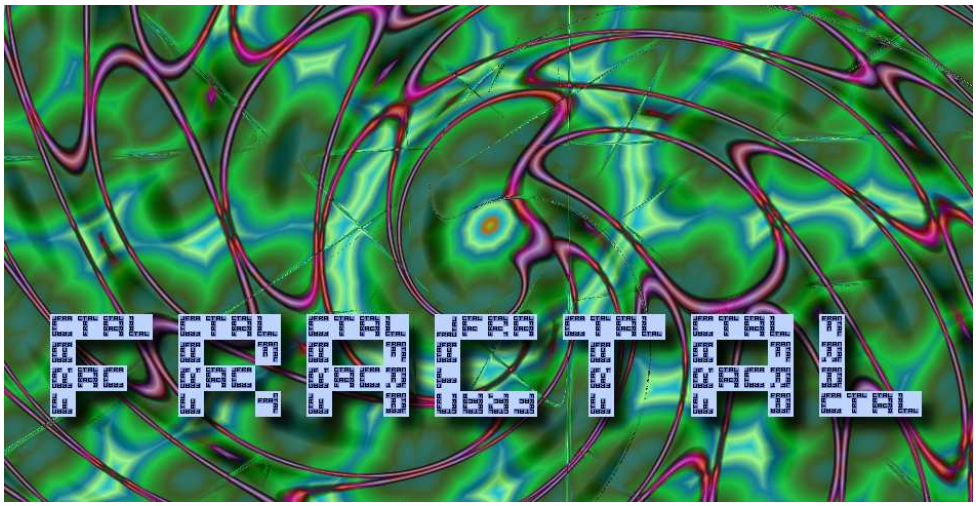
The ObjectWeb Consortium

Tutorial

Developing with Fractal



Author:
E. Bruneton (France Telecom R&D)

Released March 10, 2004
Status Final
Version 1.0.3

General Information

- Background of front-page image appears courtesy of Giuseppe Zito.
- Please send technical comments on this document to fractal@objectweb.org

Trademarks

All product names mentioned herein are trademarks of their respective owners.

Disclaimer of warranties

This document is provided "as is". France Télécom makes no representations or warranties, either express or implied, including but not limited to, warranties of merchantability, fitness for a particular purpose, or non-infringement that the contents of this document are suitable for any purpose or that any practice or implementation of such contents will not infringe any third party patents, copyrights, trade secrets or other rights.

Contents

1. Introduction

1.1. What is Fractal?

Fractal is a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms and to graphical user interfaces. Fractal is also a project with several sub projects, dealing with the definition of the model, its implementations, and the implementation of reusable components and tools on top of it.

The Fractal component model heavily uses the ***separation of concerns*** design principle. The idea of this principle is to separate into distinct pieces of code or runtime entities the various *concerns* or *aspects* of an application: implementing the service provided by the application, but also making the application configurable, secure, available, ... In particular, the Fractal component model uses three specific cases of the separation of concerns principle: namely *separation of interface and implementation*, *component oriented programming*, and *inversion of control*. The first pattern, also called the bridge pattern, corresponds to the separation of the design and implementation concerns. The second pattern corresponds to the separation of the implementation concern into several composable, smaller concerns, implemented in well separated entities called components. The last pattern corresponds to the separation of the functional and configuration concerns: instead of finding and configuring themselves the components and resources they need, Fractal components are configured and deployed by an external, separated entity.

The separation of concerns principle is also applied to the structure of the Fractal components. A Fractal component is indeed composed of two parts: a *content* that manages the functional concerns, and a *controller* that manages zero or more non functional concerns (introspection, configuration, security, transactions, ...). The content is made of other Fractal components, i.e. Fractal components can be *nested* at arbitrary levels (Fractal components can also be *shared*, i.e. be nested in several components at the same time). The introspection and configuration interfaces that can be provided by the controllers allow components to be deployed and reconfigured dynamically. These control interfaces can be used either programmatically, or through tools based on them, such as deployment or supervision tools.

More information about Fractal, including the complete specification of the component model, and several tutorials, can be found at `http://fractal.objectweb.org`.

1.2. Content of this document

This tutorial is an introduction to the Fractal component model. It explains informally how to design (section 2), implement (section 3), deploy (section 4), and dynamically reconfigure (section 5) component based applications with Fractal (and with some associated tools), in Java, by using a concrete example, namely an extremely minimal web server.

1.3. Target audience

This document is intended for those that do not know Fractal, and want to get an overview of this component model, of its motivations and benefits. If you are in this case, you should

read this document first, before reading any other document about Fractal and, in particular, before reading the Fractal component model specification.

Summary

The main characteristics of the Fractal model are recursion, reflexion and sharing. The Fractal project is made of four sub projects: model, implementations, components and tools.

2. Design

Before programming a component based software system with Fractal, one must first *design* it with components and, in particular, identify the components to be implemented. Note that this *component oriented design* task is quite independent from the subsequent *component oriented programming* task: at programming time, it is possible to merge several or even all the design time components into a single, monolithic piece of code, if desired (but then, of course, one looses the advantages of component oriented *programming*: modularity, adaptability, ...).

This section explains how to design component based applications, by using a concrete example. The next sections reuse this example to illustrate how to implement and deploy Fractal component based applications. This example application is Comanche, an extremely minimal HTTP server. A classical, non component oriented implementation of this application, in Java, is shown below:

```
public class Server implements Runnable {
  private Socket s;
  public Server (Socket s) { this.s = s; }
  public static void main (String[] args) throws IOException {
    ServerSocket s = new ServerSocket(8080);
    while (true) { new Thread(new Server(s.accept())).start(); }
  }
  public void run () {
    try {
      InputStreamReader in = new InputStreamReader(s.getInputStream());
      PrintStream out = new PrintStream(s.getOutputStream());
      String rq = new LineNumberReader(in).readLine();
      System.out.println(rq);
      if (rq.startsWith("GET ")) {
        File f = new File(rq.substring(5, rq.indexOf(' ', 4)));
        if (f.exists() && !f.isDirectory()) {
          InputStream is = new FileInputStream(f);
          byte[] data = new byte[is.available()];
          is.read(data);
          is.close();
          out.print("HTTP/1.0 200 OK\n\n");
          out.write(data);
        } else {
          out.print("HTTP/1.0 404 Not Found\n\n");
          out.print("<html>Document not found.</html>");
        }
      }
      out.close();
      s.close();
    } catch (IOException _) { }
  }
}
```

As can be seen from the source code, this server accepts connections on a server socket and, for each connection, starts a new thread to handle it (in the main method). Each connection is handled in two steps, in the run method: the request is analyzed and logged to the standard

output, and then the requested file is sent back to the client (or an error is returned if the file is not found).

