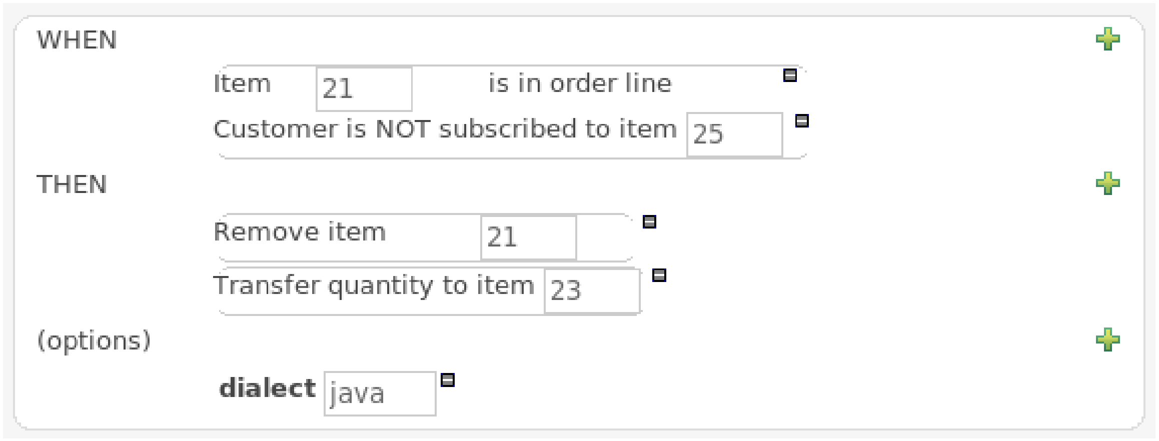
Item management rules

Unlike mediation rules, item management rules are commonly handled by business users. These rules are abstract to the events themselves, you can think of them in business terms.

Because of this, we will create these rules using the BRMS graphic interface, as opposed as typing the rule code creating 'technical' rules. We will take advantage of jBilling's domain specific language for item management rules.

Free weekends for subscribers only

You might recall that the mediation plug-in would use the free weekend phone call item for everyone. Now it is time to limit this to only those that have subscribed to our plan:



WHEN

Item 21 is in order line

Customer is NOT subscribed to item 25

THEN

Remove item 21

Transfer quantity to item 23

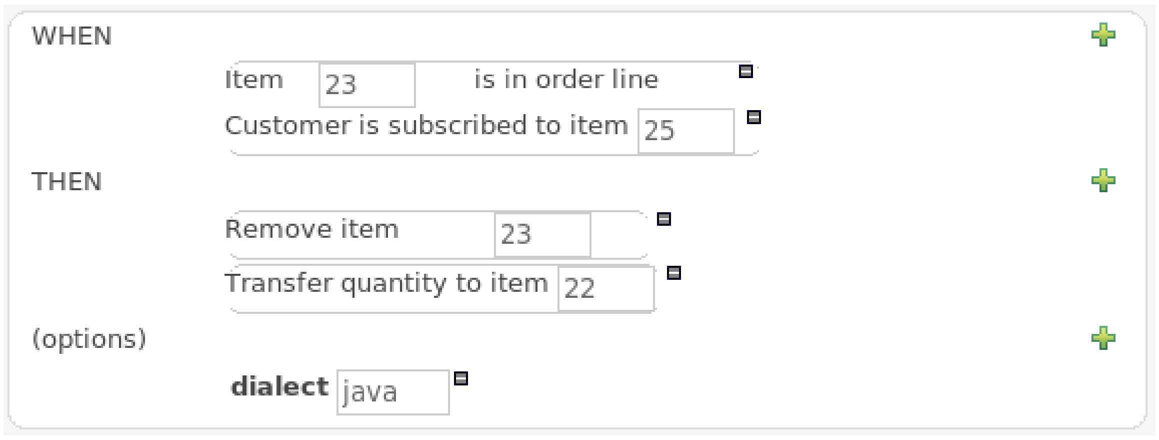
(options)

dialect java

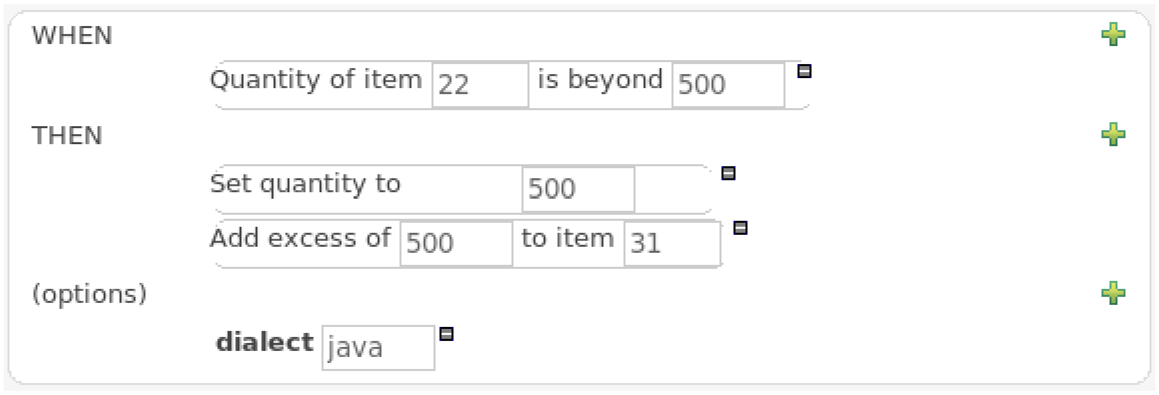
Thanks to the use of the DSL, the rule is self-explanatory. Only those subscribed to our plan can have free weekend calls; the rest get the standard item with a price of 30 cents.

