

Getting Started with Fractal

E. Bruneton



- ➔ **Introduction**
- ➔ **Design**
- ➔ **Implementation**
- ➔ **Configuration**
- ➔ **Reconfiguration**
- ➔ **Conclusion**

→ What is Fractal?

- a modular, extensible component model that can be used to design, implement, deploy and reconfigure:
 - various systems and applications (from OS to middleware to GUI)
 - in various programming languages (currently Java and C)
- an ObjectWeb project with 4 sub projects:
 - model, implementation, components, tools

→ Fractal heavily uses *separation of concerns*

- separation of interface and implementation, component oriented programming, inversion of control, separation of content and controller, separation of controller interfaces, separation of architecture description and deployment, ...

→ Minimal HTTP server

- listens on a server socket
- for each connection:
 - starts a new thread
 - reads the URL
 - sends the requested file
 - or an error message

```
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 _) { }
    }
}
```

→ Static components

- lifetime = application lifetime
- correspond to « services »

→ Dynamic components

- shorter life time
- correspond to « data »

→ Define components

- find services
- [identify data structures]
- one service = one component

```

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 _) { }
    }
}

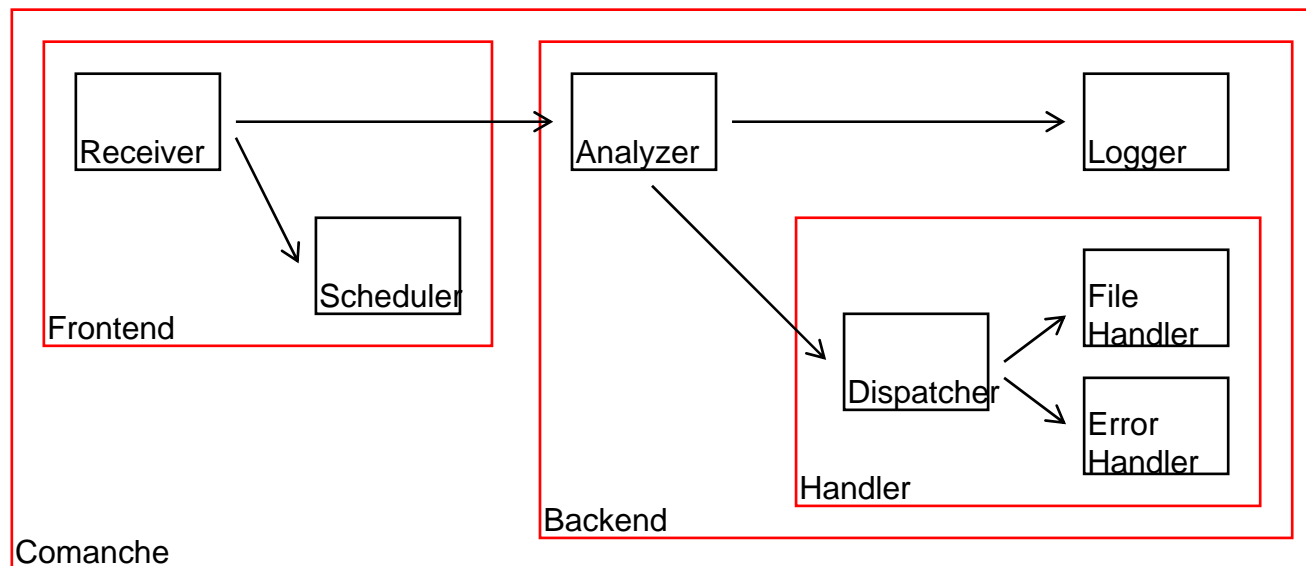
```

➔ Dependencies:

- use scenarios and use cases

➔ Hierarchical structure:

- corresponds to abstraction level



→ **Contracts between components:**

- must be designed with care: must be as stable as possible
- must deal only with functional concerns

→ **Comanche contracts**

- logger: 1 log operation, with a String parameter
- scheduler: 1 schedule operation, with a Runnable parameter
- handlers: 1 handleRequest operation
 - analyzer: reads the URL
 - dispatcher: dispatchs to associated handlers in turn, until success
 - file handler: try to read and send back the file at the requested URL
 - error handler: return an error message and always succeed

→ Choose the component's granularity

- Several design time components can be represented as a single programming time component

→ Implement the component's interfaces

- Fractal enforces a strict separation between interface and implementation

→ Comanche interfaces

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

- `public class Request { Socket s; Reader r; PrintStream out; String url; }`

➔ Components without dependencies:

- implemented as in normal Java

```
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(); }  
}
```

➔ Components with dependencies:

- first solution: does not allow static configuration

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

- second solution: does not allow dynamic reconfiguration

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

→ Components with dependencies:

```
public class RequestReceiver implements Runnable, BindingController {
    private Scheduler s;
    private RequestHandler rh;
    public String[] listFc () { return new String[] { "s", "rh" }; }
    public Object lookupFc (String n) {
        if (n.equals("s")) { return s; } else if (n.equals("rh")) { return rh; } else return null;
    }
    public void bindFc (String n, Object v) {
        if (n.equals("s")) { s = (Scheduler)v; } else if (n.equals("rh")) { rh = (RequestHandler)v; }
    }
    public void unbindFc (String n) {
        if (n.equals("s")) { s = null; } else if (n.equals("rh")) { rh = null; }
    }
    // ...
}
```

Configuration: programmatic

→ Advantages:

- most direct approach
- no tools required

→ Drawbacks:

- error prone
- the architecture is not visible
- *the architecture and deployment concerns are mixed*

```
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();  
    }  
}
```

→ Advantages

- good separation between the architecture and deployment concerns
- Allow static verifications
 - invalid or missing bindings, ...