2.1. Finding the components

In a component based application, some components are *dynamic*, i.e they can be created and destroyed dynamically, possibly quite frequently, while other components are *static*, i.e. their life time is equal to the life time of the application itself. The dynamic components generally correspond to *datas*, while the static ones generally correspond to *services*.

In order to identify components in an application, it is easier to begin by identifying its static components. In other words, one should begin by identifying the services that are used in the application. In the case of Comanche, we can immediately identify two main services, namely a request receiver service and a request processor service (corresponding to the content of the two methods of the Server class). But it is also possible to identify other, lower level services. For example, we can see that the request receiver service uses a thread factory service, to create a new thread for each request. This thread factory service can be generalized into a scheduler service that can be implemented in several ways: sequential, multi thread, multi thread with a thread pool, and so on. Likewise, we can see that the request processor uses a request analyzer service, and a logger service, before effectively responding to a request. This response is itself constructed by using a file server service, or an error manager service. This can be generalized into a request dispatcher service that dispatches requests to several request handlers sequentially, until one handler can handle the request (we can then imagine file handlers, servlet handlers, and so on).

After the services have been specified, one can look for the main data structures, in order to identify the dynamic components. But the identification of the dynamic components is not mandatory, and is generally not done, because dynamic components themselves are rarely used (this means that, at programming time, data structures are generally not implemented as components, but as ordinary objects - if the programming language is object oriented). Indeed components do not have many benefits in the case of highly dynamic, short lived structures (introspection and dynamic reconfiguration, for instance, are not very useful in this case). In the case of Comanche we can consider sockets, HTTP requests, files, streams, and even threads as such data structures. But we will not map them to dynamic components.

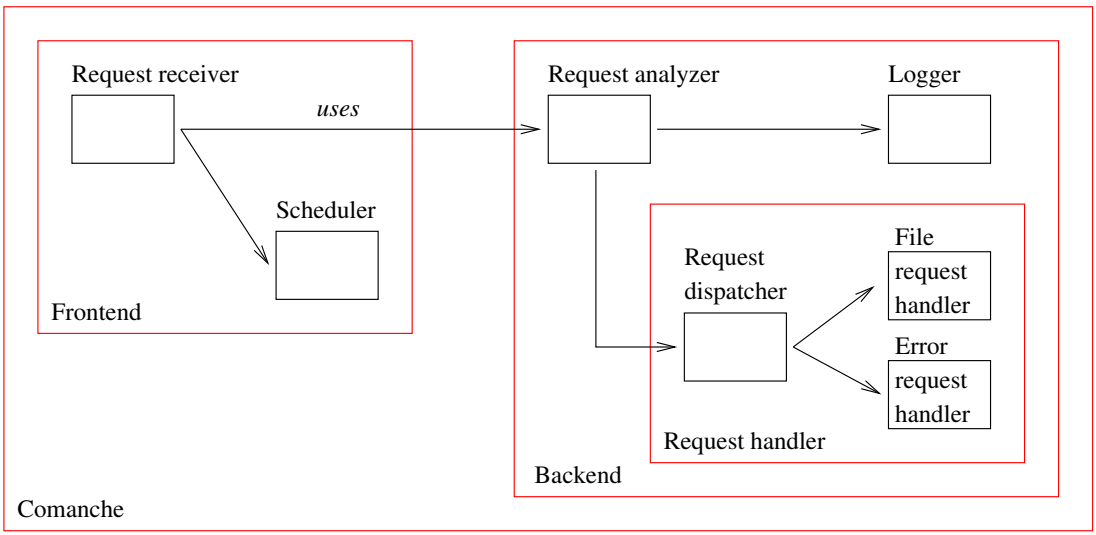
After the services have been specified, one must assign them to components. Each component can provide one or more services but, unless two services are very strongly coupled, it is better to use one component per service. In the case of Comanche, we will use one component per service. We therefore have the seven following components: request receiver, request analyzer, request dispatcher, file request handler, error request handler, scheduler and logger.

2.2. Defining the component architecture

After the components have been identified, it is easy to find the dependencies between them, and to organize them into composite components. Indeed the service dependencies are generally identified at the same time as the services themselves. If it is not the case, dependencies can also be found by looking at some use cases or scenarios. Likewise, services are generally

identified from high level to lower level services (or vice versa). It is then easy to find the dependencies and the abstraction level of each component, since the components correspond to the previous services.

This is particularly clear in the case of Comanche: indeed, by re reading the previous section, one can see that the service dependencies have already been identified. For example, the request receiver service uses the scheduler service and the request analyzer, the request analyzer uses the request dispatcher, which itself uses the file and error request handlers. One can also see that the abstraction levels have already been found: the request receiver is "made of" the request receiver itself, plus the scheduler; the request processor is made of the request analyzer, the logger, and the request handler; the request handler is itself made of the request dispatcher, and of the file and error request handlers. All this can be summarized in the following component architecture:



2.3. Defining the component contracts

After the services have been found and organized into components, and after the component hierarchy and the component dependencies have been found, only one thing remains to be done to finish the design phase, namely to define precisely the contracts between the components, at the syntaxic and semantic level (if possible with a formal language - such as pre and post conditions, temporal logic formulas, and so on). Classical object oriented design tools, such as scenarios and use cases, can be used here.