Minutes included in the plan

Our subscribers are entitled to 500 minutes of free calls. For that, we will first blindly convert any item 23 (standard call) to item 22 (free call) if the customer is a subscriber.


WHEN
Item 23 is in order line
Customer is subscribed to item 25
THEN
Remove item 23
Transfer quantity to item 22
(options)
dialect java

Then, with another rule, we will make sure that the amount of those free minutes does not exceed 500. If so, it will transfer the excess minutes to item 31 (excess minute), which is priced at 30 cents:


WHEN
Quantity of item 22 is beyond 500
THEN
Set quantity to 500
Add excess of 500 to item 31
(options)
dialect java

It would be possible to avoid having item 31 altogether and add the excess of minutes to item 23, which is the standard phone call minute. This would make the first rule more complicated, because we would not be able to transfer *all* items 23 to item 22 (the rules would contradict each other). The first rule would need to check if item 22 is present and if it so, only transfer when the quantity is not grater than 500.

The decision was to keep that rule simpler by adding another item (31).

Pricing long distance calls

It is time to give a price to those long distance calls. We switch to a different package now, the pricing package of rules. In this example, we will only set a price for a call between two specific area codes:

```
when
    $src : PricingField( name == "src", strValue matches "^604.*")
    $dst : PricingField( name == "dst", strValue matches "^512.*",
           resultId == $src.resultId)
    $result : PricingResult( itemId == 24, price == null,
          pricingFieldsResultId == $src.resultId )
```

```
     then
        modify( $result ) {
            setPrice("0.50");
        }
```

Here we simply check if the item in question is 24, since this is our long distance minute item. If so, then we check if the call is from 604 to 512. If so, the price is 50 cents.

We will probably have a large number of these kind of rules, based on more then just two fields. This will likely lead to a decision table where you can maintain these rules in a spreadsheet.

Processing some events

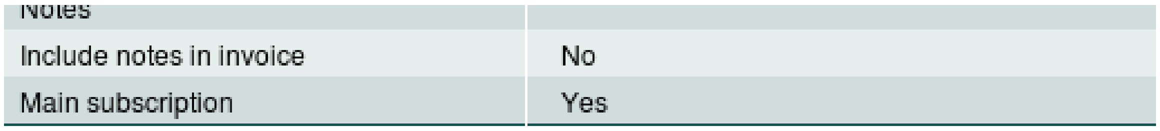
We need to put all the rules to work by running the mediation process and get a few events processed.

For that, we need first to create the main subscription orders for two customers. To do that, we have to add the preference that allows a company to use main orders:

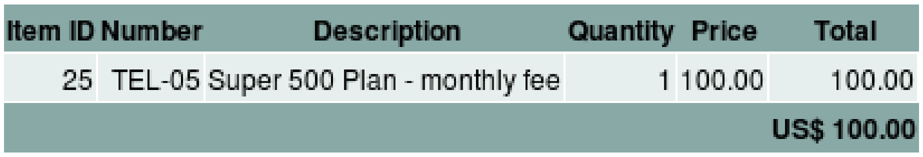
```
INSERT INTO PREFERENCE (ID,  TYPE_ID, TABLE_ID, FOREIGN_ID, INT_VALUE )
    VALUES   ( 17,  41, 5, 1, 1)
```

With that, we are adding preference 41 to *Trend* (company 1).

Now, create an order for customer 'jsmith' with the plan item (25). Make sure the 'main subscription' flag is set:



Notes	
Include notes in invoice	No
Main subscription	Yes



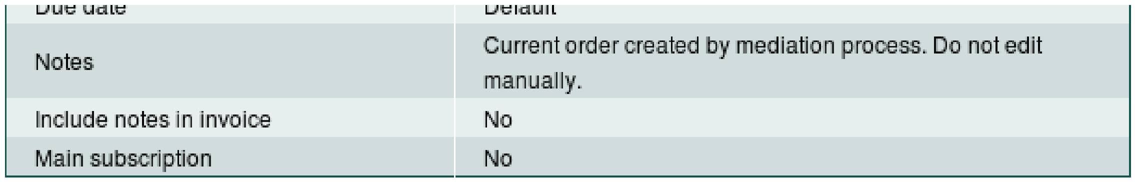
Item ID	Number	Description	Quantity	Price	Total
25	TEL-05	Super 500 Plan - monthly fee	1	100.00	100.00
					US$ 100.00

The records for Asterisk CDRs are too long to copy here in any meaningful way. Let's see the most important fields:

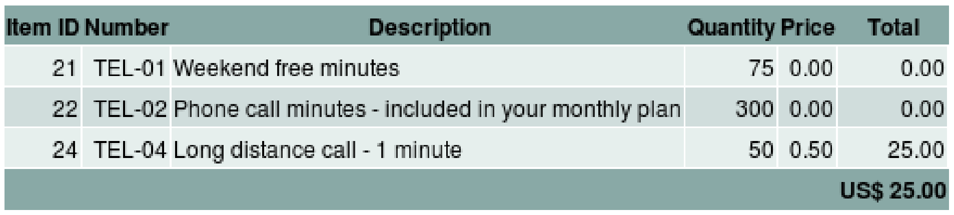
```
01,6041231234,6041231234,...,20071101-114011,20071101-114016,20071101-
114511,300,295,ANSWERED,3,jsmith

02,6041231234,5121231234,...,20071101-114011,20071101-114016,20071101-
114511,50,295,ANSWERED,3,jsmith

03,6041231234,6041231234,...,20071103-114011,20071103-114016,20071103-
114511,75,295,ANSWERED,3,jsmith

04,6041231234,6041231234,...,20071103-114011,20071103-114016,20071103-
114511,75,295,ANSWERED,3,twilson
```

We have a long distance call in the second record, two weekend calls in the last two records; the first record should get into the included minutes and of course jsmith is subscribed to the plan, but twilson is not, so that last record should not be a free phone call.

You can not run the mediation process manually, you would need to schedule it as explained in earlier chapters. After the mediation process takes place, we will have one new order for jsmith. This is a one-time order with the charges coming from the above records:



Due date	Default
Notes	Current order created by mediation process. Do not edit manually.
Include notes in invoice	No
Main subscription	No



Item ID	Number	Description	Quantity	Price	Total
21	TEL-01	Weekend free minutes	75	0.00	0.00
22	TEL-02	Phone call minutes - included in your monthly plan	300	0.00	0.00
24	TEL-04	Long distance call - 1 minute	50	0.50	25.00
					US$ 25.00

Chapter 11
Provisioning

Introduction

Provisioning can be defined, from **jBilling**'s point of view, as the process of supplying events to other systems, such as a service gateway or a telephony netwrok system, in order to have those events serviced (or "provisioned") as required.

For example, consider the service activation process: a customer acquires a service, which implies the creation of an order in **jBilling**. The system that provides the service is however separated from **jBilling** itself, and therefore activation and deactivation of the service itself is carried out separately.

This is not the best scenario, as usually you'll need more responsiveness from the service infrastructure when certain billing events occur: if the customer opens an account and pays it, activate the service, or do the exact opposite when an invoice is not paid timely. This is where provisioning comes into the picture.

The provisioning module provides **jBilling** with the ability to communicate with such systems, in a way that is both scalable and flexible enough to satisfy most integration scenarios.

Overview of the provisioning process in jBilling

Illustration 15 shows how provisioning is integrated into **jBilling** and demarcates its main components.

