A new factory for Fractal/GCM

October 4, 2010

Contents

The aim of this document is to provide developers and users with sufficient documentation for the new Fractal/GCM factory for a seemles

1 Introduction

1.1 Context

In a few words, this work consists in the design and implementation of a new component factory for the GCM component-based grid architecture, which is a extension of the Fractal specification, and whose ProActive constitutes the only implementation available today.

1.1.1 Fractal

According to its authors, Fractal is a modular, extensible and programming language agnostic component model that can be used to design, implement, deploy and reconfigure systems and applications, from operating systems to middleware platforms and to graphical user interfaces.

`http://fractal.ow2.org/`

Fractal is mostly a Research system which gathers a large community of people working on various aspect of the component-oriented paradigm.

1.1.2 ProActive

According to its authors (OASIS Research team) ProActive is a GRID Java library for parallel, distributed, and concurrent computing, also featuring mobility and security in a uniform framework. ProActive runs on Local Area Network (LAN), on clusters of workstations, or on Internet Grids.

`http://proactive.inria.fr/`

1.1.3 GCM

GCM is...

1.1.4 ADL and the component factory

ADL stands for Architecture Description Language. Altought its semantic incorrectness, one usually refers to ADL as *architecture description*.

The ADL enables the description of a component system. In the particular case of Fractal, the component system is a tree, hence the adequation of XML (note that Fractal's ADL supposedly do not depend on XML).

The XML dialect introduced with Fractal as its ADL makes use of a variety of elements and attributes. A basic tutorial for ADL syntax can be found at the following web URL:

http://fractal.ow2.org/tutorials/adl/index.html

Basically, an ADL description looks like that:

```
1 <definition name="luc.demo.ExampleComponent">
2   <interface name="r" role="server" signature="luc.demo.ExampleComponent"/>
3   <component name="impl">
4     <interface name="s" role="server" signature="luc.demo.ExampleComponent"/>
5     <content class="luc.demo.BasicExampleComponent"/>
6   </component>
7   <binding client="this.r" server="impl.s" />
8 </definition>
```

Fractal assumes that the ADL description is not stored in a file but in a Java resource, whereas it should be stored in a standard file, at least for the sake of independance to Java. In spite of it, Fractal's way of referring to an ADL resource is to call it an *ADL file*.

The component factory of Fractal is a piece of code within Fractal whose the role is to build a component out of its XML description. GCM comes with its own factory which is an extension of the Fractal factory.

The implementation of the Fractal factory is cumbersome: it consists of about 150 Java classes and configuration files. organized in numerous packages. Besides, in order to implement is own extensions, GCM adds about 40 classes and description files.

1.2 Description of the problem

The Fractal factory (and its derivative) suffer from a number of problems, which lead to high frustration when it comes to deal with it.

- its design involves way more concepts than necessary, most often these concepts are imprecise;
- its design is made of numerous anti-intuitive constructs;
- its implementation code is made of many tricks;
- it comes with no documentation, be it at the level of the code or at the level of the conception.

The objective of the new factory is then to make a factory that is shorter, cleaner, easier to understand/modify, and faster than the original Fractal/GCM factory. It particular, in order to avoid confusion, we intend to use as little concepts as they are stricly necessary. Also, we plan to document it so as a new user involved in its development/extension will have a minimal amount of stress getting into the code.

1.3 Compliance with previous work

On the one hand, the new factory must be able to load ADL files as they are described by the Fractal and GCM specifications. It must also take into account extensions which are not part of official standard but which are of interest for the

Oasis Research team. Non-official extensions which are not of interest for Oasis should not be considered, unless the new factory is foreseen as a replacement of the default factory within Fractal. In such case, the authors of these extensions should be asked a contribution to porting their code.

On the other hand, the new factory may introduce constructs which constitute improvements over the GCM ADL in its current version. As such, ADL definitions tailored for the new factory may not be loadable by the default GCM factory.

Also the new factory may expose a different API that than the default GCM factory, considering that the new API suits better its very purpose.

2 Design of the new factory

The new factory is object-oriented (which does not mean that everything in it is object). Its design objective is to use an adequate tradeoff between *modularity* (which leads to large architectures) and *simplicity* (which often leads to a lack of modularity)

2.1 Input format

The input of the factory comes in the form of files on the disk.

This constitute a difference with the original factory which requests as input an ADL file as well as a *context* aimed at stored unspecified information.

2.1.1 Passing arguments to the factory

2.1.2 Deployment descriptor

2.2 Architecture

2.2.1 Files

The new factory is made of 8 Java source files, organized in 2 packages.

`xmlheader.txt` ..... the header of the XML file, which includes the DTD specification. This header will be automatically inserted at the beginning of the XML text right before parsing.

`NewFactory.java` .. This is where everything happens. If you want to extend behavior, you need to derive this class.

`description/AttributeDescription.java` ..... Attr description

2.2.2 Lexical analysis: from XML to DOM

XML parsing to DOM is done by the parser built-in into the Java Development Kit. Because DOM structures are not handy to manipulate, the parsing process goes a little further: it converts the DOM structure into a lightweight custom tree structure whose the basic signature is:

```
1 class XMLNode
2 {
3         String getName();
4         Map<String, String> getAttributes();
5         List<XMLNode> getChildrenNodes();
6 }
```

which is more convenient to use than the general purpose DOM data structure.