The component contracts must be designed with care, so as to be the most stable as possible (changing a contract requires to change several components, and is therefore more difficult than changing a component). In particular, these contracts must deal only with the services provided by the components: nothing related to the implementation or configuration of the components themselves should appear in these contracts. For example, a `setLogger` operation has nothing to do in a component contract definition: this operation is only needed to set a reference between a component and a logger component. In other words, contracts must be defined with ***separation of concerns*** in mind (see section 1): contracts must deal only

with functional concerns; configuration concerns will be dealt with separately, as well as other concerns such as security, life cycle, transactions...

In the case of Comanche, we will use minimalistic contracts, defined directly in Java:

- The logger service will provide a single log method, with a single String parameter;
- The scheduler component will provide a single schedule method, with a single Runnable parameter (the role of this method is to execute the given Runnable at some time after the method has been called, possibly in a different thread than the caller thread);
- The request analyzer, the request dispatcher and the file and error request handlers will all provide a single handleRequest method, with a single Request parameter that will contain information about a request: its socket, the input and output streams of this socket, and the requested URL. This single handleRequest method will be implemented in several ways, in the different components:
  - the request analyzer will construct the input and output streams, and will read the input stream to get the requested URL;
  - the request dispatcher will forward each request to its associated request handlers, sequentially, until one request handler successfully handles the request;
  - the file request handler will try to find and to send back to the client the file whose name corresponds to the requested URL, if it exists;
  - the error request handler will send back to the client an error message, and will always succeed.

Summary

The first step to design a component based application is to define its components. This is done by finding the *services* used in this application. The second step is to find the dependencies and hierarchical levels of these components. The last step is to define precisely the contracts between the components.

3. Implementation

This section explains how to program Fractal component based applications, by using the Comanche example. It also introduces and motivates the concepts and APIs of Fractal that are used.

3.1. Choosing the components' granularity

As explained in section 2, component oriented design is quite independent from component oriented implementation. In particular, at programming time, it is possible to merge several or even all the design time components into a single, monolithic piece of code, if desired. For example, in the case of Comanche, one may choose to implement all the design time components into a single class, as shown in section 2. One may also choose to implement each component in its own class, or to implement some components in their own class, and to merge some other components (such as the request dispatcher and its associated request handlers) into a single class.

Using one runtime component per design time component gives maximum flexibility and extensibility, but can be less efficient that merging several design time components into a single runtime component. When in doubt, *the first solution should be preferred*: optimizations can be done later on, if needed. In the case of Comanche, we will therefore use one runtime component per design time component.

3.2. Implementing the component interfaces

Before implementing the component themselves, the first step is to implement their *interfaces*. Indeed the Fractal component model requires a strict ***separation between interfaces and implementation*** for all components (see section 1). This design pattern is indeed useful to easily replace one component implementation with another, without worrying about class inheritance problems. It also offers the possibility to add interposition objects between a client and a component implementation, in order to transparently manage some non functional concerns of the component (as in the Enterprise Java Beans model - see section 5). The only drawback of this design pattern is that it is a little less efficient than a solution without interfaces. This is why it may sometimes be needed to merge several design time components into a single Fractal component.

The component interfaces can be implemented easily, since most, if not all, of the work has been done during the definition of the component contracts (see section 2.3). In the case of Comanche, three interfaces must be implemented. They are given below (see Appendix A for the full source code of Comanche):

```
public interface Logger { void log (String msg); }
public interface Scheduler { void schedule (Runnable task); }
public interface RequestHandler { void handleRequest (Request r) throws IOException; }
```

Note that requests are represented as instances of the Request *class*. This choice was made here only to show that classes can be used in interface method parameters, *although this is*

not recommended. It is indeed better to use interfaces, even for data structures that are not represented as components: these data structures can then be implemented in various ways (including as components), without needing to change the interfaces that refer to them. Note also that it would have been better to introduce a request factory component (this was not done for simplification purposes).

3.3. Implementing the components

Now that the component interfaces have been implemented, we can implement the components themselves. The components that do not have any dependencies to other components can be programmed like ordinary Java classes. For example, the logger and scheduler components can be implemented as follows:

```
public class BasicLogger implements Logger {
  public void log (String msg) { System.out.println(msg); }
}
public class SequentialScheduler implements Scheduler {
  public synchronized void schedule (Runnable task) { task.run(); }
}
public class MultiThreadScheduler implements Scheduler {
  public void schedule (Runnable task) { new Thread(task).start(); }
}
```

In component oriented programming, and in Fractal in particular, the components that have dependencies to other components must be programmed in a specific way. Consider for example the request receiver component. A possible way to implement it, in classical object oriented programming, is as follows:

```
public class RequestReceiver {
  private Scheduler s = new MultiThreadScheduler();
  private RequestHandler rh = new RequestAnalyzer();
  // rest of the code not shown
}
```

This approach is clearly not usable in component oriented programming, since modularity is lost: with this implementation, it is impossible to change the implementation of the scheduler or of the request analyzer component used by the request receiver component, without modifying and recompiling the source code of this component. A better approach is to use ***inversion of control*** (see section 1):

```
public class RequestReceiver {
  private Scheduler s;
  private RequestHandler rh;
  public RequestReceiver (Scheduler s, RequestHandler rh) { this.s = s; this.rh = rh; }
  // rest of the code not shown
}
```

This approach solves the modularity problem, but only at deployment time: it is still impossible to change the scheduler or the request analyzer used by the request receiver *at runtime*.

