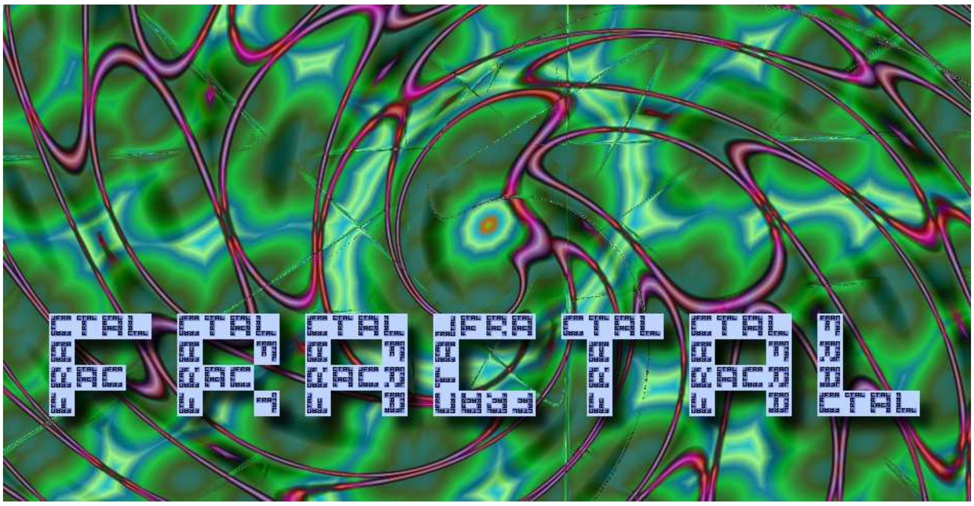
The ObjectWeb Consortium

Specification

The Fractal Component Model



Authors:

E. Bruneton	(France Telecom R&D)
T. Coupaye	(France Telecom R&D)
J.B. Stefani	(INRIA)

Released	February 5, 2004
Status	Draft
Version	2.0-3

General Information

- Authors are given in alphabetical order.
- Background of front-page image appears courtesy of Giuseppe Zito.
- Please send technical comments on this specification to fractal@objectweb.org. Authors would be glad to hear from people using, implementing or extending Fractal.

Trademarks

Disclaimer of warranties

Contents

List of Figures

1. Introduction

1.1. Rationale

By enforcing a strict separation between interface and implementation and by making software architecture explicit, component-based programming can facilitate the implementation and maintenance of complex software systems. Coupled with the use of meta-programming techniques, component-based programming can hide to application programmers some of the complexities inherent in the handling of non-functional aspects in a software system, such as distribution and fault-tolerance, as exemplified e.g. by the container concept in Enterprise Java Beans (EJB), CORBA Component Model (CCM), or Microsoft .Net.

Existing component-based frameworks and architecture description languages, however, provide only limited support for extension and adaptation. This limitation has several important drawbacks: it prevents the easy and possibly dynamic introduction of different control facilities for components such as non-functional aspects; it prevents application designers and programmers from making important trade-offs such as degree of configurability vs performance and space consumption; and it can make difficult the use of these frameworks and languages in different environments, including embedded systems.

The Fractal component model alleviates the above problems by introducing a notion of component endowed with an open set of control capabilities. In other terms, components in Fractal are reflective, and their reflective capabilities are not fixed in the model but can be extended and adapted to fit the programmer's constraints and objectives.

1.2. Overview

Main goals of the Fractal component model are to implement, deploy and manage (i.e. monitor and dynamically reconfigure) complex software systems. These goals motivate the main features of the Fractal model: composite components (to have a uniform view of applications at various abstraction levels), shared components (to model resources), introspection capabilities (to monitor a running system), and configuration and reconfiguration capabilities (to deploy and dynamically reconfigure an application). But another goal of the Fractal model is to be applicable to many software, from embedded software to application servers and information systems. Unfortunately, the advanced features of the Fractal model have a cost that is not always compatible with the limited resources of constrained environments.

