JEFF (Java Explanation Facility Framework)

Tutorial (v. 1.0)

Boris Horvat, Nemanja Jovanović and Bojan Tomić

November, 2010.

Table of Contents

1 Introduction

JEFF (Java Explanation Facility Framework) is an explanation facility framework written in Java. Explanation facilities date from the era of expert systems (ES) where they were used in order to provide an explanation about the inference process. The explanation they provided was supposed to clarify how the ES reached its conclusions (the "HOW" explanation) or why it asked some question during fact acquisition (the "WHY" explanation). Some authors suggest that there is a third type of explanation that unveils the strategic decisions that affected the inference process (the “STRATEGY” explanation). Nowadays, traditional ES development environments ("shells") are replaced by rule engines (RE) and business rule management systems (BRMS) which seem to lack explanation facility functionality intended for non-technical end users. JEFF was created in order to remedy this.

1.1 Goals and audience

The main goal of JEFF project is to provide an open-source and free (for use, development and distribution) explanation facility framework in Java. It should be:

- Simple to use
- Able to provide explanations in "natural-language-like" sentences making them easy to read and comprehend
- Able to enrich explanations with other content besides text (images, data etc.)
- Easy to integrate and use with various existing ES, RE and BRMS written in Java
- Able to provide explanations in different languages with no framework modifications
- Able to provide explanations in the form of reports in various output formats
- Easy to add new features without major modifications to the existing elements (extendable)

It is important to note that JEFF is, at this point, intended to provide the "HOW" and “STRATEGY” explanations but not the “WHY” explanation. The first one is a step-by-step explanation on how the ES reached its conclusions during the inference process. The second one is inteded to clarify strategic decisions that affected the inference process. Also, it is important to know that REs and BRMSs rarely query the user directly by asking questions. In most cases, they gather facts from databases, live business processes etc. so there is no need for a "WHY" explanation.

The intended audience for this project is from the realm of scientific and educational, but is by no means limited to them. Developers and other business users are most welcome to exploit JEFF in their projects.

1.2 Features

At this point, JEFF is licensend under the **LGPL v3 license** and has the following main features (implemented in accordance with project goals):

- Usage with any enviroment which can call Java methods
- “HOW” and “STRATEGY” explanations for the end user
- Canned text with insertion of dynamic values in certain places

- Rule trace
- Data and images can be inserted into the explanation
- Internationalization of explanations
- XML, PDF and plain text as output formats
- Output can be sent to a file or forwarded to an output stream

First of all, JEFF is implemented as a Java framework. Therefore, the only prerequisite for using JEFF is the ability to make Java method calls. Some of the leading REs, BRMSs and ES shells have this option, and this is why this approach was taken. At this point, JEFF has no graphical user interface, and can be used only through its API.

As stated, JEFF provides "HOW" and "STRATEGY" explanations for the end user. This is its main goal. The explanation is in the from of "natural-language-like" sentences which means that it can just be read and understood. But, JEFF can also be used by knowledge engineers for debugging purposes.

Regular explanations in JEFF are formed by employing canned text – predefined sentences which are just inserted when necessary (i.e. "The weather is sunny, so no rain is expected"). In this case, canned text can be customized by inserting dynamic values in certain places. For example, "The temperature is {temp}F and the weather is sunny, so no rain is expected" sentence has one dynamic value (the current temperature - in curly brackets) which can be inserted at runtime in order to adjust the explanation to the current list of facts. When the explanation is finally completed, this sentence would look something like this: "The temperature is 100F and the weather is sunny, so no rain is expected".

Besides canned text, JEFF uses rule trace to record which rules were executed in what order. This trace is ommited from the explanation by default, but can be turned on if, for example, knowledge engineers need to use the explanation for debugging purposes.

JEFF explanations can contain images and data. This is very important as many explanations need some form of graphic representation in order to be clear and/or precise. Images can have captions, and data is presented in the form of tables. Graphical representation of data is not yet available but is planned for the future.

Internationalization (i18n onward) is maybe one of the most important features in JEFF. This simply means that the explanations can be translated into different languages without the need for changing any of the framework structure or elements. All of the content that needs to be translated (canned text, image captions, data labels etc.) is stored in property files. Adding support for a different language means just adding a set of property files that contain translated content, and everything else is automatic.

A completed explanation in JEFF can be presented as plain text, PDF or XML. The first two formats were intended to be used directly by end users and the third enables a lot of flexibility if there is a need to do some special formatting, transforming etc. In all cases, end users get a report which can just be read. Images can be displayed only within PDF reports, so plain text and XML reports just contain references to images. For PDF and plain text reports data is presented in the form of tables, whereas XML reports contain a set of tags with values representing data.

Finally, these plain text, XML and PDF reports can be saved as files, or can be forwarded to an output stream. The latter enables JEFF to directly output some report to a servlet, JSP page, standard output stream (console) etc.

1.3 Basic algorithm

The basic algorithm for JEFF usage can be described in four major steps (Illustration 1).

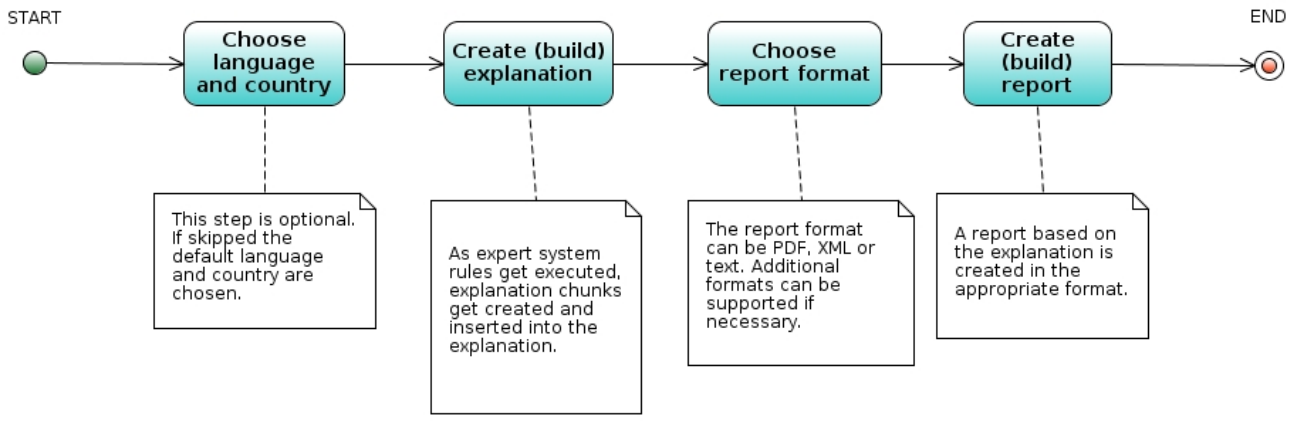

Illustration 1: JEFF - basic algorhitm

The first step is to choose a language in which the explanation should be written. When choosing a language it is also important to define the country for which it is intended because of various dialects one language can have (i.e. American English and British English). This step is optional and should be used only if there is a need to be able to provide explanations in various languages. If omitted, a default language and country are chosen and internationalization as a feature is turned off.