→ Drawbacks:

- the architecture is not visible

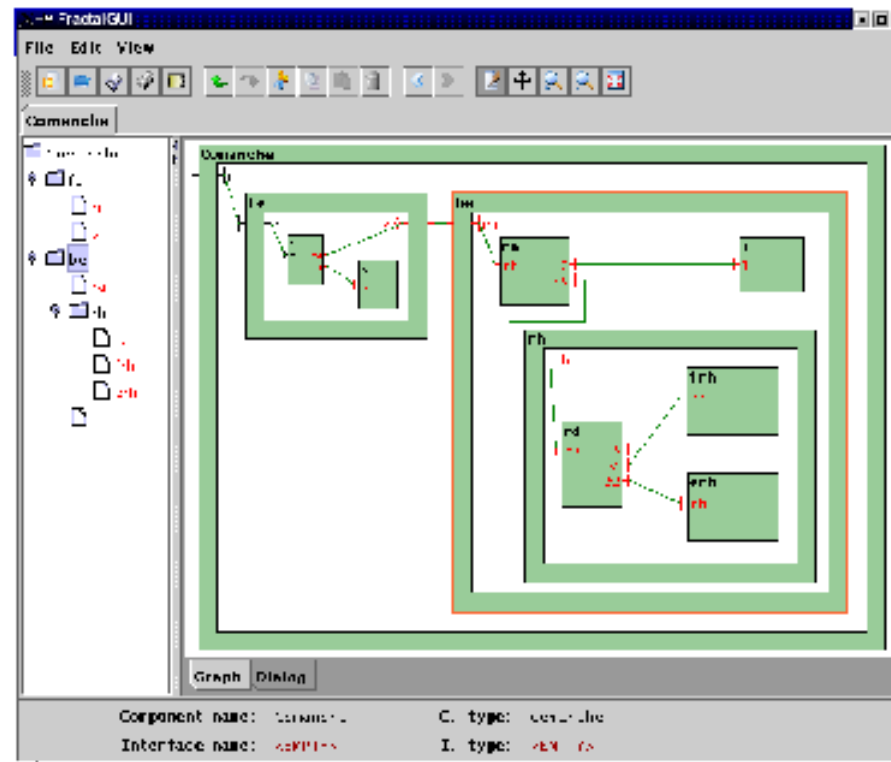
```
<component-type name="HandlerType">
  <provides>
    <interface-type name="rh" signature="comanche.RequestHandler"/>
  </provides>
</component-type>

<primitive-template name="FileHandler" implements="HandlerType">
  <primitive-content class="comanche.FileRequestHandler"/>
</primitive-template>

<composite-template name="Comanche" implements="RunnableType">
  <composite-content>
    <components>
      <component name="fe" type="FrontendType" implementation="Frontend"/>
      <component name="be" type="HandlerType" implementation="Backend"/>
    </components>
    <bindings>
      <binding client="this.r" server="fe.r"/>
      <binding client="fe.rh" server="be.rh"/>
    </bindings>
  </composite-content>
</composite-template>
```

➔ Graphical tool to edit ADL definition files

- the graph representation clearly shows the architecture



→ Reconfiguration:

- static: stop the application, change the ADL file, restart
 - not always possible, e.g. if the application must be always available
- dynamic: reconfigure the application while it is running
 - introduces consistency problems
 - basic tool to help solve them: component life cycle management

→ Example:

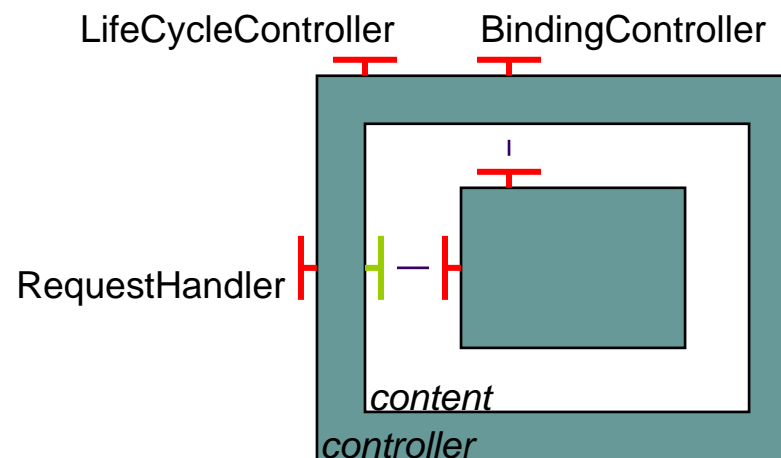
- replace the FileHandler component dynamically:
 - RequestHandler rh = **new** FileAndDirectoryRequestHandler();
 - rd.unbindFc("h0");
 - rd.bindFc("h0", rh);
 - for safety, rd (RequestDispatcher) must be suspended and resumed