Since one of the goals of Fractal is to support dynamic reconfigurations, the inversion of control principle must be used in a better way. In fact, in Fractal, a component with dependencies (or *bindings*) to other components must implement the BindingController interface, defined in the Fractal specification. This interface defines four generic methods listFc, lookupFc, bindFc and unbindFc to manage component bindings. In Fractal, the request receiver component must therefore be implemented as follows:

```
public class RequestReceiver implements Runnable, BindingController {
  private Scheduler s;
  private RequestHandler rh;
  // configuration concern
  public String[] listFc () { return new String[] { "s", "rh" }; }
  public Object lookupFc (String itfName) {
    if (itfName.equals("s")) { return s; }
    else if (itfName.equals("rh")) { return rh; }
    else return null;
  }
  public void bindFc (String itfName, Object itfValue) {
    if (itfName.equals("s")) { s = (Scheduler)itfValue; }
    else if (itfName.equals("rh")) { rh = (RequestHandler)itfValue; }
  }
  public void unbindFc (String itfName) {
    if (itfName.equals("s")) { s = null; }
    else if (itfName.equals("rh")) { rh = null; }
  }
  // functional concern
  public void run () { /* see Appendix A */ }
}
```

As can be seen, the listFc method returns the names of the dependencies of the component, and the lookupFc, bindFc and unbindFc methods are used to read, set and unset the corresponding bindings (the s and rh strings do not have to be equal to the names of the corresponding fields). The distinction between the *controller* and *content* part of Fractal components (see section 1) can also be clearly seen here: the controller part corresponds to the BindingController interface, and the content part to the Runnable interface (in the logger and scheduler components, the controller part was empty).

For components that can use a *collection* of similar bindings to other components, such as the request dispatcher component, each binding must have a name of the form prefix*postfix* where prefix is common to all the bindings of the collection, and where *postfix* is arbitrary, but distinct for each binding. The request dispatcher component must therefore be implemented as follows (here the prefix is h; the map used is sorted, in order to be able to specify a dispatch order through the binding names, but this is not mandatory):

```
public class RequestDispatcher implements RequestHandler, BindingController {
  private Map handlers = new TreeMap();
  // configuration concern
  public String[] listFc () {
    return (String[])handlers.keySet().toArray(new String[handlers.size()]);
  }
  public Object lookupFc (String itfName) {
```

```
    if (itfName.startsWith("h")) { return handlers.get(itfName); }
    else return null;
  }
  public void bindFc (String itfName, Object itfValue) {
    if (itfName.startsWith("h")) { handlers.put(itfName, itfValue); }
  }
  public void unbindFc (String itfName) {
    if (itfName.startsWith("h")) { handlers.remove(itfName); }
  }
  // functional concern
  public void handleRequest (Request r) throws IOException { /* see Appendix A */ }
}
```

Summary

Components must be implemented with an appropriate granularity, resulting from a compromise between adaptability and performance. Their interfaces must be separated from their implementation (this rule must also be applied for data structures). The implementation must not contain explicit dependencies to other components to allow static and dynamic reconfigurations.

4. Configuration

This section presents several methods to assemble and deploy Fractal components, always by using the Comanche example. It also introduces and motivates the Fractal tools that are used.

4.1. Programmatic configuration

The most direct, but also the lowest level method to assemble and deploy Fractal components is to write a specific program to do that. The role of this program is to create the components, to bind them to each other, and finally to start the application. In the case of Comanche, this deployment program is the following:

```
public class Server {
  public static void main (String[] args) {
    RequestReceiver rr = new RequestReceiver();
    RequestAnalyzer ra = new RequestAnalyzer();
    RequestDispatcher rd = new RequestDispatcher();
    FileRequestHandler frh = new FileRequestHandler();
    ErrorRequestHandler erh = new ErrorRequestHandler();
    Scheduler s = new MultiThreadScheduler();
    Logger l = new BasicLogger();
    rr.bindFc("rh", ra);
    rr.bindFc("s", s);
    ra.bindFc("rh", rd);
    ra.bindFc("l", l);
    rd.bindFc("h0", frh);
    rd.bindFc("h1", erh);
    rr.run();
  }
}
```

4.2. ADL based configuration

The previous configuration and deployment method has several drawbacks: it is error prone (it is easy to forget a binding or to create a wrong binding), the component architecture is not directly visible (the component's hierarchy description, in particular, is completely lost) and, most importantly, *this method mixes two separate concerns, namely architecture description and deployment* (it is impossible to deploy a given component architecture in several ways, without rewriting the configuration/deployment program).

In order to solve these problems, a solution is to use an Architecture Description Language (ADL). As its name implies, an ADL definition describes a component architecture, and only that, i.e. its does not describe the instantiation method. This solves the most important drawback of the previous configuration method. An ADL is also generally strongly typed, which allows the ADL parser to perform verifications about the declared component architecture. Using an ADL is therefore less error prone than using the programmatic approach.

Fractal ADL is a possible, XML based ADL that can be used to describe Fractal component configurations. A simplified ADL, not based on XML, can also be used, and other ADLs

can be created if needed (indeed these ADLs are not part of the Fractal component model itself: they are just tools based on this model). Fractal ADL is strongly typed. The first step to define a component architecture is therefore to define the types of the components. Each component type must specify what components of this type provide to, and require from other components. For example, the type of the file and error handler components (but also of the request handler and backend components - see section 2.2), in Comanche, can be defined as follows (these components provide a RequestHandler interface, and do not have dependencies):

```
<definition name="comanche.HandlerType">
  <interface name="rh" signature="comanche.RequestHandler" role="server"/>
</definition>
```

Components with dependencies are declared in a similar way. For example, the type of the request dispatcher component, in Comanche, can be defined as follows:

```
<definition name="comanche.DispatcherType" extends="comanche.HandlerType">
  <interface name="h"
      signature="comanche.RequestHandler" role="client" cardinality="collection"/>
</definition>
```

Note that this type is declared to *extend* the previous handler type: this means that the provided and required interface types declared in the handler type are inherited by the dispatcher type. Note also the optional *cardinality* attribute in the interface type definition: it means that components of this type can have a variable number of bindings (see end of section 3.3).