2.2.3 Semantic analysis: from DOM to semantic description

Each XML element type into the ADL description is represented by a `Description` class. Because every XML element declaration refer to a specific concept, each of these concept is represented as a specific sub-class of the `Description` class. Then at runtime, each XML element is represented as an instance of a sub-class of the class `Description`, as follows:

definition* and *component respectively describe the root component of the component tree; and a sub-component of a given component; they are both represented by the class `ComponentDescription`;

interface describes an interface of a given component, it is represented by the class `InterfaceDescription`;

binding describes an interface binding involving two interfaces of two parent/child components; it is represented by the class `BindingDescription`;

attributes describes attributes in a given components; it is represented by the class `AttributesDescription`;

content describes the content class for the implementation code of a given component. It is *not* represented by any description. Instead it is represented as a field into the class `ComponentDescription`.

The creation of the description is carried on by static methods in the sub-class of the class `Description`. Each of these static method is in charge of creating the description object for the host class.

2.2.4 Component creation: from semantic description to components

The creation of the component out of the semantic description of the component system is done using Fractal and GCM APIs. Because of this, components created by the new factory are of the very same nature than those created by the original factory. They can perfectly interoperate.

All the code is located in the class `NewFactory`.

3 Proposals for a better ADL

The ADL as it comes from the Fractal specifications presents a number of weaknesses and imprecisions. For example, it is not clear that the *definition* element is intrinsiquely different from the *component* one. Also, the usefulness of the *arguments* attribute within a *definition* element is to be discussed. This section addresses these imprecisions and proposes a number of enhancements of the ADL.

3.1 The DTD declaration is no longer required

It is not the responsibility of the XML file to declare which piece of software will be in charge of validating it. Instead the parser is aware of the structure constraints of the XML code.

3.2 Storing the ADL in files rather than in Java resources

CGM is implemented in Java. This said, the implementation language of a given software should remain a detail to the user. In Fractal, the input data (the ADL file) is expected to be a Java resource, making Fractal applications dependant of Java, even at the user-level. This somehow breaks the rule that Fractal is a specification intended to be implemented in any language.

Instead, the input data should be expected to be a file.

3.3 the *definition* XML element

A component is described by the use of the *component* element, except at the root-level where the component is to be referred as a *definition*. This choice probably comes from the fact, reinforced by the use of a DTD, that the *definition* element allows extra attributes that the *component* element does not. The *arguments* attribute might be of these.

3.4 the *arguments* attribute

Suppressing the *arguments* attribute into component description element.

3.5 the *content* XML element

In the description of a component, only one *content* element is allowed, and this element accepts only one attribute, the *class* attribute.

```
1 <component name="boh">
2         ...
3         <content class="java.util.ArrayList" />
4         ...
5 </component>
```

Instead, the content information for a component should be expressed as an attribute in the `component` description.

```
<component name="boh" content="java.util.ArrayList">
        ...
</component>
```

4 How to...

4.1 Perform a new verification

Verifications are all performed in the `check()` method of the `Description` class. Depending on the description you want to perform the verification (component, interface, etc), you will have to look into the corresponding description class (respectively `ComponentDescription`, `IntefaceDescription`, etc).

Adding a new verification consists in adding a line to the `check()` method. Typically, such line is in the form:

```
if (!condition)
        throw new ADLException("condition failed");
```

For convenience purpose, the use of the following construct is encouraged:

```
Assertions.ensure(condition, "condition failed");
```

4.2 Add a new attribute to an existing XML element

As described in Section **??**, each element type is represented by a `description` class. In turn, each attribute of the element is represented as a field into its corresponding `description` class. Adding a new attribute consists then in adding a new field in this class.

In order to make the new attribute need to be considered by the semantic analyser, you need to add a line of code into the corresponding method.

4.3 Add a new XML element

XML elements have a semantic corresponding description. To add a new XML element, you need to define such a description.

- Mandatory attributes have no default value. They must be assigned at construction time.
- Optional attributes have a default value. They are reassigned only if they are explicity given by the user.

4.3.1 Example: the interface element

Mandatory attributes must be provided at construction time:

```
1  String name  = n.getAttributes().get("name");
2  Role role = n.getAttributes().get("role").equals("client") ? Role.CLIENT : Role.SERVER;
3  Class<?> signature = Clazz.findClassOrFail(n.getAttributes().get("signature"));
4  InterfaceDescription id = new InterfaceDescription(name, role, signature);
```

Optional elements are set right after construction, if the user provided a value of them:

```
1  if (n.getAttributes().get("contigency") != null)
2  {
3          id.setContingency(n.getAttributes().get("contigency").equals("mandatory")
4                  ? Contingency.MANDATORY : Contingency.OPTIONAL);
5  }
```

4.4 Modify the way attributes values are processed

XML attribute values are parsed by the method:

`String replaceArgument(String v, Map<String, String> argumentValues)`

method in the `Attributes` class. Its default behavior is to replace the pattern `${name}` by the value of the *name* argument by its value found in the associative map `argumentValues`.

Override it to get the behavior you want.

5 Features

5.1 Already supported

The following features are already supported by the new factory:

- component interfaces (singleton);
- component attributes;
- interface bindings;
- sub-components;
- ADL inheritance;
- ADL arguments (plus extra features);

5.2 Under progress

The following lists the features that are currently under development.

- support for componentized membranes (Paul Naoumenko's topic);
- support for collection interfaces;

5.3 Will be suppored in the future

The following lists the features that are will be developed in the near future, ordered by importance.

1. support for collection components (Amine Rouini's topic);
2. support for shared components;
3. support for bindings to web-services;

6 Conclusion and future works