The second step is more important as it involves interaction with the ES, RE or BRMS. When the inferencing process starts, and rules get executed, little pieces of explanation (explanation chunks) get created and inserted into the explanation. The idea is to form the complete explanation step by step and to explain all important conclusions as they appear. The connection between the ES, RE or BRMS and JEFF is made in the rules themselves. When a rule gets executed, it is supposed to call a JEFF method for inserting an explanation chunk into the explanation. Some leading REs and BRMSs written in Java have the option of using Java objects and calling Java methods within the rules themselves and this is why this approach was chosen.

When the explanation is created, it can be transformed into a report. The third step is to choose a report format such as plain text, XML or PDF. At this moment, only these three are supported, but additional formats can be introduced if necessary.

The final step is to create a report (in the desired format) based on the explanation. Generally, this involves transforming the explanation into a text file, XML file or PDF file. Optionally, an output stream or writer can be passed to JEFF so the report (text, XML or PDF) can be served to the standard output stream ("console"), browser (via servlet or JSP) etc.

1.4 About this tutorial

This tutorial is intended to provide a short and simple introduction into JEFF and how it is supposed to be used. It is by no means comprehensive and exaustive and does not explain JEFF classes in detail. For this, we refer you to the JEFF API documentation.

The idea behind this tutorial is to explain JEFF through an example ES implemented in Drools[1]. Probably the best way to use this tutorial is to download the example and follow step by step instructions provided in the next sections.

The prerequisites, of course, are that the reader knows Java, is familiar with the Eclipse IDE[2] and posseses some basic knowledge on the workings of Drools. As far as software goes, readers need to have the Eclipse IDE and Drools 5.0 (or higher) installed in order to run the example that is provided. Also, all of the JAR files distributed with JEFF need to be somewhere in the classpath. Since the example provided with this tutorial is in the form of an Eclipse project, the easiest way to do this is to include these files in the project as libraries.

2 The Wine Advisor prototype example

What we have chosen is a well known ES - the wine advisor prototype[3]. It can be used to help someone choose some wine for dinner when he/she doesn't know how to make a selection. The knowledge base for this ES was acquired as a textual listing[4] and a part of it has been implemented in Drools 5.0 as a Drools project in the Eclipse IDE. This is what makes the basis of the example that will be used throughout this tutorial. Drools is a business rule management system (BRMS) with a forward chaining inference engine tailored for the Java language and has a plugin for the Eclipse IDE.

As stated, this expert system can help someone make a decision what wine to buy for what meal. In order for the expert system to come up with a decision, it needs to know a couple of things i.e. gather a few facts. One of the first things is what type of meal the wine is meant for. It can be any of the following:

- to be consumed before a meal (aperitif)
- to accompany cheese
- to accompany an entrée
- to be consumed with dessert or after dinner

The next step depends on the previous selection, so if the wine is to be consumed before a meal (aperitif) then the user needs to tell his/hers “wine body”[5] preference. However, if the wine is supposed to accompany cheese, it is important to detail on what type of cheese is to be served: a variety of cheeses, primarily mild cheeses or primarily strong cheeses. If, on the other hand, the wine is to accompany an entrée, the type of entrée needs to be defined: cold meats, fish, game, Italian meat, lamb, light meat (port, veal), red meat or shellfish. And, in case of dessert, the user needs to enter the type of dessert: is it very sweet (such as chocolate), fruit or primarily fruit.

1 Drools, http://www.jboss.org/drools

2 The Eclipse IDE, http://www.eclipse.org

3 Wine advisor prototype, http://www.expertise2go.com/e2g3g/wine/

4 Wine advisor prototype knowledge base, http://www.expertise2go.com/e2g3g/e2g3gdoc/wine.kb

5 “An aspect of the taste of the wine that describes how heavy it feels on the palate. Heavier wines are described as full-bodied: other designations include light and medium-bodied”, as cited in: http://www.expertise2go.com/e2g3g/wine/

Based on these conditions it is up to the expert system to come up with some recommendation for a wine. It can suggest any of the following: Riesling, Port, Dry sherry, Champagne or sparkling white, Sauvignon Blanc, Red burgundy, Chianti, Chardonnay, rose, Chablis, Merlot or Pinot noir.

Now that we know what the expert system will be used for, the first thing that needs to be done is to see the rules that will be used for drawing conclusions in our example. In addition regular rules, there are also (so called) meta-rules which are used to guide the overall inference process. All rules are stored in the "wine_selection_rules.drl" file.

Here is an example of regular rule:

```
rule "Sweet dessert"
      no-loop
      agenda-group  "dessert"
when
   wr: WineRequest (consumationTime == "to accompany dessert" &&
                            dessert == "very sweet such as chocolate" &&
                            recommendedGenericWineType == null)
then
    wr.setRecommendedGenericWineType("Port");
    System.out.println(
    "The rule that was executed was \"Sweet dessert\" from the \"dessert\" group");
    update(wr);
end
```

The rule states that, if the wine is supposed to be served with a very sweet dessert, the recommendation would be Port. The name of the rule is "Sweet dessert" (after the reserved word "rule"), the group to which the rule belongs to is "dessert" (after the reserved word "agenda-group"[6]), the condition that needs to occur for the rule to be triggered is placed after the reserved word "when" and what happens when the condition is fulfilled is placed after the reserved word "then". The condition is checked by retrieving attribute values from the "wr" object - an instance of the WineRequest class.

An example of meta-rule[7] would be:

```
rule "Activate only rules for dessert wines"
      no-loop
when
      wr: WineRequest (consumationTime == "to accompany dessert")
then
      Drools.setFocus("dessert");
end
```

Here, the Drools inference engine is notified to activate only a specific group of rules that refers to dessert wines (in this case the rules of the agenda group "dessert").

Now it is necessary to initiate the expert system. This is done by loading the knowledge base into the system, after which the initial facts are given to it. In the end the conclusion, or recommendation, that was made by the expert system can be shown. All of this is done in the StartWineSelection class (see the "StartWineSelection.java" file).