In order to achieve these contradictory goals, the Fractal component model is not defined as a big, fixed specification that all Fractal components must follow, but rather as an extensible system of relations between well defined concepts and corresponding APIs that Fractal components *may* or *may not* implement, depending on what they can or want to offer to other components. This set of specifications can be organized, and is presented in this document as increasing "levels of *control*", i.e. in increasing order of *reflective* capabilities (*introspection* and *intercession*).

At the lowest level of control, a Fractal component is a runtime entity that does not provide any control capability to other components, and is therefore like an object (such a component can be used in only one way, namely by calling methods on it). In fact, an object *is* a Fractal component without any control capability (see section 2). This feature is useful to handle cases where components have to be connected to legacy software.

At the next level of control capability, which can be called the external "introspection" level (see section 3), a Fractal component can provide a standard interface, similar to the IUnknown interface in COM, that allows one to discover all its external interfaces or, in other words, its boundary (like an object, a Fractal component can provide several interfaces).

At the next level of control capability, which can be called the "configuration" level (see section 4), a Fractal component can provide control interfaces to introspect and modify its *content*, i.e. what is inside its boundary. In the Fractal model, this content is made of other Fractal components, called its *sub components*, bound together through *bindings*. A Fractal component can therefore choose to provide or not an interface to control the set of its sub components, the set of bindings between these sub components, and so on.

In addition to these control capabilities, the Fractal model also specifies a framework for the instantiation of components (see section 5), and a simple type system for Fractal components (see section 6). Like the above control capabilities, the instantiation framework and the simple type system are optional. In fact, in the Fractal model, *everything* is optional. This has advantages and drawbacks, which are discussed in section 7.

As a result of this modular and extensible organization (anyone is free to define its own control interfaces, in order to provide new introspection and intercession capabilities), and given the fact that the Fractal component model is not tied to a specific language, Fractal components can be used in very different situations, from operating systems to middleware platforms, from graphical user interfaces to information systems, and from highly optimized but unreconfigurable configurations, to less optimized but highly dynamic and reconfigurable systems or applications.

2. Foundations

At the lowest control capability level, a Fractal component does not provide any introspection or intercession function to other components. Such a component, called a *base component*, can be used in only one way, namely by invoking operations on its component interfaces. A *component interface* is an *access point* to a component that implements a *language interface*. *Component interfaces and language interfaces should not be mixed up*: a component interface is an access point that *implements* a language interface; a language interface is a *type*. Despite of this risk, sentences such as “a component has an interface that implements the language interface X” will often be abbreviated, in the rest of this document, into “a component has an interface X”, in order to improve readability.

This section defines the pseudo Interface Definition Language used in the rest of this document to define the language interfaces implemented by Fractal component interfaces. It also defines a framework to get access to component interfaces.

2.1. Interface Definition Language

In order to allow Fractal components implemented in potentially distinct programming languages to interoperate, some standard protocols for local and remote operation invocations are necessary. One way to ensure this is to use an Interface Definition Language (IDL), and mappings from the IDL to existing programming languages. The IDL compiler can then generate stubs and skeletons to make the conversion from language specific protocols to standard protocols, and vice versa.

Despite of this, in the Fractal component model, *IDLs and mappings are optional*, as everything else. This means that Fractal component interfaces can be defined either directly in *any* programming language, or indirectly via *any* IDL. As a consequence, the constraints and costs associated to the use of an IDL do not have to be paid for, if interoperability is not needed.

In this document the component interfaces are specified in a pseudo IDL language (see below), in order to show that the Fractal component model does not enforce the use of any existing programming or interface definition language. A possible definition of these interfaces is also given in Java, C and OMG IDL (see Appendix A). These definitions can be used as “standard” definitions to provide interoperability between Java components only, between C components only, and between any components (respectively).

The non normative, pseudo IDL language used in this document for illustration purposes is a modified subset of Java, so as to be immediately understandable by Java programmers. In this language, an interface definition is a Java interface definition with the following restrictions:

- modifiers (**public**, **final**, ...) are not allowed;
- field (i.e. constant) declarations are allowed only with literal expressions;
- inner interface and class definitions are not allowed;
- class types are not allowed in array types, formal parameter types and return types.

and with the following extensions:

- a new **any** type, meaning any interface type, is available to replace java.lang.Object;
- a new **string** primitive type is available to replace java.lang.String.