After the component types have been defined, the components themselves can be defined. Here Fractal ADL distinguishes between components that do not expose their content, called primitive components, and components that do expose it, called composite components. A primitive component is defined by specifying its component type and the Java class that implements it. For example, the file handler component can be defined as follows:

```
<definition name="comanche.FileHandler" extends="comanche.HandlerType">
  <content class="comanche.FileRequestHandler"/>
</definition>
```

Composite components are defined by specifying their sub components, and the bindings between these sub components. For example, the Comanche composite component, which represents the whole application, and which contains the frontend and backend components, can be defined as follows (see Appendix B for the definitions of the sub components):

```
<definition name="comanche.Comanche" implements="comanche.RunnableType">
  <component name="fe" definition="comanche.Frontend"/>
  <component name="be" definition="comanche.Backend"/>
  <binding client="this.r" server="fe.r"/>
  <binding client="fe.rh" server="be.rh"/>
</definition>
```

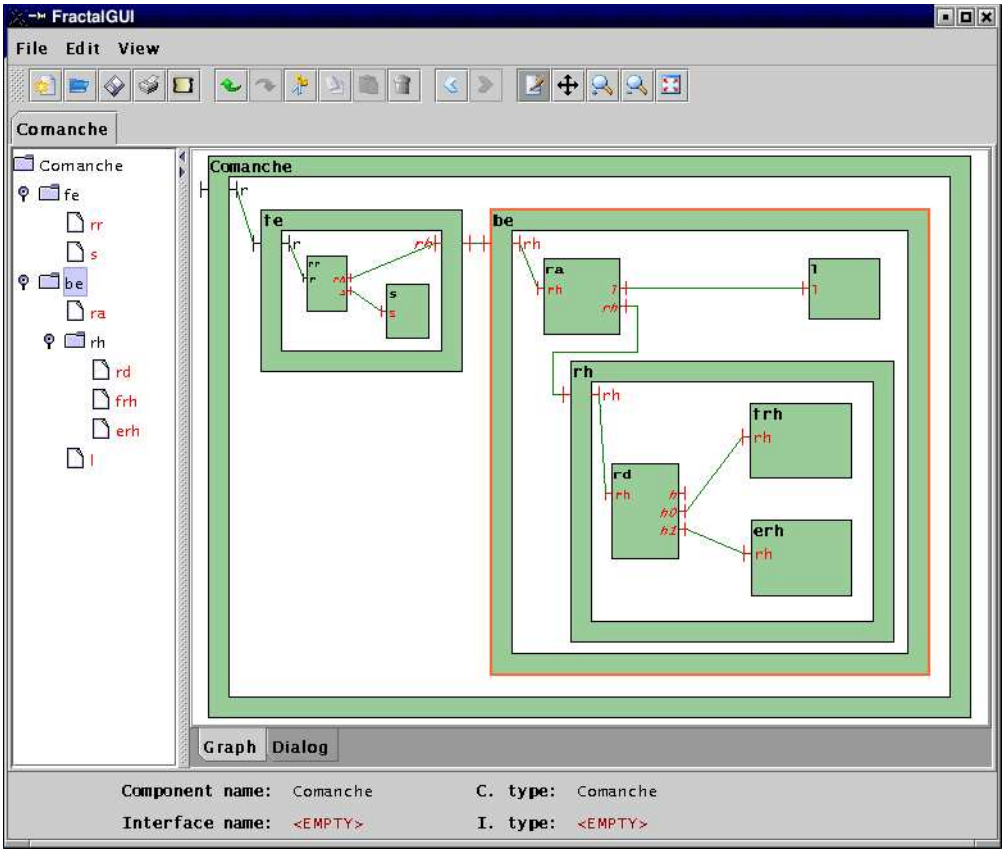
This definition says that the Comanche component provides a Runnable interface (to start the application), that it contains two sub components named fe and be, that the Runnable interface provided by Comanche (this.r) is provided by the Runnable interface of its frontend

sub component (fe.r), and that the request handler required by the frontend sub component (fe.rh) is provided by the backend component (be.rh).

Once the application's architecture has been defined, it can either be compiled, which gives a Java class, in source code, similar to the Server class shown in section 4.1, or it can be directly interpreted. In both case, the Fractal ADL parser performs preliminary verifications to check the architecture and, in particular, to check that there is no missing or invalid binding.

4.3. GUI based configuration

Using an ADL is better than using a programmatic approach, but does not help a lot to visualize the architecture of an application. To solve this problem, the best solution is to use a graphical tool such as Fractal GUI, which is shown below:



This tool can be used to graphically and interactively visualize, define or modify component architectures. The edited architectures can then be saved in various ADLs (currently only the Fractal ADL is supported).

Summary

Components can be configured and deployed in three different ways. The programmatic approach is the most direct but mixes different concerns. The ADL based approach correctly separates these concerns. The graphical and interactive approach also solves this problem and, in addition, provides a better architectural representation.

5. Reconfiguration

This section explains how Fractal component based applications can be dynamically reconfigured. It also introduces and motivates the concepts and APIs of Fractal that are used.

5.1. Life cycle management

In order to reconfigure an application, the easiest solution is to stop it, to modify its architecture description, and then to restart it. But this solution is not always possible, in particular if the service provided by the application must not be interrupted. In these cases, the application must be reconfigured at runtime, i.e. while it is in execution. But this solution has its own drawbacks: in general, if the reconfiguration process is not carefully synchronized with the normal application activities, the application's state may become inconsistent, or the application may simply crash. A basic practice to realize this synchronization is to temporarily *suspend* the normal application activities in the (hopefully small) part of the application that needs to be reconfigured. This is what is called *life cycle* management in Fractal.

As an example, let us suppose we want to replace, in a Comanche server, the file request handler component with a new implementation that can manage both files and directories. We will consider here that the HTTP server must be always available, so that the previous change must be done dynamically, while the server is running. If rd is a reference to the request dispatcher component, this reconfiguration can be done with the following instructions:

```
RequestHandler rh = new FileAndDirectoryRequestHandler();
rd.unbindFc("h0");
rd.bindFc("h0", rh);
```

But this reconfiguration method may produce undefined results, or even errors. Indeed the RequestDispatcher component uses a TreeMap to manage its bindings (see end of section 3.3), but this class does not support concurrent accesses. Modifying this map in a reconfiguration thread, while it is also accessed, via an iterator, in the applicative threads that execute the handleRequest method, may therefore cause errors or inconsistencies.