Loading the knowledge base:

```
KnowledgeBase kbase = readKnowledgeBase();
```

6 Rule groups are generally used in order to divide very large knowledge bases into manageable pieces.

7 Meta-rules are often utilized in order to guide and optimize the inference process by activating only certain parts of the knowledge base (certain rules) and thus focusing the inference on the most likely solutions. In Drools, meta-rules can be used together with rule groups (agenda groups) and can activate or deactivate certain rule groups.

```
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
KnowledgeRuntimeLogger logger =
            KnowledgeRuntimeLoggerFactory.newFileLogger(ksession,"test");
```

First, it can be seen that the loading of the knowledge base is done through a call to a private method ("readKnowledgeBase"). This method is automatically generated by Drools and, in most cases, the only part that is subject to change is the following line because it points out the location of the knowledge base.

```
ResourceFactory.newClassPathResource("wine_selection_rules.drl")
```

The second step is to define objects that will be used as initial facts.

```
WineRequest wr = new WineRequest();
wr.setConsumationTime("to accompany dessert");
wr.setDessert("very sweet such as chocolate");
```

Here it can be seen that we are working with a WineRequest instance ("wr") and that we are setting up values based on which the ES will come to conclusions. These values refer to consumation time (when will the wine be consumed) and dessert type (what type of dessert is served after the meal).

After this, we insert this Wine Request instance into the ES session (working memory) and we launch Drools.

```
ksession.insert(wr);
ksession.fireAllRules();
```

When the ES comes to a conclusion, the derived facts are stored in the WineRequest instance that we already used to store initial facts. Then, we can show[8] these conclusions in the standard output stream ("console"):

```
  if (wr.getRecommendedGenericWineType() != null)
      System.out.println("Recommended generic wine type: "
                              + wr.getRecommendedGenericWineType());
  if (wr.getSuggestedVarietalWine() != null)
      for (String wine : wr.getSuggestedVarietalWine())
                  System.out.println("Varietal wine: " + wine);
```

Besides these few steps that need to be taken in order for the ES to work, there is one more thing that we omitted to mention, and that is to define classes that will work as fact carriers. These classes should be filled with information about the problem (facts) and the ES should be able to find a solution using the rules from the knowledge base and these facts. In our example, this is a single class – WineRequest.

```
public class WineRequest {
      //Initial facts supplied by the user
      private String consumationTime = null;
      private String preferredBody = null;
      private boolean sparklingWine = false;
      private String entree = null;
      private String preferredWineColor = null;
      private String preferredTaste = null;
      private String dessert = null;

      //Intermediate conclusions
      private String entreeCategory = null;
```

8 Since in our example we have two types of recommendation we need to check for both of them, the suggested varietal wine as well as recommended generic wine

```
        //Goal facts
        private String recommendedGenericWineType = null;
        private String[] suggestedVarietalWine = null;

        //Getters and setters...

}
```

This is not the complete code listing of this class as it also needs to have the appropriate get and set methods for each attribute. This is because Drools uses them in order to retrieve values from these attributes.

3 Introduction to the JEFFWizard

Although JEFF can be used in several ways, the easiest (and at the same time the best way) would be trough a single class - JEFFWizard. This class is used as a facade for JEFF, i.e. it is in charge of connecting and creating all necessary objects and providing all functionalities. To be more exact – this is all that an average user needs to make an explanation in any ES that supports Java.

3.1 Initialization

Before we start adding explanation chunks it is necessary to initialize the JEFFWizard. However, for that to be done, the first thing that the user needs is to be familiar with a feature that is known as internationalization (i18n). This option enables translation to any language without changing the program structure. JEFF uses the Java internationalization[9] feature and translations are stored in property files. More about the i18n and property files will be explained in the following sections. In accordance with this, the procedure of initialization depends on whether we want to create a wizard which will use i18n or not. Therefore we have 4 types of constructors.

Empty constructor – sets all values to null and option i18n to false.

```
JEFFWizard()
```

Constructor that receives only the name of the owner, the rest of the values sets to null and option i18n to false. The owner of the report is the person (or some entity) for whom the explanation is intended for. This information is optional, and can be ommited.

```
JEFFWizard(String owner)
```

Constructor that receives the name of the owner as well as the explanation title. The rest of the values are set to null and option i18n is set to false. If set, the title will be displayed in the final explanation. The title is optional, and can be ommited.

```
JEFFWizard(String owner, String title)
```

Constructor that receives the name of the owner, the explanation title and the option of i18n, the rest of the values sets to null.

```
JEFFWizard(String owner, String title, boolean internalization)
```

Constructor that receives the name of the owner, language, country, title and the option of i18n.

```
JEFFWizard(String owner, String language, String country, String title, boolean
internalization)
```

9 For more information about i18n in Java please visit http://java.sun.com/docs/books/tutorial/i18n/index.html

On the given example, the JEFFWizard instance is initialized as follows:

```
new JEFFWizard("Bojan Tomic", "srb", "RS", "Wine recommendation", true);
```

It can be seen that the name of the owner is "Bojan Tomic", language is Serbian ("srb"), the name of the country is Republic of Serbia ("RS"), the title is "Wine recommendation" and the i18n option is set to true.

Once set, the values can be changed by calling the appropriate set methods. For example the method for changing the owner would be

```
setOwner("Boris Horvat");
```

Note that the change can be done at any time until the process of creating explanation starts. This process starts by calling the method:

```
createExplanation();
```

After the call to the "createExplanation" method, the wizard is fully initialized and can be used for creating explanations.

3.2 Making the wizard available to the ES

After we have initialized a JEFFWizard instance and called the "createExplanation" method, it is necessary to make it available to the ES, so the ES can use it by calling its methods. In Drools, this is done by inserting the wizard into the ES session. Since this object needs to be accessible to all rules in the knowledge base, it should be inserted into the session as a global variable:

```
ksession.setGlobal("wizard", wizard);
```

Of course, it is now necessary to reference this global variable from the appropriate DRL file (or multiple files if the knowledge base consists of more than one). In our example, the following line is added to the top of the "wine_selection_rules.drl" file:

```
global JEFFWizard wizard;
```

The wizard is now accessible to all rules in the knowledge base.

4 Inserting content into the explanation

Now that we have created the wizard and made it accessible to the ES, all that is left to do is to fill the explanation, step by step, by adding individual explanation chunks. As stated at the beginning, the idea is that whenever an ES comes to a certain conclusion, the explanation of that conclusion is added to the explanation as an explanation chunk. When the inferencing is complete, the explanation should clarify how the conclusions were made and, in some cases, what were the strategic decisions on behalf of the ES that guided the inference process. This is achieved by calling the appropriate JEFFWizard methods in the then part of the rule (see the following rule example).

```
rule "Sweet dessert"
      no-loop
      agenda-group  "dessert"

      when
            wr: WineRequest (consumationTime == "to accompany dessert" &&
```

```
                                    dessert == "very sweet such as chocolate" &&
                                    recommendedGenericWineType == null)
      then
            wr.setRecommendedGenericWineType("Port");

            Object [] content = new Object[1];
            content[0] = "Port";

            wizard.addText( "dessert", "Sweet dessert", null, content);

            update(wr);
end
```

As we can see on the given example, right after the conclusion has been made:

```
            wr.setRecommendedGenericWineType("Port");
```

an explanation of that conclusion was generated by the following commands

```
            Object [] content = new Object[1];
            content[0] = "Port";

            wizard.addText( "dessert", "Sweet dessert", null, content );
```

This is just one of the JEFFWizard methods, and it is used for adding new textual explanation chunks into the explanation. JEFFWizard class contains a lot of methods, and all that you need to know for now is that this method will add a new chunk of text into the explanation. It will also remember the rule that triggered it, as well as the group that this rule belongs to and also the content[10] of that explanation.