Exceptions are allowed in operation declarations, *but are not considered as classes*, as in Java: they are instead considered as abstract names, denoting categories of errors.

2.2. Naming and binding

In order to invoke operations on a component interface, one must first identify the interface to be called, and then get an access to this interface. This section defines a framework for doing so, based on names, naming contexts and binders (see Figure 1). This framework is mainly designed to access remote interfaces, but can also be used in a single address space.

```
package org.objectweb.naming;

interface Name {
  NamingContext getNamingContext ();
  byte[] encode () throws NamingException;
}

interface NamingContext {
  Name export (any o, any hints) throws NamingException;
  Name decode (byte[] b) throws NamingException;
}

interface Binder extends NamingContext {
  any bind (Name n, any hints) throws NamingException;
}
```

Figure 1: Naming API

A *name* designates a component interface. Names can take many forms, such as Java references or Interoperable Object References (IORs). A name does not necessarily give direct access to the interface it designates (a CORBA IOR does not give direct access to the designated remote interface; on the contrary, a Java reference can be used directly to call methods on the designated interface). A name is represented by the Name interface.

A name is always associated to a *naming context*, and is generally invalid outside this context. For example the naming context of a Java reference is the Java Virtual Machine (JVM) in which the designated object resides. This name is meaningless outside this context and, in particular, in another JVM. The naming context of an IOR is the CORBA IOR "name space". An IOR is meaningless outside this context and, in particular, in the Java RMI over JRMP context. The Name interface specifies a getNamingContext operation, which returns the naming context of the name on which this operation is invoked.

A name can be serialized in many forms, such as a string or a byte array. For example a Java reference can be serialized as a string representing the memory address of the object, in decimal or hexadecimal form. An IOR can be serialized as a string containing a host name or IP address, a TCP port number, and an object key. In serialized form, a name can be sent over a network, or stored in a file or a database. The Name interface specifies an encode operation, which returns an encoded form of the name on which this operation is invoked, as a byte array.

A naming context is represented by the NamingContext interface. A naming context creates and manages names in its context. In particular, a naming context can create a name for a given component interface. The NamingContext interface specifies an export operation for doing that: this operation takes as argument a component interface and optional hints, and returns a name for this interface. A naming context can also deserialize names in serialized form. The NamingContext interface specifies a decode operation for doing that: this operation takes as argument a serialized name, as a byte array, and returns the corresponding name.

In order to access the interface designated by a name, a *binding* must be established to this interface. For example, in order to access a remote interface designated by an IOR, a socket must be opened to send an invocation message to the remote interface. These bindings are created by *binders*. A binder is represented by the Binder interface, which extends the NamingContext interface. This interface specifies a bind operation that takes a name as parameter, creates a binding to the interface designated by this name, and returns a delegate (or proxy) of this interface, or the interface itself, to invoke operations on it.

The org.objectweb.naming.NamingException exception must be thrown when an error occurs in the operations of the Name, NamingContext and Binder interfaces.

3. Introspection

At the next control capability level, beyond the base level where components do not provide any control function, a Fractal component can provide introspection functions to introspect its *external* features, i.e. its boundary. This section defines more precisely the external features of Fractal components, and specifies the interfaces related to the introspection of these features. *The interfaces related to the introspection (and reconfiguration) of the internal features of Fractal components are specified in the next section.*

3.1. External component structure

Depending on the level of observation, or *scale*, a Fractal component can be seen as a black box or as a white box. When seen as black box, i.e. when its internal organization is not visible, the only visible details of a Fractal component are some *access points* to this black box, called its *external interfaces* (see Figure 2). Each interface has a name, in order to distinguish it from the other interfaces of the component (a component can have several interfaces implementing the same language interface). All the external interfaces of a component must have distinct names, but two interfaces in two distinct components may have the same name. One may distinguish two kinds of interfaces: a *client* (or *required*) interface emits operation invocations, while a *server* (or *provided*) interface receives them.