In order to solve this problem, a generic solution is to 1) temporarily suspend the application activities in the request dispatcher component, 2) execute the above reconfiguration instructions, and 3) resume the application activities. In order to do that the request dispatcher component must implement the LifeCycleController interface, defined in the Fractal specification. This interface defines two methods startFc and stopFc, to suspend and resume the application activities in a component, and an additional getFcState method. A possible way to implement this interface is to use a counter of the number of threads that are currently executing a functional method, and to wait for this counter to become null to stop the component:

```
public class RequestDispatcher implements RequestHandler, BindingController, LifeCycleController {
  private boolean started;
  private int counter;
  public String getFcState () {
    return started ? STARTED : STOPPED;
  }
```

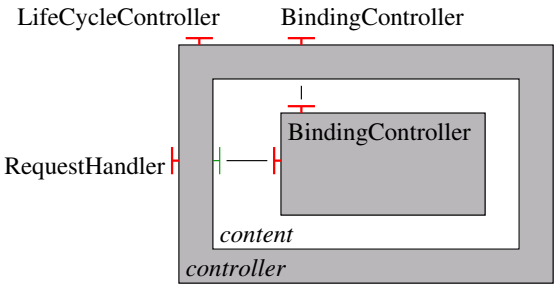
```
  public synchronized void startFc () {
    started = true;
    notifyAll();
  }
  public synchronized void stopFc () {
    while (counter > 0) { try { wait(); } catch (InterruptedException _) {} }
    started = false;
  }
  public void handleRequest (Request r) throws IOException {
    synchronized (this) {
      while (counter == 0 && !started) { try { wait(); } catch (InterruptedException _) {} }
      ++counter;
    }
    try {
      // original code
    } finally {
      synchronized (this) {
        --counter;
        if (counter == 0) { notifyAll(); }
      }
    }
  }
  // rest of the class unchanged
}
```

A better solution is to completely separate this life cycle management concern from the functional concern, in order to be able to instantiate the request dispatcher component with or without life cycle management (this management adds an overhead to the functional methods, which we do not want to pay if dynamic reconfigurations are not needed). This can be done easily, thanks to the separation between interface and implementation, by using, between the request analyzer and the (unmodified) request dispatcher component, an interceptor of the following form:

```
public class Interceptor implements RequestHandler, LifeCycleController {
  public RequestHandler delegate;
  private boolean started;
  private int counter;
  public String getFcState () { /* as above */ }
  public synchronized void startFc () { /* as above */ }
  public synchronized void stopFc () { /* as above */ }
  public void handleRequest (Request r) throws IOException {
    synchronized (this) { /* as above */ }
    try {
      delegate.handleRequest(r);
    } finally {
      synchronized (this) { /* as above */ }
    }
  }
}
```

This interceptor can also implement the BindingController interface by delegation. It can there-

fore becomes equivalent to the modified request dispatcher component described above, and can in fact be seen as a request dispatcher component with life cycle management, *encapsulating* a request dispatcher component without this control interface. This can be represented as follows:



We can see here the advantages of separating the architecture description from the instantiation method. Indeed the Comanche architecture, described only once, can now be instantiated either as described in section 4, or with *automatically generated* encapsulating components, i.e. "interceptors" like the above one, around some or all the primitive components.

We can also see here that the distinction between the controller and content parts is mainly a design distinction (controllers and contents are defined as *abstract* entities in the Fractal specification): in practice, these two parts are not necessarily implemented by distinct objects (the controller and content parts of the nested component are in a single object, but the controller and content part of the encapsulating component are distinct objects - but this is not mandatory: the encapsulating component can also be automatically generated as a *sub class* of the nested component's class!).

5.2. Introspection

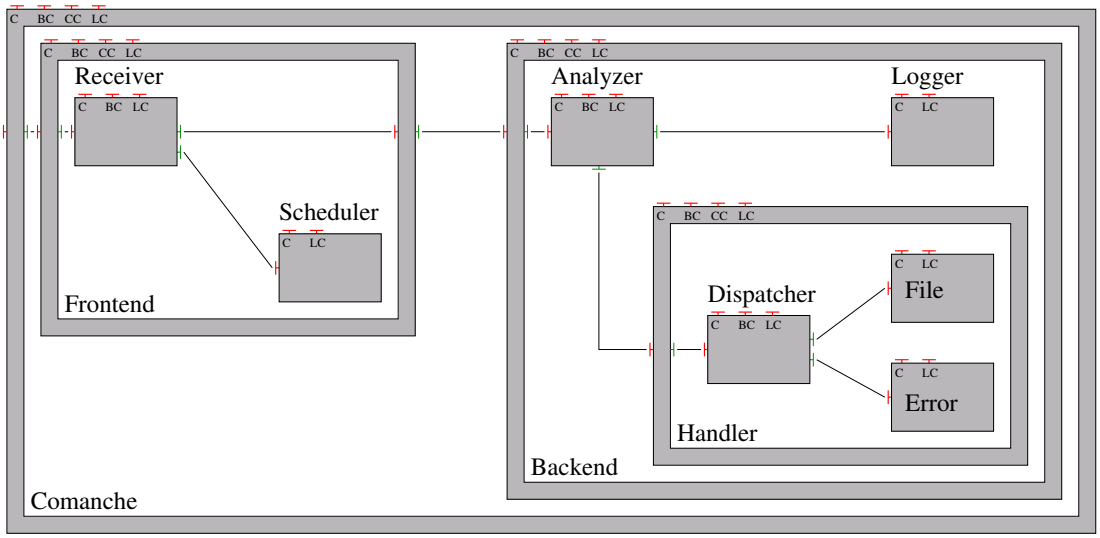
In order to dynamically reconfigure a component based application, one must first get the reference of the components to be reconfigured. For example, in the example of the previous section, the reference rd of the request dispatcher component was needed. One way of doing this is to start from the reference of a "well known" component, and to navigate in the application with the BindingController.lookupFc method. For example, the rd reference can be obtained from the reference rr of the request receiver component with the following expression: ((BindingController)rr.lookupFc("rh")).lookupFc("rh"). But this method only works for small applications. For more complicated applications, with hundreds of components and many hierarchical levels of composite components, and for applications whose architecture evolves at runtime, it is not very realistic. In such cases, a more realistic solution is to create and to maintain consistent, at runtime, a full description of the component architecture, including the composite components, so that the component architecture can be fully *introspected* dynamically.