At the moment, there are three types of content that can be added to the explanation.

- Text - it is most commonly used and represents content that is in textual form.
- Image – represents a picture that needs to be inserted into the explanation
- Data – represents data that needs to be added into the explanation (for example price lists, performance data tables etc.)

Although there are only three types of explanation chunks based on the three types of content that JEFF supports at the moment, new types can be easily added through subclassing (for this, see the JEFF API documentation).

4.1 Inserting text into the explanation

Based on the type of the explanation chunk, different methods are used for adding different content. Adding textual explanation chunks is probably most complicated, not due to the complexity of the methods, but because there are so many method variations that can be used.

In JEFF, each explanation chunk has something that is called context which points out to the meaning of the chunk. There are eight types of context:

- Informational – this is the default context and represents content that is intended just to inform the user about something (e.g. “There is nothing like a glass of Port after dinner.”)

10 The textual content will actually be formed later on by finding the adequate property file that holds the explanations for the desired langugage and country. The appropriate text chunk is to be identified by the rule name and group name. Then, the content will be completed by inserting all dynamic values into it. In this case it is the first element of the objects’ array (“content[0]”). Finally, this textual content it will be inserted into the complete explanation.

- Warning – when you want to emphasize that the content warns you about something (e.g. “The reactor core is starting to overheat.”)
- Error - when you want to emphasize that the content points out to a certain mistake (e.g. “The money transfer failed due to unknown error.”)
- Positive - when you want to emphasize that the content points out to a positive conclusion (e.g. “The company has made a profit of $ 10,000.”)
- Very positive - when you want to emphasize that the content points out to a very positive conclusion (e.g. “The company has made the profit of $ 500,000.”)
- Negative - when you want to emphasize that the content points out to a negative conclusion (e.g. “The company has a loss of $ 10,000.”)
- Very negative - when you want to emphasize that the content points out to a very negative conclusion (e.g. “The company has a loss of $ 500,000.”)
- Strategic - when you want to emphasize that the content points out to strategic conclusion made by the ES and that this strategic conclusion affects future inferences of the ES (e.g. “The problem with your car is most likely in the electric system.”). This is how the “STRATEGY” explanation is formed.

When it comes to context, two things need to be clarified. First of all, the main idea is to be able to distinguish different meanings between explanation chunks and to enable rearranging the explanation chunks based on their context (it may seem natural that errors and warnings should be displayed at the beginning of the explanation) or displaying them in a different manner (it may also seem natural that errors should be marked with bold fonts and in red color). JEFF hasn't yet got the option of rearranging or manipulating explanation chunks, and they appear in the order they were inserted. But the XML report it generates can easily be manipulated through XSLT based on the provided contexts.

Second, all context markings are optional, and very subjective. One may decide not to use context markings, which is fine – every chunk has informational context then. On the other hand, what may seem, let's say, positive to some, may seem negative for others. For example, the conclusion “The company has made a profit of $ 10,000.” may seem positive for small companies, but negative for larger companies as they see this as very small profit. It is, therefore, left to the knowledge engineer to decide whether to use contexts, and what to use each context for.

When adding text explanation chunks, all of these contexts[11] are represented. Therefore, every method that inserts text explanation chunks can be found in seven different forms depending on the context. This can be seen on a given example.

```
public void addText (Object content) – Informational [12]context

public void addTextWarning (Object content) – Warning context

public void addTextError (Object content) – Error context

public void addTextPositive (Object content) – Positive context

public void addTextNegative (Object content) – Negative context

public void addTextVeryPositive (Object content)  –  Very positive
```

11 In methods for inserting pictures and data not all are represented

12 This is the only context that isn’t emphasized in method’s signature, because it is considered to be default

```
public void addTextVeryNegative (Object content) -  Very negative

public void addTextStrategic (Object content) -  Strategic
```

As already mentioned, the “addText” method can have seven forms, depending on the context. However, there can also be four additional variations of each form depending on the method parameters. You may have already seen some of them, but we will now explain each of these four types.

The first variation is the simplest one and has only one parameter – content. Being that this method is for adding textual explanation the content that is being added has to be either a String or an array of Object instances. I18n plays a significant role here and, when we are not using i18n, full textual explanations need to be passed as content (String). However if we are using i18n then only parameters that are dynamically inserted into the textual content need to be provided as an array of Object instances (more on this in the following section on property files).

```
public void addText (Object content)
```

The second variation has two parameters. Besides content, there is one more in which we insert the name of the rule that this explanation refers to.

```
public void addText (String rule, Object content)
```

The third method variation has three parameters. The first two are already mentioned above and the third represents the name of the group that the rule belongs to.

```
public void addText (String group,  String rule, Object content)
```

The fourth variation has four parameters, besides the other three already mentioned above. The fourth parameter represents an array of tags that can be used for even more comprehensive description of the explanation content. For example, it can be used to categorize explanations and to search for a specific one. Since JEFF doesn't yet have the option of manipulating or searching explanation chunks, it is perhaps best to use these tags when transforming the generated XML report with XSLT.

```
public void addText (String group, String rule, String[] tags, Object content)
```

Lets see the rule that was presented at the begining of this section again.

```
rule "Sweet dessert"
      no-loop
      agenda-group  "dessert"
when
            wr: WineRequest (consumationTime == "to accompany dessert" &&
                                    dessert == "very sweet such as chocolate" &&
                                    recommendedGenericWineType == null)
then
            wr.setRecommendedGenericWineType("Port");

            Object [] content = new Object[1];
            content[0] = "Port";

            wizard.addText( "dessert", "Sweet dessert", null, content);

            update(wr);
end
```

It is now clear that the idea is to add a text explanation chunk that refers to the conclusion that Port is the appropriate wine for this meal. The context is informational, the rule group is “dessert", the rule

itself is "Sweet dessert", and there are no tags defined, therefore their value is set to null. Since we are using the i18n feature, the content represents an array of Object instances that has only one element and its value is "Port". If we didn't use i18n, we would have to pass a String containing a full textual explanation as content. The method call would them look something like this:

```
String content =
            "If it is sweet that you like, then a glass of Port must accompany it";

wizard.addText( "dessert", "Sweet dessert", null, content);
```

If, for example, we wanted to mark this textual chunk with positive context, we would make the following method call:

```
wizard.addTextPositive( "dessert", "Sweet dessert", null, content);
```

This goes for all contexts so, if we wanted to mark this textual chunk with a negative context, we would make the following method call:

```
wizard.addTextNegative( "dessert", "Sweet dessert", null, content);
```

And, now it is important to understand exactly how i18n works in JEFF, and where does the rest of the textual explanation come from. This is the topic of the next section.