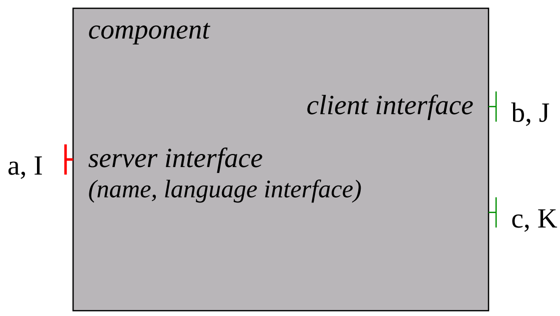


Figure 2: External view of a Fractal component

The interfaces of a component can be introspected with two language interfaces, specified in the next two sections: one to get the list of interfaces of a component, and one to introspect the interfaces themselves. These two interfaces are of course optional, as everything in the Fractal model: a component can provide both interfaces, only the first one, or none of them.

3.2. Component introspection

In order to discover the external interfaces of a component, a component can provide an interface that implements the Component interface (see Figure 3). This language interface provides two operations named getFcInterfaces and getFcInterface, that can be used to retrieve the interfaces of the component. The first operation takes no arguments, and returns an array containing all the external interfaces, either client or server, of the component, including the

Component interface. The second operation takes the name of an interface as parameter, and returns this interface, if it exists.

```
package org.objectweb.fractal.api;

interface Component {
   any[] getFcInterfaces ();
   any getFcInterface (string itfName) throws NoSuchInterfaceException;
   Type getFcType ();
}

interface Type {
   boolean isFcSubTypeOf (Type t);
}
```

Figure 3: Component introspection API

The getFcInterfaces and getFcInterface operations return references that give access to requested interfaces. In other words, the references returned by these operations can be used directly, after an appropriate cast, to invoke operations on the component's server interfaces (no explicit binding is needed). For example, if a component has a server interface named account implementing the language interface Account, then the getBalance operation of this interface can be invoked with a code like ((Account)c.getFcInterface("account")).getBalance(), where c is a reference to the Component interface of the component.

The Component interface also provides a getFcType operation, which returns the type of the component, as a Type reference. This interface defines a minimal notion of type, which actually defines only one operation named isFcSubTypeOf, whose role is to test if a given type is a sub type or not of another type. This interface can be extended to define more useful type systems for components and component interfaces, such as the one defined in section 6.

The org.objectweb.fractal.api.NoSuchInterfaceException exception must be thrown in the getFcInterface operation when a requested component interface is not found.

A component interface implementing Component must be named component.

3.3. Interface introspection

By default the references returned by the getFcInterface and getFcInterfaces operations provide access to the requested interfaces, and nothing more. In particular, it is impossible to find the names of these interfaces. In order to provide such interface introspection functions, a component can ensure that the references returned by the above operations are castable into the Interface type (see Figure 4). This interface specifies four operations to get the name of a component interface, to get its type (as a Type reference), to get the Component interface of the component to which it belongs, and to test if the interface is internal or not (see section 4.1).

Note that the getFcItfOwner operation allows one to discover all the interfaces of a component from any interface of this component, and not only from its Component interface. For example, if a is a reference to the Account interface of a such component, the Component interface of this

```
package org.objectweb.fractal.api;

interface Interface {
   string getFcItfName ();
   Type getFcItfType ();
   Component getFcItfOwner ();
   boolean isFcInternalItf ();
}
```

Figure 4: Interface introspection API

component can be retrieved with a code like ((Interface)a).getFcItfOwner(), if the component provides interface introspection functions. The result can then be used to get the reference of any other interface of the component.

Notes

- Component and Interface have very distinct roles and should not be mixed up. On the one hand, Component is a language interface that is provided by a component just like any other language interface. On the other hand, in the case of a component providing interface introspection, Interface is a language interface that is implemented by *all* the component interfaces: any component interface of a such component implements both a specific language interface, such as Account or Component, *and* Interface.
- A functional interface such as Account is likely to provide a getName or getOwner operation, as the Interface interface. And since, in the case of components that provide interface introspection, a reference of one type should be castable to the other, there is a risk of name conflicts (at least in some languages, such as Java). In order to reduce these risks, the Interface operations follow the pattern *verb*Fc*noun*, where Fc stands for Fractal component. This pattern has then been generalized to all the Fractal APIs.
- Providing the Component interface is quite easy, but supporting the Interface interface can imply some runtime time overheads. This is why this interface is optional.