➔ LifeCycleController interface: first solution

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


→ LifeCycleController interface: better solutions

- implement this interface in a separate class or in a sub class
 - better separation of concerns
 - components can then be deployed with or without life cycle management
- generate this separate class or sub class automatically
 - dynamic or static code generation



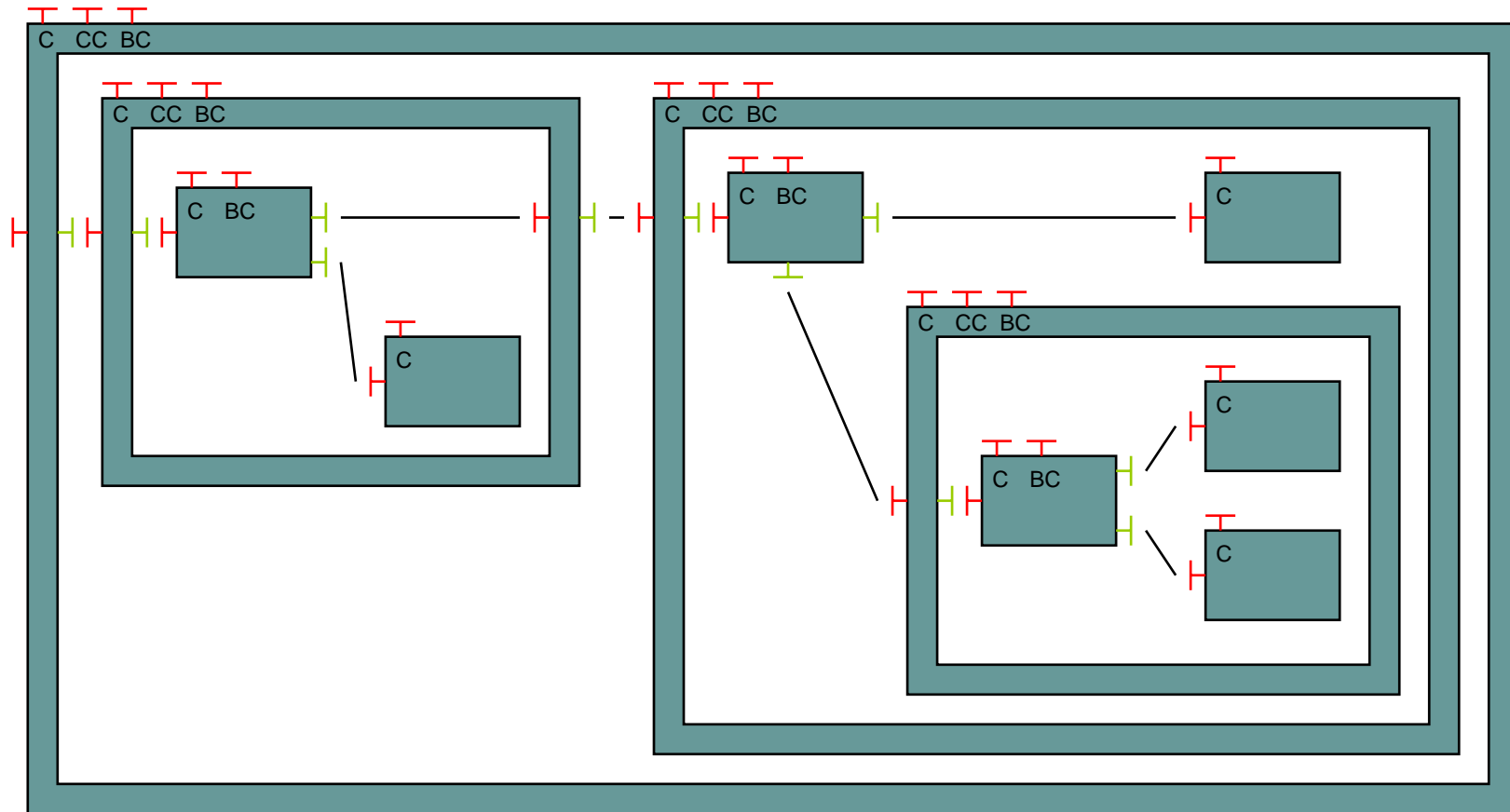
➔ **Before stopping and reconfiguring a component, one must get a reference to it**

➤ hence the following introspection (and reconfiguration) interfaces:

```
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);  
}
```

Some deployment choices



→ Fractal

- uses well known design patterns and organize them into a uniform, extensible, language independent component model

→ Benefits

- enforces separation of interface and implementation
 - ensures a minimum level of flexibility
- enforces separation of the functional, configuration and deployment concerns
 - allows applications to be instantiated in various ways
 - from fully optimized but unconfigurable configurations to less optimized but dynamically reconfigurable configurations