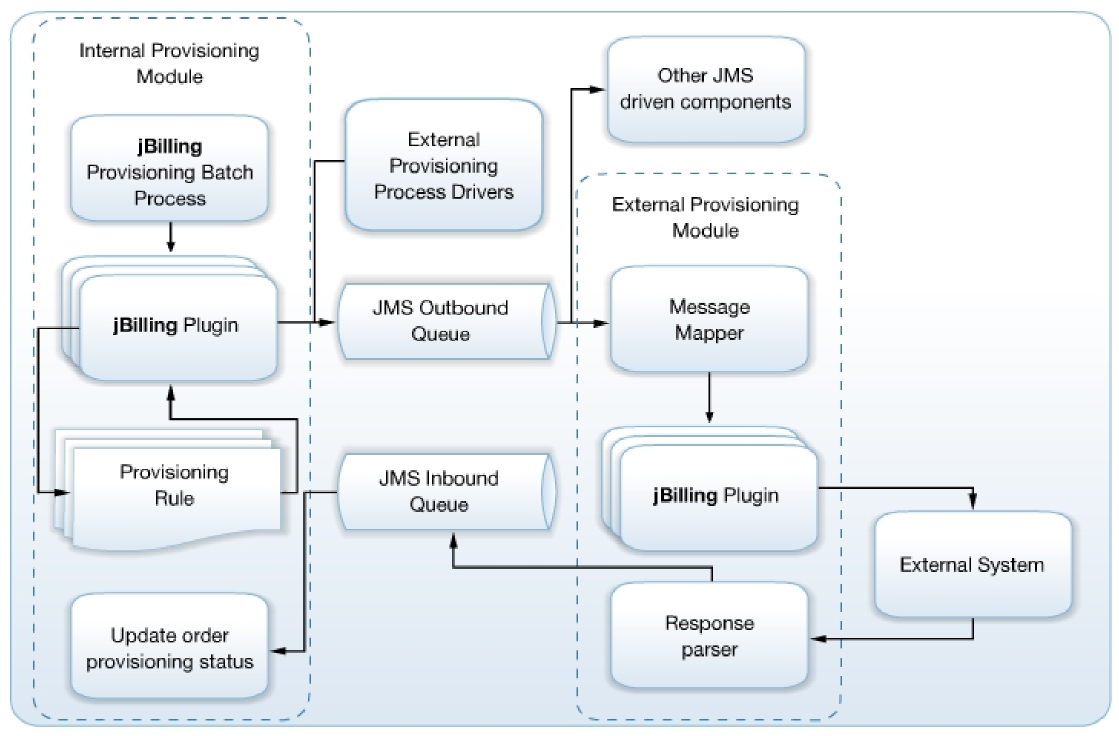

Internal Provisioning Module
jBilling Provisioning Batch Process
jBilling Plugin
Provisioning Rule
Update order provisioning status
External Provisioning Process Drivers
JMS Outbound Queue
JMS Inbound Queue
Other JMS driven components
External Provisioning Module
Message Mapper
jBilling Plugin
Response parser
External System

Illustration 15: High level design of the provisioning module

The provisioning process is initiated from a batch task that can be scheduled from within **jBilling**. This task identifies all orders that require provisioning and determines if specific events (such as a newly created subscription) have occurred. If so, it fires the appropriate event, which is captured by the provisioning plug-in task processor.

The plug-in processor receives these events and passes its data to the business rules component. Business rules take care of determining the type of event that is being processed and what commands and parameters must be passed to the external system.

The use of business rules provides great flexibility in integration, as all commands and parameters that must be passed to the external system can be abstracted in the business rules and can therefore be easily modified or adapted to any system.

Once the exchange pattern has been populated by the rules engine, the plug-in proceeds to prepare a message containing all commands and parameters just generated, and submits this message to an outbound JMS queue.

At the receiving end of this queue lies the external provisioning component. This component takes care of receiving the message and determining how the commands indicated in the message must be passed to the external system. It then submits the commands to the external system and processes its responses.

All responses from the external system are wrapped in a JMS message and submitted back to the internal provisioning component by means of an inbound message queue. The response message is parsed by the internal component and the provisioning status of the order is updated.

The advantage of JMS

The use of JMS provides great flexibility and scalability to the process: you can deploy the external provisioning module in a separate server, and can easily control the amount of processing threads that will service the queue and respond to incoming messages. Also, you can take advantage of the persistence and fault-tolerance capabilities provided by all JMS infrastructures.

As can be noted on Figure 15 above, the JMS queues effectively decouple the internal and external provisioning modules. This not only provides infrastructural advantage (in terms of scalability), but also provides sophisticated functionality: the JMS infrastructure allows multiple message producers for each queue and (when "Topic" queues are used, as is the case here) multiple consumers.

This not only implies that you can substitute either the internal or external modules with customized message producers or consumers that generate or service provisioning requests in a completely different manner, but also that you're able to integrate other modules in the provisioning process without detaching the built-in components.

Therefore you can have customized provisioning event generators that post messages to the outbound queue and drive the external provisioning module as if those messages were coming from the internal module, allowing you to have a single point of interaction with the external system.

In the same manner, you can provide customized JMS-driven components that receive notifications from the outbound or inbound queue to provide additional functionality (for example, a custom component listening to the inbound queue to generate e-mail confirmations whenever a successful provisioning operation takes place).

The only requirement an external JMS-aware component needs to know to instantly participate in the provisioning process is to be able to "understand" the language in which provisioning messages are transmitted.

An Example

Throughout this chapter, we will refer to a hypothetical telecommunications infrastructure that provides cell phone services to paying customers to exemplify the use of the provisioning modules to feed an external system. We assume that **jBilling** is being used to provide billing services to this infrastructure, and keep track of payments, subscription cancellations and charging for the services provided.

For the sake of simplicity, we assume there is only one service type (which maps to a single item type in all orders), which is a flat rate plan. **jBilling** will handle the monthly recurring payments that users will pay to receive the service.

The external system we're trying to talk to handles activation and deactivation of the cell phone service. It has no notion of the payments and invoicing handled by jBilling, and only needs to know when to activate a new subscriber (because it has started to pay the monthly fee) or when to deactivate a former subscription (because the user has stopped paying the monthly fee).

Therefore, we need to notify the external system about subscription activation and deactivation events that occur within **jBilling** during billing operations.

While we progress through this chapter, we will return to this example and provide code and implementation details about how the provisioning process helps orchestrate the interactions between **jBilling** and the telecommunications infrastructure in this example.

The internal provisioning module

The internal provisioning module takes care of identifying the events that require provisioning, applying business rules to these events and relay the commands produced by the rules to the external provisioning module.

In order to activate the provisioning process, the `jbilling.properties` file needs to be modified to contain the following line:

```
process.run_provisioning=true
```

Setting this property to `true`, a daily batch process that runs within **jBilling** will generate provisioning events as they are detected.

The provisioning status of orders

Orders (and order lines) now have a status field that indicates their provisioning status. All orders entered will track the status of the provisioning process. This status serves to identify which orders still need to be provisioned and which have already been subject to this process.