4.2 I18n in JEFF

As stated, JEFF uses Java i18n[13] so its i18n features reflect Java i18n features. First of all, all content that should be translated is stored in property files as key-value pairs. This means that adding support for different languages means just inserting a few property files with the translated content. If i18n is turned off, no property files are needed.

Second, property file names cannot be given randomly, but in an exactly defined way. JEFF uses this strict naming scheme in order to find the correct translations. First, they have to end with the extension ".properties". Also, the langugage and the name of the country play a role in the naming process. For JEFFWizard to be able to use i18n, four property files need to be present for each language that is supposed to be supported. These are the base names for these files and they will be used only if the i18n feature is on and if there is no language nor country defined at the beginning (which means that the default language is supposed to be used) .

```
text.properties – contains the translation of all textual explanations.

image_captions.properties - contains the translation of the image captions as well
                            as the explanation title translation (see the section
                            on inserting images into explanations).

units.properties – contains the translation of the unit names that are used (see
                   the section on inserting data into explanations).

dimension_names.properties - contains the translation of the dimension names (see
                             the section on inserting data into explanations).
```

Since language and name of the country play a role in the naming process, the names of the files that contain translations for the Serbian language ("srb"), which is used in the Republic of Serbia ("RS") should be (note that the language and country names are added to the base name of each file):

```
text_srb_RS.properties
```

13 For more information about i18n in Java please visit http://java.sun.com/docs/books/tutorial/i18n/index.html

```
image_captions_srb_RS.properties
units_srb_RS.properties
dimension_names_srb_RS.properties
```

If we want to add support for, let's say, Canadian (“CA”) french (“fr”), we would add files with the following file names:

```
text_fr_CA.properties
image_captions_fr_CA.properties
units_fr_CA.properties
dimension_names_fr_CA.properties
```

In case there are many rules in the knowledge base and rule groups are used for dividing it into smaller pieces, we can also divide the property files that contain textual explanations into the same groups. Therefore, instead of having just one property file for all textual explanations (“text.properties”) we would have several, each one corresponding to one rule group. In this case, the group name participates in the naming of the file. So, if we have a group of rules named "dessert", the base name for the property file containing textual explanations for this group would be (notice that the group name is added to the beginning):

```
dessert_text.properties
```

If this file was to refer to the Serbian language (“srb”), which is used in the Republic of Serbia (“RS”) its name would be:

```
dessert_text_srb_RS.properties
```

And if this file was to refer to the english language (“en”), which is used in the United states of America (“US”) its name would be:

```
dessert_text_en_US.properties
```

If the group name contains blank characters, they will be replaced with underscores. The name is also always trimmed – leading and trailing blank spaces are removed. So, if the name of the rule group is “after dinner”, the property file would be:

```
after_dinner_text.properties
```

It is important to know that all of the property files need to be placed somewhere in the classpath of the application in order for the application to be able to use them.

So, what are the property files in our example? They are placed in the “resources” package and are as follows:

Default language	**after_dinner_text.properties**	“after dinner” group textual expl.
	aperitif_text.properties	“aperitif” group textual expl.
	dessert_text.properties	“dessert” group textual expl.
	dinner_entree_text.properties	“dinner entree” group textual expl.
	dimension_names.properties	Dimension name translations
	image_captions.properties	Image captions and title transl.
	units.properties	Unit name translations
Serbian language	**after_dinner_text_srb_RS.properties**	“after dinner” group textual expl.
	aperitif_text_srb_RS.properties	“aperitif” group textual expl.
	dessert_text_srb_RS.properties	“dessert” group textual expl.

	dinner_entree_text_srb_RS.properties	"dinner entree" group textual expl.
	dimension_names_srb_RS.properties	Dimension name translations
	image_captions_srb_RS.properties	Image captions and title transl.
	units_srb_RS.properties	Unit name translations

Two groups of property files exist: the first refers to the default language (the first seven files), and the second to the Serbian language (the other seven files that end with "srb_RS"). In this case, english is the default language. You can see that by opening any of the first seven files and reviewing its content. If you set the i18n option on in the beginning and do not define any langugage and country data, the default language will be used. If you enter "srb" and "RS" as language and country, the other seven files will be used.

The second thing that can be noticed is that there are four property files which correspond to four rule groups in the knowledge base: "after dinner", "aperitif", "dessert" and "dinner entree". When textual explanation chunks get added these property files will be searched for the appropriate textual explanations.

So, what is in these property files? Property files in Java are always written as key-value pairs, and here it is the same. The key is on the left side of the equation sign and the value is on the right. In JEFF, the value represents the translation, and the key depends on what is the property file for. The one thing that applies to all keys is that they cannot have blank spaces. Each blank space should be preceeded with a backslash. For example, if the key is "property one key", it would have to be noted in the property file as "property\ one\ key".

If you open the "dimension_names_srb_RS.properties" file, you will find the following content:

```
Name\ of\ the\ beverage = Naziv alkoholnog pica
Price = Cena
```

This file contains dimension name translations and the key is the dimension name in the default language (in this case english), while the value is the translation in Serbian. There are two dimension names ("Name of the beverage" and "Price") and two translations ("Naziv alkoholnog pica" and "Cena").

If you open the "image_captions_srb_RS.properties" file, you will find the same pattern but with different content:

```
Wine\ recommendation = Preporuceno vino
A\ bottle\ of\ Port = Boca Porta
```

This file contains image caption and title translations. The title is "Wine recommendation" (see the section on JEFFWizard initialization), and the translation is "Preporuceno vino". There is also an image caption ("A bottle of Port") that is translated ("Boca Porta"). The original expression is the key and the translated expression is the value.

The same goes for property files that contain unit translations, but there are some differences when textual explanations are concerned. If you open the "dessert_text.properties" file you will see this content:

```
Fruit-based\ dessert = With any kind of fruit-baised dessert, we recommend {0} as
one of the best

Sweet\ dessert = If it is sweet that you like, then a glass of {0} must accompany
it
```

And this one is the same, only for the Serbian language ("dessert_text_srb_RS.properties")

```
Fruit-based\ dessert = Sa bilo kojim desertom na bazi voca, mi preporucijemo {0}
kao bolji izbor

Sweet\ dessert = Ako slatke deserte volite, onda casa {0} najvise prija
```

What can be deducted from these two files? The file has the same key-value pair structure and the value represents a translation in the form of canned text. But the key is not the original expression in the default language. The name of the rule is the key.

And since it is possible to dynamically add values to the canned text presented here, it is necessary to put markings (such as {0},{1},{2}...) at the places where that content needs to be inserted[14].