This full introspection can be done, in the Fractal component model, by instantiating, in addition to the primitive components (and optional encapsulating components to manage some non functional concerns), the composite components described in the component architecture. In addition, it must also be ensured that all these components provide the necessary interfaces

to provide information about themselves. BindingController is one of these interfaces (with its listFc and lookupFc methods), but it is not sufficient. One must also be able to find the sub components of a composite component and, before that, one must also be able to find the interfaces provided by a component. The ContentController and Component interfaces, specified in the Fractal component model, are defined precisely for this goal. In Java, these interfaces are defined as follows (see the Fractal specification for more details):

```
public interface Component {
  Object[] getFcInterfaces ();
  Object getFcInterface (String itfName);
  Type getFcType ();
}
public interface ContentController {
  Object[] getFcInternalInterfaces ();
  Object getFcInterfaceInterface(String itfName);
  Component[] getFcSubComponents ();
  void addFcSubComponent (Component c);
  void removeFcSubComponent(Component c);
}
```

When the Comanche application is instantiated with full introspection capabilities, the result is the following architecture (C, BC, CC and LC stands for Component, BindingController, ContentController and LifeCycleController respectively; note also that the components without life cycle management encapsulated in the primitive components are not shown):



Summary

Fractal provides some basic tools to allow dynamic reconfiguration, namely reflexion and life cycle management. But dynamic reconfiguration is difficult and not yet completely solved in Fractal (in particular, state management is not yet specified).

6. Conclusion

The Fractal component model uses well known design patterns, and organizes them into a uniform, language independent model, that can be applied to operating systems, middleware platforms or graphical user interfaces.

The Fractal component model brings several benefits: it enforces the definition of a good, modular design; it enforces the separation of interface and implementation, which ensures a minimum level of flexibility; it enforces the separation between the functional, configuration and deployment concerns, which allows the application's architecture to be described separately from the code, for example by using an Architecture Description Language, and allows applications to be instantiated in various ways (from fully optimized but unreconfigurable configurations to less efficient but fully dynamically reconfigurable configurations). All these features should reduce the development time, and should also increase the reusability of components and component architectures, i.e. they should increase *productivity*.

The Fractal component model and its associated tools, although not yet as stable as other component models such the Enterprise Java Beans model, are ready to be used, and have already been used successfully in several applications such as THINK, a library of Fractal components to build operating system kernels, Speedo, an implementation of the Java Data Object (JDO) specification, or Fractal GUI, the graphical tool to edit Fractal component configurations.

More information about Fractal, including the complete specification of the component model, and several tutorials, can be found at `http://fractal.objectweb.org`.

A. Comanche source code

Here is the full source code of Comanche:

```
/* ================ Component interfaces ================ */

public interface RequestHandler {
  void handleRequest (Request r) throws IOException;
}

public interface Scheduler {
  void schedule (Runnable task);
}

public interface Logger {
  void log (String msg);
}

public class Request {
  public Socket s;
  public Reader in;
  public PrintStream out;
  public String url;
  public Request (Socket s) { this.s = s; }
}

/* ================ Component implementations ================ */

public class BasicLogger implements Logger {
  public void log (String msg) { System.out.println(msg); }
}

public class SequentialScheduler implements Scheduler {
  public synchronized void schedule (Runnable task) { task.run(); }
}

public class MultiThreadScheduler implements Scheduler {
  public void schedule (Runnable task) { new Thread(task).start(); }
}

public class FileRequestHandler implements RequestHandler {
  public void handleRequest (Request r) throws IOException {
    File f = new File(r.url);
    if (f.exists() && !f.isDirectory()) {
      InputStream is = new FileInputStream(f);
      byte[] data = new byte[is.available()];
      is.read(data);
      is.close();
      r.out.print("HTTP/1.0 200 OK\n\n");
      r.out.write(data);
    } else { throw new IOException("File not found"); }
  }
```

```
}

public class ErrorRequestHandler implements RequestHandler {
  public void handleRequest (Request r) throws IOException {
    r.out.print("HTTP/1.0 404 Not Found\n\n");
    r.out.print("<html>Document not found.</html>");
  }
}

public class RequestDispatcher implements RequestHandler, BindingController {
  private Map handlers = new TreeMap();
  // configuration concern
  public String[] listFc () {
    return (String[])handlers.keySet().toArray(new String[handlers.size()]);
  }
  public Object lookupFc (String itfName) {
    if (itfName.startsWith("h")) { return handlers.get(itfName); }
    else return null;
  }
  public void bindFc (String itfName, Object itfValue) {
    if (itfName.startsWith("h")) { handlers.put(itfName, itfValue); }
  }
  public void unbindFc (String itfName) {
    if (itfName.startsWith("h")) { handlers.remove(itfName); }
  }
  // functional concern
  public void handleRequest (Request r) throws IOException {
    Iterator i = handlers.values().iterator();
    while (i.hasNext()) {
      try {
        ((RequestHandler)i.next()).handleRequest(r);
        return;
      } catch (IOException _) { }
    }
  }
}

public class RequestAnalyzer implements RequestHandler, BindingController {
  private Logger l;
  private RequestHandler rh;
  // configuration concern
  public String[] listFc () { return new String[] { "l", "rh" }; }
  public Object lookupFc (String itfName) {
    if (itfName.equals("l")) { return l; }
    else if (itfName.equals("rh")) { return rh; }
    else return null;
  }
  public void bindFc (String itfName, Object itfValue) {
    if (itfName.equals("l")) { l = (Logger)itfValue; }
    else if (itfName.equals("rh")) { rh = (RequestHandler)itfValue; }
  }
  public void unbindFc (String itfName) {
    if (itfName.equals("l")) { l = null; }
```

```
    else if (itfName.equals("rh")) { rh = null; }
  }
  // functional concern
  public void handleRequest (Request r) throws IOException {
    r.in = new InputStreamReader(r.s.getInputStream());
    r.out = new PrintStream(r.s.getOutputStream());
    String rq = new LineNumberReader(r.in).readLine();
    l.log(rq);
    if (rq.startsWith("GET ")) {
      r.url = rq.substring(5, rq.indexOf(' ', 4));
      rh.handleRequest(r);
    }
    r.out.close();
    r.s.close();
  }
}

public class RequestReceiver implements Runnable, BindingController {
  private Scheduler s;
  private RequestHandler rh;
  // configuration concern
  public String[] listFc () { return new String[] { "s", "rh" }; }
  public Object lookupFc (String itfName) {
    if (itfName.equals("s")) { return s; }
    else if (itfName.equals("rh")) { return rh; }
    else return null;
  }
  public void bindFc (String itfName, Object itfValue) {
    if (itfName.equals("s")) { s = (Scheduler)itfValue; }
    else if (itfName.equals("rh")) { rh = (RequestHandler)itfValue; }
  }
  public void unbindFc (String itfName) {
    if (itfName.equals("s")) { s = null; }
    else if (itfName.equals("rh")) { rh = null; }
  }
  // functional concern
  public void run () {
    try {
      ServerSocket ss = new ServerSocket(8080);
      while (true) {
        final Socket socket = ss.accept();
        s.schedule(new Runnable () {
          public void run () {
            try { rh.handleRequest(new Request(socket)); } catch (IOException _) { }
          }
        });
      }
    } catch (IOException e) { e.printStackTrace(); }
  }
}
```