These are the acceptable status codes:

- `PROVISIONING_STATUS_ACTIVE`
- `PROVISIONING_STATUS_INACTIVE`
- `PROVISIONING_STATUS_PENDING_ACTIVE`
- `PROVISIONING_STATUS_PENDING_INACTIVE`
- `PROVISIONING_STATUS_FAILED`
- `PROVISIONING_STATUS_UNAVAILABLE`

These values can be used from within the provisioning rules to determine the status of the provisioning process and aid in deciding what specific operations should be forwarded to provision the event.

Provisioning events and the provisioning plug-in

jBilling's pluggable architecture introduces the notion of events, which is the way in which the system notifies the occurrence of a given set of conditions to a chain of observers for the purpose of having the event serviced. The description of events and the way in which they are implemented and produced within **jBilling** is beyond the purpose of this chapter and is described in detail in Chapter 7 of the **jBilling**'s Extension Guide.

Provisioning conditions are handled as **jBilling** events. A daily batch process scans the database searching for order that require provisioning. Once it finds one, it determines whether a specific event has taken place for that order and, if necessary, raises an event that will be fed to the chain of plug-ins configured in **jBilling**. This is the starting point of the provisioning process.

The type of event that is fired determines which specific type of provisioning action that will take place:

- `SubscriptionActiveEvent`: fired when a subscription becomes active (usually it is used to trigger the activation of the related service). This event refers to the entire order, rather than specific order lines.
- `SubscriptionInactiveEvent`: fired when a subscription changes state and becomes inactive (usually this event is used to trigger deactivation of the related service). This event refers to the entire order.
- `NewQuantityEvent`: fired when the quantity of an order line is changed (usually it is used to trigger a change of service type or level). This event refers to a single order line.

It is important to note that the event serves as a signal of the occurrence of a given condition, and makes no assumptions whatsoever regarding the handling of the condition. This is delegated to the plug-in chain.

The event is pushed through the plug-in chain in search of a plug-in that will service the event. This is exactly the same way in which **jBilling** services other events that occur in the system, such as payment failures or unavailability of payment gateways (again, refer to the Extension Guide for further information regarding events, plug-ins and their implementation details).

By default, provisioning events are serviced by business rules written in the Drools language, as described in the next section. **jBilling** provides a default implementation that services provisioning events, called `ProvisioningCommandsRulesTask`. You can of course write a dedicated plug-in to service provisioning events, this is just the default implementation and will be described below.

In order to properly serve provisioning requests with business rules, you will need to activate this plug-in by configuring it in the plug-in list under the `System -> Plugins` menu of **jBilling**'s User Interface. Refer to the **jBilling** User's Guide and Extension Guide for instructions on how to activate plug-ins in the system.

The purpose of this plug-in is that of evaluating the order (or order line) for which the event is taking place against the set of provisioning rules you provide. These rules will describe what specific interactions should take place with the external system in order to properly service the event.

In the case of our sample telecommunications infrastructure, we're only interested in the `SubscriptionActiveEvent` and `SubscriptionInactiveEvent` events, which notify us about subscription activations and deactivations, which are the cases in which we need to interact with the external system to provide or suspend the cell phone service. In this case, we ignore the `NewQuantityEvent` condition, whose main purpose is that of sensing changes in rate plans or service types. Since our sample cell phone operators only provides a single flat rate plan, we can safely ignore changes in rate plans or service types.

Provisioning rules

Business rules expressed in the Drools language are a very flexible manner of capturing business-related decisions and have been widely adopted within **jBilling**. The provisioning process is not an exception, as the default internal provisioning module delegates all decisions on how to process a specific provisioning event to business rules.

Business rules are introduced elsewhere and will not be described here, please refer to the "Chapter 6 Processing with rules" section of this document, Chapter 8 of **jBilling**'s Extension Guide or the Drools User Guide for further information regarding the syntax and usage of business rules.

Let's therefore concentrate on a simple provisioning rule:

```
rule 'provisioning_activate'
when
    $orderLine : OrderLineDTO(itemId == 2, provisioningStatusId ==
        Constants.PROVISIONING_STATUS_INACTIVE)
    CommandManager(eventType == "activated")
then
    command.addCommand("activate_user", $orderLine.getId());
    command.addParameter("msisdn", "12345");
    command.addParameter("imsi", "11111");
end
```

The rule above matches all events of type "activated" (the `eventType` property has two possible values, "`activated`" or "`deactivated`"), for order lines with item ID 2 having a provisioning status of `PROVISIONING_STATUS_INACTIVE` (which means they were inactive prior to the provisioning operation being serviced).

If these conditions apply, the rule adds a list of commands and parameters to the "`command`" object (which is a global import of type `com.sapienter.jbilling.server.provisioning.task.ProvisioningCommandsRulesTask.CommandManager`), by calling the appropriate `addCommand()` and `addParameter()` methods.

These commands and parameters are the ones that will be passed to the external system via the JMS message and their quantity, type and value is heavily dependent on the type of service you're trying to involve in the provisioning process.

You can pass several commands and parameters, in the same manner:

```
rule …
when
    …
then
    command.addCommand("do_something", $orderLine.getId());
    command.addParameter("parm1", "12345");
    command.addCommand("do_something_else", $orderLine.getId());
    command.addParameter("parm2", "54321");
end
```

Each command is identified by a command name (the first argument to the `addCommand()` call, "activate_user" in this case), which will be important later, when the message commands will be translated into external calls.