Lets now look at the rule example from the previous section.

```
rule "Sweet dessert"
      no-loop
      agenda-group  "dessert"
when
            wr: WineRequest (consumationTime == "to accompany dessert" &&
                                    dessert == "very sweet such as chocolate" &&
                                    recommendedGenericWineType == null)
then
            wr.setRecommendedGenericWineType("Port");

            Object [] content = new Object[1];
            content[0] = "Port";

            wizard.addText( "dessert", "Sweet dessert", null, content);

            update(wr);
end
```

When this rule gets executed, the "addText" method will also get executed. If i18n is on, and the user has chosen the default language, a series of actions will take place:

1. JEFF will look for the file named "dessert_text.properties" (based on group name)
2. It will then search that file for the key-value pair where the key is "Sweet\ dessert" (rule name)
3. It will take the matching value:

   ```
   If it is sweet that you like, then a glass of {0} must accompany it
   ```

   and insert dynamic values at the designated places. In this case, the dynamic value is "Port" ("content[0]") and the place is designated with curly braces and number zero ("{0}").
4. It will insert the finalized expression into the explanation.

   ```
   If it is sweet that you like, then a glass of Port must accompany it
   ```

4.3 Inserting images into the explanation

Similarly to the text adding procedure, there are methods that allow adding pictures as a part of the explanation. What you need to remember here is that the context itself is not so important for pictures. Therefore, there is practically only one method that we use and its context is informational. However, there can be slight differences depending on the parameters. From that point on we can distinct four

14 See java.text.MessageFormat class in the Java API for details

method variations.

The first variation is the simplest. Its only parameter is content. In this case, this is not a simple String, but an object of the ImageData [15]class (an ImageData instance is expected as an argument).

```
public void addImage (Object content)
```

The second variation has two parameters. Besides content parameter there is one more in which we insert the name of the rule that this explanation refers to.

```
public void addImage (String rule, Object content)
```

The third variation has three parameters. The first two are already mentioned above and the third represents the group to which that rule belongs to.

```
public void addImage (String group,  String rule, Object content)
```

The fourth variation has four parameters, besides the other three already mentioned above. The fourth parameter represents an array of tags that can be used for even more comprehensive description of the explanation content. As described in the previous sections, it can be used to categorize explanations and to search for a specific one.

```
public void addImage (String group, String rule, String[] tags, Object content)
```

ImageData class provides information about the picture that is inserted into the explanation. What is important to remember is that this class contains two pieces of data. The first one is the picture's URL, and it cannot be ommitted. The second is the picture's caption and it is optional. So, if we wanted to initialize an ImageData object with an URL ("/images/port.jpg") and a caption ("A bottle of Port") it would look something like this:

```
ImageData imageData = new ImageData("/images/port.jpg", "A bottle of Port");
```

Let's now suppose that we want to add a picture together with the explanation about the appropriate wine, and that we just want to modify the rule that we used as an example up to now. The rule would then be as follows.

```
rule "Sweet dessert"
      no-loop
      agenda-group  "dessert"

      when
            wr: WineRequest (consumationTime == "to accompany dessert" &&
                                    dessert == "very sweet such as chocolate" &&
                                    recommendedGenericWineType == null)
      then
        wr.setRecommendedGenericWineType("Port");

            Object [] content = new Object[1];
            content[0] = "Port";

            wizard.addText( "dessert", "Sweet dessert", null, content );

            ImageData imageData =
                  new ImageData("/images/port.jpg", "A bottle of Port");
            wizard.addImage("dessert", "Sweet dessert",null, imageData );
```

15 This class will be explained below for now all that you need to remember is that this object keeps the information about the picture that is to be inserted into the explanation

```
            update(wr);
end
```

We can see that the picture has a caption (“A bottle of Port”) and that it is to be inserted after the textual explanation chunk. The rule group is “dessert”, the rule is “Sweet dessert” and there are no tags.

It is important to note now that you can insert multiple explanation chunks per rule. The only limitation is that, if you use i18n, you can insert only one textual chunk per rule, but that's all. The reason for this is that the rule group and name are used as keys for accessing property files with textual content (see previous section). Even this limitation can be avoided if you modify the rule identifier, which is entered as an argument value, by adding some extensions etc. (“Sweet dessert 1”, “Sweet dessert 2”...). Therefore, the conclusion made in one rule can be explained by using some textual content, but also one or more images, data chunks etc.

In case we decide to use i18n, we would have to translate the image caption and place it in the appropriate property file. So, if we would like to translate a certain caption to the Serbian language (srb), and the country is Republic of Serbia (RS), the translation of the caption would be in the file

```
image_captions_srb_RS.properties
```

The translations consist of keys and their values. The key is the original (defalut) caption and the value is the translated caption. The key is on the left of the equals sign and the value is on the right (note that if the key contains blank spaces, they must be noted with a backslash sign preceeding them - “\ “).

```
A\ bottle\ of\ Port = Boca Porta
```

When the image is inserted into the explanation the value will be added to the explanation as the image’s caption.

One more thing that needs to be mentioned is that the displaying of the image depends on the type of the report. At this point, JEFF can produce reports as plain text, XML or PDF. Since the first two formats do not support images, only the URL and the caption are entered into the report, whereas PDF reports display images as intended.

4.4 Inserting data into the explanation

Data explanation chunks are intended for representing data sets in explanations. The idea is to have the option of backing up certain claims and conclusions with concrete data. There are three types of context available for data explanation chunks:

- Informational - this context is considered to be basic and neutral
- Positive - when you want to emphasize that the data points out to something positive
- Negative - when you want to emphasize that the data points out to something negative

Therefore, we have the following methods for inserting data explanation chunks:

```
public void addData (Object content)
public void addDataPositive (Object content)
public void addDataNegative (Object content)
```

Here we also have 4 different variations.

The first variation has only content as parameter, and in this case it’s not a simple string but an object

of one of the classes: SingleData, OneDimData, TwoDimData and ThreeDimData[16].

```
public void addData (Object content)
```

The second variation has two parameters. Besides content, there is the name of the rule that this explanation refers to.

```
public void addData (String rule, Object content)
```

The third variation consists of three parameters. The first two are already mentioned above and the third represents the group name which that rule belongs to.

```
public void addData (String group,  String rule, Object content)
```

The fourth variation has four parameters. Besides the other three already mentioned above, the fourth parameter represents an array of tags that can be used for even more comprehensive description of the explanation content. For example, it can be used to categorize explanations and to search for a specific one.

```
public void addData (String group, String rule, String[] tags, Object content)
```

Classes SingleData, OneDimData, TwoDimData and ThreeDimData are used to represent data that is to be inserted into the explanation chunk. Class SingleData is the simplest one. It is designed to represent only one data value, while ThreeDimData is the most complex one and it is used to show three-dimensional data arrays. It is a little hard to understand these classes at the beginning because it requires knowledge of few more dependent classes.