4. Configuration (introspection & intercession)

At the next level of control capability, beyond the "introspection" level where components provide interfaces to introspect their external features, a Fractal component can provide control interfaces to introspect and reconfigure its *internal* features. This section defines these internal features, and specifies some possible interfaces to introspect and reconfigure them.

4.1. Internal component structure

Internally, a Fractal component is formed out of two parts: a *controller* (also called membrane), and a *content* (see Figure 5). The content of a component is composed of (a finite number of) other components, called *sub components*, which are under the control of the controller of the enclosing component. The Fractal model is thus recursive and allows components to be nested (i.e. to appear in the content of enclosing components) at an arbitrary level. A component that exposes its content is called a *composite* component. A component that does not expose its content, but has at least one control interface (see below), is called a *primitive* component. A component without any control interface is called a base component (see section 2).

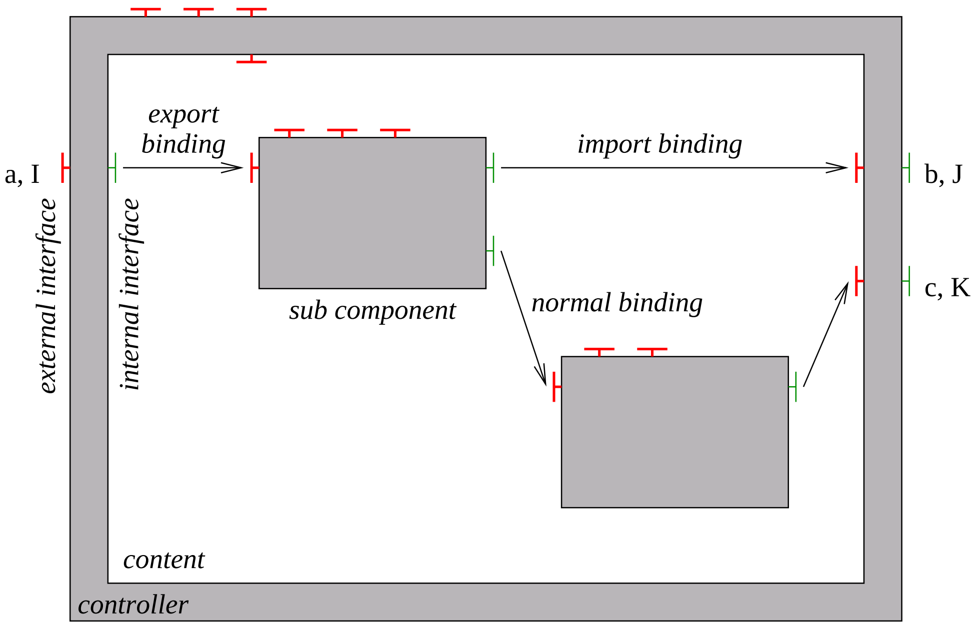


Figure 5: Internal view of a Fractal component

The controller of a component can have *external* and *internal* interfaces. External interfaces are accessible from outside the component, while internal interfaces are accessible only from the component's sub components. All the external interfaces of a component must have distinct names, all its internal interfaces must have distinct names, but a component can have an external and an internal interface of the same name. A *functional* interface is an interface that corresponds to a provided or required functionality of a component, while a *control* interface is a *server* interface that corresponds to a "non functional aspect", such as introspection, configuration or reconfiguration, and so on. By convention, an interface is considered to be a control interface if its name is equal to `component`, or ends with `-controller`. All other interfaces are considered to be functional interfaces.

The controller of a component embodies the control behavior associated with a particular component. In particular, a component controller can:

- Provide an explicit and causally connected representation of the component's sub components;
- Intercept oncoming and outgoing operation invocations targeting or originating from the component's sub components;
- Superpose a control behavior to the behavior of the component's sub components, including suspending, check pointing and resuming activities of these sub components.