The order line ID number is passed in the command definition and serves in identifying the order line for which the event was raised when the provisioning response is processed.

Of course, you can supply additional rules that capture other conditions. You can apply whatever conditions you wish on the order line (item type, quantity, amount, etc.) or use order line data as parameter data in the command calls.

The Drools working memory for provisioning business rules contains two types of objects that can be used to express rule conditions or actions in business rules:

- `command`: a global `CommandManager` object. It contains two methods for adding commands and command parameters to the provisioning message:
    - `addCommand(String commandName, Long orderLineId)`: is used to add a command to the provisioning message. The command will later translate to an interaction in the external provisioning module.
    - `addParameter(String name, String value)`: a name-value pair that describes a parameter to be passed to the external provisioning module (usually used to communicate data to the provisioned system).
- `CommandManager`: you can use this object in rule conditions as well. This object exposes the following properties:
    - `eventType`: Has two possible values: "`activated`" or "`deactivated`", indicating whether the event happened during subscription activation or deactivation, respectively (in case of `NewQuantityEvents`, it is "`activated`" when the previous quantity for the order line was 0, "`deactivated`" when the new quantity for the order line has been set to 0, and has no meaningful value when both prior and new quantities are different from 0).
    - `OrderDTO`: the object that describes order parameters. You can use it to match most order data, such as status, recurrences, periods, user, etc.
- `OrderLineDTO`: used in conditions to match the order line that triggered the event (in case of a `NewQuantityEvent`) or any line in the order (in case of `SubscriptionActiveEvent` or `SubscriptionInactiveEvent`). OrderLineDTO contains most data regarding the order line, such as item, quantity, amount, price, etc.

Returning to our sample cell phone operator, we would need to capture our business conditions (which are service activation and deactivation) in two business rules:

```
rule 'ServiceActivation'
when
    CommandManager ( eventType == "activated" )
    $line : OrderLineDTO ( provisioningStatusId ==
        Constants.PROVISIONING_STATUS_INACTIVE )
then
    command.addCommand("activate_service", $line.getId());
    command.addParameter("number", $line.getDescription());
end
```

```
rule 'ServiceDeactivation'
when
    CommandManager ( eventType == "deactivated" )
    $line : OrderLineDTO ( provisioningStatusId ==
        Constants.PROVISIONING_STATUS_ACTIVE )
then
    command.addCommand("deactivate_service", $line.getId());
    command.addParameter("number", $line.getDescription());
end
```

The "`ServiceActivation`" rule takes care of mapping service activation events. We're interested in "`activated`" `eventTypes` for order lines whose provisioning status is `INACTIVE` (this serves to filter out events for subscriptions that have already been activated in the past). We pass a single command to the external provisioning module ("`activate_service`") with a single parameter (the "`number`" parameter, containing the cell phone number, which for the sake of simplicity of the example has been retrieved from the description property of the order line).

The "`ServiceDeactivation`" rule takes care of the opposite situation: mapping service deactivations signaled for subscriptions that are still in the `ACTIVE` state. This translates to the single command "`deactivate_service`", with the single parameter "`number`" containing the cell phone number.

In this case, we don't enforce a specific item ID in the conditions of these rules as we're assuming that our simplified cell phone operator has only one item which maps to the only rate it provides. This is all we need to capture the two simple conditions that are relevant to our contrived cell phone operator. A "normal" service would probably need several rules filtering out changes in rates or plans, recharges of prepaid traffic, failures in the provisioning calls, etc.

Once all provisioning rules have been evaluated and the `CommandManager` object has been populated with the commands and parameters to be passed to the external provisioning module, the plug-in proceeds to generate a message that will be posted to the outbound JMS queue.

The provisioning queues

All JMS messages transit in queues defined at system level. By default, **jBilling** uses Apache ActiveMQ as JMS provider. The queues defined by default on the standard installation are `queue.jbilling.provisioningCommands` as the outbound queue and `topic.jbilling.provisioningCommandsReply` as the inbound queue.

Of course, you can choose a different JMS provider or change the name of the queues if needed. Queue parameters in **jBilling** are abstracted inside a Spring configuration file named `jbilling-jms.xml`, located under the `jbilling/conf` directory.

Please refer to the Spring documentation to determine how to configure a different JMS provider or change the queue names (although the latter is seldom necessary).

The external provisioning module

The external provisioning module services the outbound JMS queue and takes care of converting the JMS message generated by the provisioning plug-in (or some other client) into a series of commands that will be forwarded to the external system.

Commands describe the interactions with the provisioned system that must take place in order to service a provisioning event. Commands can have a series of parameters, a set of name-value pairs indicating the name of the parameter and its value.

The first activity carried out by the external provisioning module is that of decoding the input message and creating a command map that describes the commands received. This command map translates into a simple `List<String,String>` object that contains the list of name-value pairs of all parameters passed to the external module.

The Message Mapper must be configured to establish the commands and parameters that our external provisioning module accepts and validate incoming messages and responses. Configuration of the Message Mapping component is the subject of the next section.