The first dependent class is Dimension. This class remembers two things about the data: name of the dimension (for example price) and its measurement unit name (for example euro). The dimension name is mandatory, while the unit name is optional. Therefore we have:

```
Dimension dimension = new Dimension("Price", "euro");
```

For example, if we have information that something costs 15 Euros, we can represent it by using a SingleData instance together with the already initialized Dimension instance like this:

```
SingleData singleData = new SingleData (dimension, 15);
```

The OneDimData class is used to represent one-dimensional data arrays. OneDimData also keeps information about the dimension information of this data. For instance, let's say that we want to present price growth for certain goods in Euros and the values are 24, 26, 27, 28. The dimension name is “price”, the dimension unit is “euro”, and the values are inserted as a list of Object instances. Note that you can insert any content in this list (strings, decimal values, etc.) and not just numeric values.

```
Dimension dimension = new Dimension("Price", "euro");
ArrayList<Object> values = {24, 26, 27, 28};
OneDimData singleData = new OneDimData (dimension, values);
```

To be able to use the TwoDimData class, first we have to be familiar with another “helper” class – Tuple. Being that class TwoDimData represents two-dimensional data arrays, class Tuple is used to represent one pair of values from that array (one tuple). For example, we have the name of a wine and its price:

```
Tuple tuple = new Tuple("Maryvale", "87.50");
```

Now that we have an array of pairs of values, we need to remember dimension names and units for two

16 These classes will be explained below

dimensions. For example, if we want to represent a data table containing a wine list with wine names and prices, we can achieve that like this:

```
Dimension dimension1 = new Dimension("Name of the wine");
Dimension dimension2 = new Dimension("Price", "euro");

ArrayList<Tuple> values = new ArrayList<Tuple>();
values.add(new Tuple("Maryvale", "87.50"));
values.add(new Tuple("Monterra", "103.50"));

new TwoDimData( dimension1, dimension2, values);
```

The first dimension ("Name of the beverage") refers to the first value of all Tuple objects in the list (“Maryvale”, “Monterra”), while the other dimension ("Price") refers to the second value of all Tuple objects in the list (“87.50”, “103.50”).

Being that the ThreeDimData class represents three-dimensional data arrays, class Triple is used to represent one triplet of values from that array. For example, we can have a triple representing some wine together with its price and year:

```
Triple triple = new Triple ("Maryvale", "87.50", "1973");
```

Now that we have an array of triples of values from ThreeDimData, we need to memorize three dimension names and (optionally) units. For example, if we want to show the name of the wine, its price and year we can achieve it like this

```
Dimension dimension1 = new Dimension("Name of the beverage");
Dimension dimension2 = new Dimension("Price", "euro");
Dimension dimension3 = new Dimension("Year");

ArrayList<Triple> values = new ArrayList<Triple>();
values.add(new Triple("Maryvale", "87.50","1973"));
values.add(new Triple("Monterra", "87.50","1987"));

new ThreeDimData( dimension1, dimension2, diimension3, values);
```

The first dimension refers to the first value of all Triple objects in the list, the second dimension refers to the second value of all Triple objects in the list, while the third dimension refers to the third value of all Triple objects in the list.

Finally, if we want to insert any data into the explanation, all we have to do is make an instance of SingleData, OneDimData, TwoDimData or ThreeDimData and insert it as content of the data explanation chunk.

Here is an example of a rule that refers to wines that are suitable for consumation after dinner. Besides the textual explanation, a data chunk is inserted in order to provide a price list of the appropriate wines.

```
rule "After dinner"
      no-loop
      agenda-group  "after dinner"

      when
            wr: WineRequest (consumationTime == "to be consumed after dinner" &&
                                    recommendedGenericWineType == null)
      then
            wr.setRecommendedGenericWineType("Port");

            Object [] content = new Object[1];
            content[0] = "Port";
```

```
            wizard.addText( "after dinner", "After dinner", null, content );
            wizard.addImage( "after dinner", "After dinner", null,

            new ImageData("/images/port.jpg"));

            ArrayList<Tuple> values = new ArrayList<Tuple>();
            values.add(new Tuple("Mer Soliel, \"Barrel Fermented\" Central Coast",
                  "67.50"));
            values.add(new Tuple("Maryvale", "87.50"));
            values.add(new Tuple("Monterra", "103.50"));

            wizard.addData( "after dinner", "After dinner", null,
                    new TwoDimData( new Dimension("Name of the beverage"),
                                    new Dimension("Price", "euro"),
                                         values));
            update(wr);
end
```

When using i18n only two things about data can be translated. It is the name of the dimension and its measurement unit. For this, two files are used, so if we want to find appropriate dimension name and unit translations for the Serbian language, we have to look for them in the following property files:

```
dimension_names_srb_RS.properties
units_srb_RS.properties
```

The content of the first file is:

```
Name\ of\ the\ beverage = Naziv alkoholnog pica
Price = Cena
```

And the content of the second file is:

```
euro = dinari
```

One thing that needs to be known is that, depending on the type of report, data chunks will be shown differently. When generating a PDF report, the data will be shown in the form of a table where the dimensions will represent the column headers (Illustration 2). Depending of the type of the content we used, different tables will be created (with different number of columns): for ThreeDimData we'll have a three column table, TwoDimData – two column table, OneDimData – one column table, while SingleData will also be presented with one column but will have just one row (two if you count column headers). In the textual reports the data will be shown in a similar manner where as in XML reports the data will be presented inside special tags.

CONTEXT: INFORMATIONAL
GROUP: after dinner
RULE: After dinner

Name of the beverage	Price [euro]	Year
Maryvale	87.50	1985
Monterra	57.50	1985

CONTEXT: INFORMATIONAL
GROUP: after dinner
RULE: After dinner

Name of the beverage	Price [euro]
Maryvale	87.50
Monterra	57.50

CONTEXT: INFORMATIONAL
GROUP: after dinner
RULE: After dinner

Name of the beverage
Maryvale
Monterra

CONTEXT: INFORMATIONAL
GROUP: after dinner
RULE: After dinner

Name of the beverage
Monterra

Illustration 2: Data chunks in a PDF report

5 Generating reports

Now that we have created our ES, inserted a JEFFWizard object into it and connected the rules with commands for inserting explanation chunks, we can start the ES. As rules get executed, the explanation simultaneously gets generated.

In the beginning, it was mentioned that the class JEFFWizard is responsible for managing all objects and functionalities that JEFF possesses. One of these is an Explanation class instance. This object, actually, represents the explanation which the expert system has generated. It contains information about the explanation (owner, language, country) as well as the list of explanation chunks - the explanation itself.