Each component controller can thus be seen as implementing a particular semantic of composition for the component's sub components. The control capability of a controller is not limited by the model. For instance, it can be mainly interception-based as in industrial component frameworks containers for instance; or it can be void (i.e. no control is exercised - in this case, the controller can still be useful for it can provide a representation of its content and manifest a containment relationship).

A component may appear in the content of (be *shared* by) several distinct enclosing components (see section 4.4 and Figure 9). A component that is shared among two or more distinct components is subject to the control of their respective controllers. The exact semantics of the resulting configuration (e.g. which control behavior is enacted) is in general determined by an encompassing component that encloses all the relevant components in the configuration.

A *binding* is a communication path between component interfaces. The Fractal model distinguishes between *primitive bindings* and *composite bindings*. A primitive binding is a binding between one client interface and one server interface, in the same address space, which means that the operation invocations emitted by the client interface should be accepted by the specified server interface. A primitive binding between a client interface c and a server interface s of two components C and S must verify one of the following constraints (see Figure 5):

- c and s are external interfaces, and C and S have a direct common enclosing component. Such bindings are called *normal bindings*.
- c is an internal interface, s is an external interface, and S is a sub component of C. Such bindings are called *export bindings*.
- c is an external interface, s is an internal interface, and C is a sub component of S. Such bindings are called *import bindings*.

In addition to these structural constraints, which ensure that primitive bindings cannot "cross" component boundaries except through interfaces, a primitive binding can be established between a client and a server interface only if the server interface can accept at least all the operation invocations that the client interface can emit. In other words, the (language) type of the server interface must be a sub type of the type of the client interface (the two interfaces can of course be of the same type since a sub typing relation must be reflexive). The last constraint is that a client interface can be bound to at most one server interface, while several client interfaces can be bound to the same server interface.

A composite binding is a communication path between an arbitrary number of component interfaces, of arbitrary language types. These bindings are represented as a set of primitive bindings and *binding components* (stubs, skeletons, adapters, ...). A binding component is a normal Fractal component, whose role is dedicated to communication. Binding components are also called *connectors*: hence Fractal *does* have connectors, although this concept is not a core concept here, as component or interface - this is why there is no special API for them. It should be noted that the binding concept defined here is exactly is the same as the binding concept defined in section 2.2: in particular, primitive bindings correspond to local bindings, i.e. to bindings inside a single address space.

4.2. Attribute control

An *attribute* is a configurable property of a component, such as the text or color of a button, or the maximum size of a pool or cache component. Attributes are generally of primitive type, and are used to configure the state of components without needing to use bindings (it is possible to configure the text of a button, for example, by binding this button component to a text component; but this is overly complex for what is needed). A component can provide an AttributeController interface to read and write its attributes from outside the component (see Figure 6).

```
package org.objectweb.fractal.api.control;
interface AttributeController { }
```

Figure 6: Attribute control API

In this case, the component must actually provide a sub interface of this interface, since the AttributeController interface is in fact empty. This sub interface must contain one getter and/or setter operation per configurable attribute. For example:

- a component that wants to provide an AttributeController interface for a read only string attribute foo must provide a sub interface of this interface containing the following operation: string getFoo();
- a component that wants to provide an AttributeController interface for a write only string attribute foo must provide a sub interface of this interface containing the following operation: void setFoo(string foo);
- a component that wants to provide an AttributeController interface to configure two string attributes foo and bar must provide a sub interface of this interface containing the following operations: string getFoo(), void setFoo(string foo), string getBar() and void setBar(string bar);

It is a requirement of this specification that setters and getters must follow the lexicographic and typing conventions introduced informally in the example above with respect to names and signatures of setters and getters (these conventions are those of the Java Beans component model).

A component interface implementing AttributeController must be named attribute-controller.

