Motivations

Component-Oriented Programming is expensive!
- Several files per component: Interfaces, implementation, meta-data files

Business and technical code weaving
- Component lifecycle, bindings, attributes handled in program code

Component meta-information redundancy
- Some meta-data duplicated in both program code and ADL

These drawbacks impact
- Development time (technical code cost, which is error prone)
- Coherency maintenance cost (architecture description ⇔ program code)
- Evolution support (application, component model)
Fractal Component Model

An interesting component model :-)  

- Various implementations: Julia, AOKell, ProActive, etc.  
- Various extensions: FAC, Confract, etc.  
- Various tools: Fractal ADL, Fractal GUI, Fractal Explorer, SAFRAN, etc.  
- Various applications: Dream, CLIF, Speedo, GoTM, etc.

BUT, it may quickly become boring :-(

- Implementation of the component business code  
- Definition of an AttributeController interface  
- Implementation of Fractal controller callbacks 
  - BindingController: to support client interfaces  
  - LifeCycleController: to be notified of start/stop transitions  
  - XXXAttributeController: to support attribute (re)configuration  
- Definition of the primitive component ADL description  
- Definition of the composite component ADL description

Example: Binding Controller Overhead

```java
public class ClientImpl implements Main, BindingController {
    private Service service;
    public void main (final String[] args) {
        service.print("hello world");
    }
    public String[] listFc () { return new String[] {"s"]; }
    public Object lookupFc (final String cItf) { 
        if (cItf.equals("s")] { return service; }
        return null;
    }
    public void bindFc (final String cItf, final Object sItf) {
        if (cItf.equals("s")] { service = (Service)sItf; }
    }
    public void unbindFc (final String cItf) {
        if (cItf.equals("s")] { service = null; }
    }
}

BEST OVERHEAD (lines) = 5 + 3 * NB-BINDINGS
```
Fraclet: @OP for Fractal

Application of Attribute-Oriented Programming principles

• Annotate the program code with common component concerns
• Generate the technical code for the Fractal component model

Annotations define only the component semantics

• No assumption on the technical stuff
• Extensible to additional concerns

Generators complete the component program code

• Component glue (programming model)
• Architecture descriptions (Fractal ADL)

```java
@Interface(name="r",signature=Runnable)
public class MyComponent implements Runnable {
    @Binding
    protected Runnable delegate;
    @Attribute
    protected int repeat;
    @Lifecycle(on="stop")
    protected void onStop() {
        System.out.println("Stopping…");
    }
    public void run() {
        for (int i=0;i<this.repeat;i++)
            this.delegate.run();
    }
}
```

Overview of Fraclet Annotations

Fraclet defines 7 annotations

• 4 structural annotations
  – `@interface <CLASS>`: describes an interface provided by the component
  – `@binding <FIELD>`: describes an interface required by the component
  – `@attribute <FIELD>`: describes a component attribute
  – `@component <CLASS>`: describes the component membrane

• 3 behavioral annotations
  – `@lifecycle <METHOD>`: handles lifecycle transitions
  – `@controller <FIELD>`: accesses the controller part of the component
  – `@logger <FIELD>`: defines a component logger

Fraclet supports 2 types of annotations

• XDoc annotations (Javadoc comments) for Java <= 1.5
• Java5 annotations for Java >= 1.5
Overview of Fraclet Generators

Fraclet provides 5 generators

- 2 Java generators
  - AttributeController interface
  - Component glue
- 2 Fractal ADL generators
  - Primitive definition
  - Abstract composite definition
- 1 Monolog generator
  - Monolog configuration

Fraclet supports 2 generation engines

- XDoclet: uses XDoc annotations to extend the content class
- Spoon: uses Java5 annotations to modify the content class

Revisiting HelloWorld...

```java
/** @interface name=m */
public interface Main {
    void main (String[] args);
}

public class ClientImpl implements Main {
    /** @binding name=s */
    protected Service service;
    public void main (final String[] args) {
        service.print("hello world");
    }
}

public class ServerImpl implements Service {
    /** @attribute */
    protected String header;
    /** @attribute */
    protected int count;
    public void print (final String msg) {
        for (int i = 0; i < count; ++i)
            System.err.println(header + msg);
    }
}
```

<definition name="HelloWorld" extends="ClientImplComp">
    <component name="s" definition="ServerImpl(\(\rightarrow\),3)">
</definition>
Fraclet Benefits

Reduction of the program code size
- Between 40% and 70% of the original code size
- Binding controller overhead: from $5 + 3 \times \text{NB-BINDINGS}$ to $\text{NB-BINDINGS}$
- But this is not the only benefit …

Simplification of component-oriented programming
- Hiding technical details
- Ensuring coherency between program code and artifacts

Evolution support
- Software evolution
  - evolution of the program code (e.g., adding an attribute)
- Component model evolution
  - evolution of the component specification (e.g., modification of the API)

Related Work

Generative Programming
- Generating technical code from ADL descriptions
- Generating technical code & ADL from model descriptions (e.g., FractalGUI)
  - + Enforces separation of concerns (architect $\Rightarrow$ developer)
  - -- Increases the size of handwritten code (ADL + business code)
  - -- No support for code instrumentation (controllers, logging, etc.)

Aspect-Oriented Programming
- Limited to code advising and injection (e.g., AOKell)
- No support for external artifacts generation

Complementary approaches
- Providing reverse engineering support for generative programming approaches
  (UML $\Rightarrow$ ADL $\Rightarrow$ code $\Rightarrow$ ADL $\Rightarrow$ UML)
Fraclet Perspectives

Program code validation
- Component model specification
- Hidden communication path detection

Reverse engineering support: e.g., Fractal GUI
- Annotated program code generation using Fractal GUI
- Fractal GUI description file generation using Fraclet

Component model independency
- Developing the application only once (PIC - Platform Independent Code)
- Executing on various component models (PSC: Fractal, OpenCOMJ, etc.)

Fractal Explorer support
- E.g., defining annotations to mark invokable methods
- E.g., generating Fractal Explorer configuration file

Conclusion

Fraclet addresses the problem of business and technical code tangling
- Non-functional code merged with business code
- Redundancy of some meta-data between program code and ADL

Fraclet proposes to apply Attribute-Oriented Programming to Fractal
- 7 annotations (structural, behavioral) using Java5 or Xdoc annotations
- 5 generators (Java, Fractal ADL, Monolog) using Spoon or Xdoclet engines

Fraclet provides added values to Fractal-Based developments
- Continuous integration of the technical code
- Reduction of the program code size (~50%)
- Support for application and component model evolution

Fraclet implementations are available at http://fractal.objectweb.org/
Questions?

Fraclet is already applied in
- **GoTM**: a framework for the construction of transaction services
- **ProActive**: a platform for Grid Computing
- **COSMOS**: a framework for the composition of system resources
- **Deployment Framework**: a generic deployment tool
- **DACAR**: an autonomous deployment framework
- **FAC**: an extension to support AOP at component level
- …