B. Comanche architecture definition

Here are the Fractal ADL definitions describing the Comanche architecture:

```
<!– ============== Component types ============== –>

<definition name="comanche.RunnableType">
  <interface name="r" signature="java.lang.Runnable" role="server"/></provides>
</definition>

<definition name="comanche.FrontendType" extends="comanche.RunnableType">
  <interface name="rh" signature="comanche.RequestHandler" role="client"/>
</definition>

<definition name="comanche.ReceiverType" extends="comanche.FrontendType">
  <interface name="s" signature="comanche.Scheduler" role="client"/>
</definition>

<definition name="comanche.SchedulerType">
  <interface name="s" signature="comanche.Scheduler" role="server"/>
</definition>

<definition name="comanche.HandlerType">
  <interface name="rh" signature="comanche.RequestHandler" role="server"/>
</definition>

<definition name="comanche.AnalyzerType">
  <interface name="a" signature="comanche.RequestHandler" role="server"/>
  <interface name="rh" signature="comanche.RequestHandler" role="client"/>
  <interface name="l" signature="comanche.Logger" role="client"/>
</definition>

<definition name="comanche.LoggerType">
  <interface name="l" signature="comanche.Logger" role="server"/>
</definition>

<definition name="comanche.DispatcherType" extends="comanche.HandlerType">
  <interface name="h"
      signature="comanche.RequestHandler" role="client" cardinality="collection"/>
</definition>

<!– ============== Primitive components ============== –>

<definition name="comanche.Receiver" extends="comanche.ReceiverType">
  <content class="comanche.RequestReceiver"/>
</definition>

<definition name="comanche.SequentialScheduler" extends="comanche.SchedulerType">
  <content class="comanche.SequentialScheduler"/>
</definition>

<definition name="comanche.MultiThreadScheduler" extends="comanche.SchedulerType">
```

```
  <content class="comanche.MultiThreadScheduler"/>
</definition>

<definition name="comanche.Analyzer" extends="comanche.AnalyzerType">
  <content class="comanche.RequestAnalyzer"/>
</definition>

<definition name="comanche.Logger" extends="comanche.LoggerType">
  <content class="comanche.BasicLogger"/>
</definition>

<definition name="comanche.Dispatcher" extends="comanche.DispatcherType">
  <content class="comanche.RequestDispatcher"/>
</definition>

<definition name="comanche.FileHandler" extends="comanche.HandlerType">
  <content class="comanche.FileRequestHandler"/>
</definition>

<definition name="comanche.ErrorHandler" extends="comanche.HandlerType">
  <content class="comanche.ErrorRequestHandler"/>
</definition>

<!- ============== Composite components ============== ->

<definition name="comanche.Handler" extends="comanche.HandlerType">
  <component name="rd" definition="comanche.Dispatcher"/>
  <component name="frh" definition="comanche.FileHandler"/>
  <component name="erh" definition="comanche.ErrorHandler"/>
  <binding client="this.rh" server="rd.rh"/>
  <binding client="rd.h0" server="frh.rh"/>
  <binding client="rd.h1" server="erh.rh"/>
</definition>

<definition name="comanche.Backend" extends="comanche.HandlerType">
  <component name="ra" definition="comanche.Analyzer"/>
  <component name="rh" definition="comanche.Handler"/>
  <component name="l" definition="comanche.Logger"/>
  <binding client="this.rh" server="ra.a"/>
  <binding client="ra.rh" server="rh.rh"/>
  <binding client="ra.l" server="l.l"/>
</definition>

<definition name="comanche.Frontend" extends="comanche.FrontendType">
  <component name="rr" definition="comanche.Receiver"/>
  <component name="s" definition="comanche.MultiThreadScheduler"/>
  <binding client="this.r" server="rr.r"/>
  <binding client="rr.s" server="s.s"/>
  <binding client="rr.rh" server="this.rh"/>
</definition>

<definition name="comanche.Comanche" extends="comanche.RunnableType">
  <component name="fe" definition="comanche.Frontend"/>
```

```
  <component name="be" definition="comanche.Backend"/>
  <binding client="this.r" server="fe.r"/>
  <binding client="fe.rh" server="be.rh"/>
</definition>
```