The external provisioning module pushes this command map through a stack of external provisioning plug-ins (which follow the standard **jBilling** plug-in conventions). These plug-ins take care of processing the provisioning request and performing the actual interaction with the external systems. This is where most of the coding effort goes when implementing a provisioning service. We will describe the creation of external provisioning plug-ins later in this chapter, for the moment we're just interested in knowing that the message mapper will receive the message generated by the rules in the internal module and relay the command map to our plug-in.

Configuration of the Message Mapping component

As previously noted, the Message Mapping component of the external provisioning module needs to know what commands and parameters can be passed to the provisioning plug-ins. This serves to validate incoming messages (remember that the external provisioning module could receive messages from other sources, and therefore messages could be garbled or incorrectly formed).

The file that containing configuration data for the message mapper is located under `jbilling/conf` and is named `jbilling-provisioning.xml`. This file is a Spring bean definition file and describes the mapping of a JMS message into commands.

The command translation logic is divided into commands. Each command describes a set of interactions and defines a set of parameters that will be passed to the provisioned system. It also defines a set of processors, which are the entities that describe the interaction with the external system.

Please refer to the Spring documentation if you need further information regarding bean factories and their use. Let's concentrate on describing the functionality of the configuration parameters and their use. The following demonstrates the configuration for the "activate_service" and "deactivate_service" commands introduced in the rules created for our sample cell phone operator.

```
<beans>
  <bean id="provisioning"
class="com.sapienter.jbilling.server.provisioning.config.Provisioning">
    <property name="commands">
      <list>
        <bean
class="com.sapienter.jbilling.server.provisioning.config.Command">
          <property name="id" value="activate_service" />
          <property name="fields">
            <list>
              <bean
```

```
class="com.sapienter.jbilling.server.provisioning.config.Field">
                <property name="name" value="number" />
              </bean>
            </list>
          </property>
          <property name="processors">
            <list>
              <bean
class="com.sapienter.jbilling.server.provisioning.config.Processor">
                <property name="id" value="sample_cell_service" />
                <property name="requests">
                  <list>
                    <bean
class="com.sapienter.jbilling.server.provisioning.config.Request">
                      <property name="order" value="1" />
                      <property name="submit"
value="newSubscription:number,|number|;" />
                      <property name="postResult" value="true" />
                    </bean>
                  </list>
                </property>
              </bean>
            </list>
          </property>
        </bean>
        <bean
class="com.sapienter.jbilling.server.provisioning.config.Command">
          <property name="id" value="deactivate_service" />
          <property name="fields">
            <list>
              <bean
class="com.sapienter.jbilling.server.provisioning.config.Field">
                <property name="name" value="number" />
              </bean>
            </list>
          </property>
          <property name="processors">
            <list>
              <bean
class="com.sapienter.jbilling.server.provisioning.config.Processor">
                <property name="id" value="sample_cell_service" />
                <property name="requests">
                  <list>
                    <bean
class="com.sapienter.jbilling.server.provisioning.config.Request">
                      <property name="order" value="1" />
                      <property name="submit"
value="cancelSubscription:number,|number|;" />
                      <property name="postResult" value="true" />
                    </bean>
                  </list>
                </property>
              </bean>
            </list>
          </property>
        </bean>
      </list>
```

```
    </property>
  </bean>
</beans>
```

Both commands above define a single processor named "`sample_cell_service`" in their "`processors`" property. The name of the processor is used to identify the external provisioning plug-in that will take care of servicing the interaction.

The "`sample_cell_service`" processor defines two requests to the external system. Requests have an "`order`" attribute that indicates the order in which they should be executed.

Request property "`submit`" indicates the interactions that will be posted to the external system (its "`value`" attribute contains the command that will be passed to the plug-in, such as "`newSubscription:number,|number|;`" (note the parameter placeholders such as "`|number|`" that indicate the location of the command in which parameter values will be expanded when a JMS message is translated). This specific example maps to a command named "`newSubscription`" with a parameter named "`number`".

Request properties have two more properties that can be configured: "postResult" is a boolean property that controls whether the command response is posted back to the inbound JMS queue, and "continueOnType" indicates the value of the response result that represents a valid transaction and allows the plug-in to proceed with the next request command, if present. If no "continueOnType" property is specified, execution will stop at this request.

The following table describes all attributes of the configuration file in greater detail:

Tag	Required	Multiplicity	Description
`provisioning`	Yes	1	Root tag.
`command`	Yes	1	Requires an id attribute, to identify this command. The value of the id attribute will map to the 'command' parameter received in the JMS message.
`field`	No	0..*	Each field contains the name of a parameter passed in the JMS message. The 'defaultValue' attribute is optional, if present, its value is used only if the field is not present in the JMS message, or its value is null or empty.
`processor`	Yes	1..*	The id attribute is required and will allow the MDB to find the right plug-in. Every plug-in has to have the 'id' parameter present.
`request`	Yes	1..*	The order attribute is required and allows the requests to be sorted for execution.