What is left to do now is for this object (which contains the explanation that was generated by the expert system) to be transformed into something that is more suitable for presentation. In other words,

some sort of a report needs to be generated. If, for any reason, you wish to acquire this Explanation instance, you just have to make a call to the following JEFFWizard method:

```
getExplanation()
```

There are three types of reports that can be generated automatically by JEFF:

- TXT report – it is plain text (just characters with no formatting options) and, as such, it has the advantage that it can be viewed from almost any program that can interpret text. The disadvantage would be that it is not very suitable for showing data and it cannot show images.
- XML[17] report – represents an XML file that is not suitable for reading by humans, but it is portable and easy to use by other programs. This is the format of choice if you wish to do a lot of transformations and formatting of the explanation.
- PDF[18] report – represents the most suitable format for a person to read. It is capable to show tables, pictures as well as text.

One of the most important things to remember is that the reports can be generated as actual files on the file system, but it is also possible for the created report to be forwarded to some stream. The types of stream that are used are PrintWriter for XML and TXT, and OutputStream for PDF. This way, the report can be easily forwarded anywhere, without having to create it on the spot. For instance, PDF reports can be forwarded to a web browser for showing its content.

5.1 Generating TXT reports

Generating reports is quite easy. After the ES finishes the inference process, it is necessary to choose the report type and start the automated report generation process. For TXT reports, we have two JEFFWizard methods that do this:

```
generateTXTReport(String filePath, boolean insertHeaders)
generateTXTReport(PrintWriter stream, boolean insertHeaders)
```

The first method saves the report to a text file, and the second forwards the report to an output stream. Both methods have a second parameter which indicates if chunk headers should be inserted into the report or not. Chunk headers can be used to do some debugging as they contain information on the chunk context, rule and group names as well as chunks. Regular users do not need this, so it is best to turn off this option by passing a “false” value as argument.

In our example, the following command is used to generate a TXT report file (see the “StartWineSelection.java” file):

```
wizard.generateTXTReport("Reports/textReport.txt", false);
```

The first parameter is the file path (where should the report be generated) and the second parameter indicates that the chunk headers should not be inserted into the report.

If we wish to forward this textual report to an output stream, the code would look something like this:

```
PrintWriter pwStream = //some initialization code
wizard.generateTXTReport(pwStream);
```

The textual report that was created based on the initial facts entered at the beginning follows:

---

17 For the making of this report framework DOM4J is used, for more information see http://www.dom4j.org/

18 For the making of this report framework iText is used, for more information see http://itextpdf.com/

```
Creation date: 5/13/10 5:53 PM
Report owner is: Bojan Tomic
The language used: srb
The country is: RS

Preporuceno vino

Ako slatke deserte volite, onda casa Port najvise prija

Caption is: Boca Porta
The path to this image is: /images/port.jpg
```

If we choose the default language, we will get the same report but in english (note that the language and country data is ommited as they are not entered):

```
Creation date: 5/14/10 1:16 PM
Report owner is: Bojan Tomic

Wine recommendation

If it is sweet that you like, then a glass of Port must accompany it

Caption is: A bottle of Port
The path to this image is: /images/port.jpg
```

We can see that this report contains two explanation chunks and that each one has informational context. After the context there is the name of the rule and the group which it belongs to. The last information there is the actual content of the explanation.

The first chunk is textual and therefore the context is text. Being that the option of i18n is used, the content is translated to language srb, country RS.

Pictures in the TXT files are shown only in form of information about them. In this case we have the path to the picture that is inserted into the explanation. The data is shown in a manner that looks like a table. From this we can actually see the disadvantage of TXT files – it is not possible to show pictures and the tables are not well structured.

5.2 Generating XML reports

When generating XML reports we use the following method

```
wizard.generateXMLReport("xmlReport.xml");
```

The parameter that is passed in this case is the path of the file where the report should be generated.

```
PrintWriter pwStream = new PrintWriter("Reports/xmlReport.xml");
wizard.generateXMLReport(pwStream);
```

This is the same example only a stream is passed as a parameter that points to the location on the hard drive (inside the folder Reports)

The example of the XML report that was created follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<explanation date="10.3.10. 21.56" owner="Bojan Tomic" language="srb" country="RS">
    <textualExplanation rule="Sweet dessert" group="dessert"
context="informational">
        <content>Ako slatke deserte volite, onda casa Port najvise prija</content>
    </textualExplanation>
```

```
    <imageExplanation rule="Sweet dessert" group="dessert" context="informational">
        <content>
            <imageUrl>/images/port.jpg</imageUrl>
        </content>
    </imageExplanation>
    <dataExplanation rule="After dinner" group="after dinner"
context="informational">
        <content>
            <tupleValue>
                <value1 dimensionName="Ime alkoholnog pica">Maryvale</value1>
                <value2 dimensionName="Cena" dimensionUnit="euro">87.50</value2>
                <value1 dimensionName="Ime alkoholnog pica">Monterra</value1>
                <value2 dimensionName="Cena" dimensionUnit="euro">57.50</value2>
            </tupleValue>
        </content>
    </dataExplanation>
</explanation>
```

Here we can see that this type of report is not suitable for people to read, but it is for other programs that are capable of reading XML files. Mainly for this reason this type of report is very suitable for storing explanation and for its transfer.

The first and the main tag is explanation tag which contains the basic information about the report in its attributes. Inner tags carry information about the chunks (textualExplanation, imageExplanation, dataExplanation).The associated attributes contain information about rule, group and context of the explanation chunk. The main inner tag of any chunk is the content tag. It holds information about the content of the explanation chunk.

Tag content has different inner tags depending on the type of the explanation chunk. If it is textual explanation chunk, then it only contains text. However If it is an image explanation chunk then the information about the picture is kept in a separate tag – imageUrl, and if it is a data explanation chunk then it only contains information about data. Data that are type SingleData or OneDimData are entered directly into the content tag, while the TwoDimData and ThreeDimData data are entered into tags tupelValue and tripleValue, respectively. Information regarding the dimensions is persevered as attributes of the tag that holds the data. In the case SingleData or OneDimData the content tag holds the attributes – dimensionName and dimensionUnit. In the case of TwoDimData and ThreeDimData the tags tupelValue and tripleValue hold those attributes.

5.3 Generating PDF reports

When generating XML reports we use method

```
wizard.generatePDFReport("pdfReport.pdf");
```

The parameter is the path of the file where the report should be generated.

```
OutputStream outStream = new OutputStream("Reports/pdfReport.pdf");
wizard.generatePDFReport(outStream);
```

This is the same example, only a stream is passed that points to a location in the hard drive, inside the folder Reports

On the given example we can see that the PDF report is the most suitable for reading, because this type of report can show both the data and the pictures properly.

CONTEXT: INFORMATIONAL
GROUP: dessert
RULE: Sweet dessert

CONTEXT: INFORMATIONAL
GROUP: after dinner
RULE: After dinner

Ime alkoholnog pica	Cena [euro]
Maryvale	87.50
Monterra	57.50

CONTEXT: INFORMATIONAL

GROUP: dessert
RULE: Sweet dessert
Ako slatke deserte volite, onda casa Port najvise prija