`submit`	Yes	1	The string that will be sent to the plug-in, defining placeholders for the input parameters as needed.
`response`	No	0..1	Only groups the response related tags
`post_result`	No	0..1	If present, then the MDB has to post the results on the inbound queue.
`continue_on_type`	No	0..1	If not present, then do not continue to the next request. If present, then continue only on the type specified. Every call to a plug-in has to return the field 'result'. If the value of that field matches the value of this tag, continue to the next request.

The external provisioning plug-in

The specific plug-in that handles the command execution takes care of creating the connection with the external system as necessary (opening a telnet session or whatever other operations are necessary to establish communications with the external system).

There are two external provisioning plug-ins shipped with **jBilling**: CAI and MMSC, covering two use cases for the plug-in (one using a telnet session with the external system, the other relies on web service calls to interact with the MMSC service). Most probably, you'll need to implement your own plug-in to handle communications with the external system, but these two could serve to illustrate the way in which the plug-in service the command map passed by the message mapper.

External provisioning plug-ins have a plug-in category of their own (category 18, see the Extension Guide for further information on plug-in categories and their role in plug-in configuration). They follow the same principle to which all **jBilling** plug-ins must adhere: extend the abstract class `PluggableTask`. In addition, they need to implement the `IExternalProvisioning` interface, which mandates implementation of two methods:

```
Map<String, Object> sendRequest(String id, String command) throws
TaskException
```

and

```
String getId()
```

The value returned by the plug-in's `getId()` method must match the processor's `id` property indicated in the message mapping configuration. This is how the message mapper identifies which processors to invoke for each specific command, and allows you to separate commands in different plug-ins (for example, if you need to feed two different external systems for the same provisioning event).

The `sendRequest()` method is the one that actually performs the interaction with the external system. When called, its `id` parameter contains the GUID of the JMS message to uniquely identify the originating JMS message, whereas the `command` parameter contains the resulting string that was constructed based on the XML configuration described in the previous section. All the parameters (enclosed in '|') have been parsed so you get the real values.

How are the command parameters passed to the plug-in? The plug-in receives a single string that follows the format defined in the above configuration. Thus, it is the plug-in responsibility to extract the parameters from this string.

The plug-in parameters (do not confuse with the command parameters. The plug-in parameters are used for every call and do not change in between calls) are typically used to store data needed to reach the external system (port numbers, IP addresses) and establish a session (user name, password...).

We can now finally code the plug-in for our sample cell phone operator. Here, we simply ignore the details of the connection to the external system, as actual implementations vary greatly (telnet sessions, web services, RESTful HTTP calls, etc.). We will just flesh out the code that provides the connection and assume a generic `externalCall` object with two methods (`newSubscription()` and `cancelSubscription()`, each accepting a single number argument and returning an int indicating the outcome of the operation: 1 if successful and 0 if unsuccessful).

```
class SimpleProvisioningTask extends PluggableTask
        implements IExternalProvisioning {

   private static final String PROCESSOR = "sample_cell_service";

    public String getId() {
        return PROCESSOR;
    }

    public Map<String,Object> sendRequest(String id,
            String command) throws TaskException {

        // Obtain and validate the "number" parameter
        // Here we use a helper method that fetches the value
        // of number from the command ex:
        //    cancelSubscription:number,123456789;
        String number = getCommandParameter(“number”, command);
        if (number == null || number.equals("")) {
            throw new TaskException("invalid number");
        }

        // The command string contains the command names indicated
        // in the mapping configuration:
        //    - addSubscription
        //    - cancelSubscription
        int result = 0;
        if (command.startsWith("addSubscription")) {
            result = externalCall.newSubscription(number);
        } else if (command.startsWith ("cancelSubscription")) {
            result = externalCall.cancelSubscription(number);
        } else {
            throw new TaskException("Invalid command " + command);
        }
        Map<String,Object> resultMap = new HashMap<String,Object>();
        resultMap.put("result", Integer.valueOf(result));
        return resultMap;
    }
}
```

Note that a single provisioning event could involve more than one call to the provisioned system, or even parallel calls to separate external systems. Each command is encapsulated in a separate Request in the mapping configuration.

If the request has been configured with a `postResult` property with a "`true`" value, the response provided by the external system will be propagated by means of the JMS inbound queue. The response JMS message will contain all input parameters that were present in the input JMS message (prefixed by "`in_`") and all output parameters generated by the external system (prefixed by "`out_`").

Let's see the output message mapping in action. Assuming an input message containing the following fields:

Name	Value
id	fffa1ea0-a44a-11dd-aa93-0002a5d5c51b
command	activate_service
number	6041231234

The generated output message will contain the following fields:

Name	Value
in_id	fffa1ea0-a44a-11dd-aa93-0002a5d5c51b
in_command	activate_service
in_number	6041231234
out_result	success

Error handling

In case of errors during execution of the external provisioning plug-in, caused by lack of availability of the connection to the external system or other conditions, the external provisioning module nevertheless provides an error response. This response has the '`out_result`' property populated with an "`unavailable`" value, and an "`exception`" property containing the exception message that details the type of error condition raised during execution of the plug-in. An error message for the previous example would look like the following:

Name	Value
in_id	fffa1ea0-a44a-11dd-aa93-0002a5d5c51b
in_command	activate_service
in_number	6041231234
out_result	unavailable
exception	NullPointerException