4.3. Binding control

A component can provide the BindingController interface to bind and unbind its client interfaces to other components through *primitive* bindings (see Figure 7).

```
package org.objectweb.fractal.api.control;

interface BindingController {
  string[] listFc ();
  any lookupFc (string clientItfName)
    throws NoSuchInterfaceException;
  void bindFc (string clientItfName, any serverItf)
    throws NoSuchInterfaceException, IllegalBindingException, IllegalLifeCycleException;
  void unbindFc (string clientItfName)
    throws NoSuchInterfaceException, IllegalBindingException, IllegalLifeCycleException;
}
```

Figure 7: Binding control API

This interface defines the following operations:

- The listFc operation returns the names of the client interfaces of the component. These names are the names that can be passed as first argument to the lookupFc operation.
- The lookupFc operation takes as parameter the name of a client interface of the component, either external or internal, and returns the server interface that is bound to this client interface, or null if there is no such interface. If the component to which the server interface belongs supports the interface introspection (see section 3.3), the reference returned by this operation can be cast to Interface.
- The bindFc operation takes as parameters the name of a client interface of the component, either external or internal, and a server interface of another component, and binds these two interfaces together. As above, the server interface can be cast or not to Interface, depending on the introspection capabilities provided by the server component.
- The unbindFc operation takes as parameter the name of a client interface of the component, either external or internal, and unbinds this interface.

These operations *may* throw a NoSuchInterfaceException exception if a specified client interface does not exist, an IllegalLifeCycleException exception when a component is not in an appropriate state to perform an operation, and an org.objectweb.fractal.api.control.IllegalBindingException exception in case of other errors related to bindings.

A component interface implementing BindingController must be named binding-controller.

Composite bindings are created with the general naming and binding framework (see section 2.2). For example, in order to bind two component interfaces that are not in the same address space, a binding component must first be created between the two interfaces, by using the naming and binding framework (typically, the server interface will be exported in some distributed NamingContext, the returned Name will be encoded, sent over the network, decoded, and finally a Binder will be used to create the binding component from this name; all this can be done explicitly or implicitly, as the effect of passing an interface reference in a remote operation invocation). The BindingController interface will then be used to create the primitive bindings between the client interface and the binding component, and between the binding component and the server interface.

4.4. Content control

A component can provide the ContentController interface to add and remove sub components in this component (see Figure 8).

```
package org.objectweb.fractal.api.control;

interface ContentController {
  any[] getFcInternalInterfaces ();
  any getFcInternalInterface (string itfName) throws NoSuchInterfaceException;
  Component[] getFcSubComponents ();
  void addFcSubComponent (Component c)
    throws IllegalContentException, IllegalLifeCycleException;
  void removeFcSubComponent (Component c)
    throws IllegalContentException, IllegalLifeCycleException;
}
interface SuperController {
  Component[] getFcSuperComponents ();
}
```

Figure 8: Content control API

This interface defines three operations to get the list of sub components of a component, and to add and remove sub components in a component:

- the getFcSubComponents operation returns the list of sub components of the component, as an array of Component references. The getFcSuperComponents operation, in the SuperController interface (see Figure 8), provides the opposite function: it returns the components that contain this component, and which are called its *super* components.
- the addFcSubComponent operation takes a component c as parameter, as a Component reference, and adds this component to the component's content. If C and C' are the set

of sub components of the component before and after c is added, then $c \in C'$ but C' is not necessarily equal to $C \cup \{c\}$.

- the removeFcSubComponent operation takes a component as parameter, as a Component reference, and removes this component from the component's content. If C and C' are the set of sub components of the component before and after c is removed, then $c \notin C'$ but C' is not necessarily equal to $C - \{c\}$.

A given component can be added to several other components. Such a component is said to be *shared* between these components. Shared components are useful, paradoxically, to preserve component encapsulation. Consider, for example, a menu and a toolbar components (see Figure 9), with an "undo" toolbar button corresponding to an "undo" menu item. It is natural to represent the menu items and toolbar buttons as sub components, encapsulated in the menu and toolbar components, respectively. But, without sharing, this solution does not work for the "undo" button and menu item, which must have the same state (enabled or disabled): these components, or an associated state component, must be put outside the menu and toolbar components. With component sharing, the state component can be shared between the menu and toolbar components, in order to preserve component encapsulation. Shared components are also useful to help separate "aspects" in component based applications. For example, a shared logger component allows one to avoid adding a Logger client interface to many components.

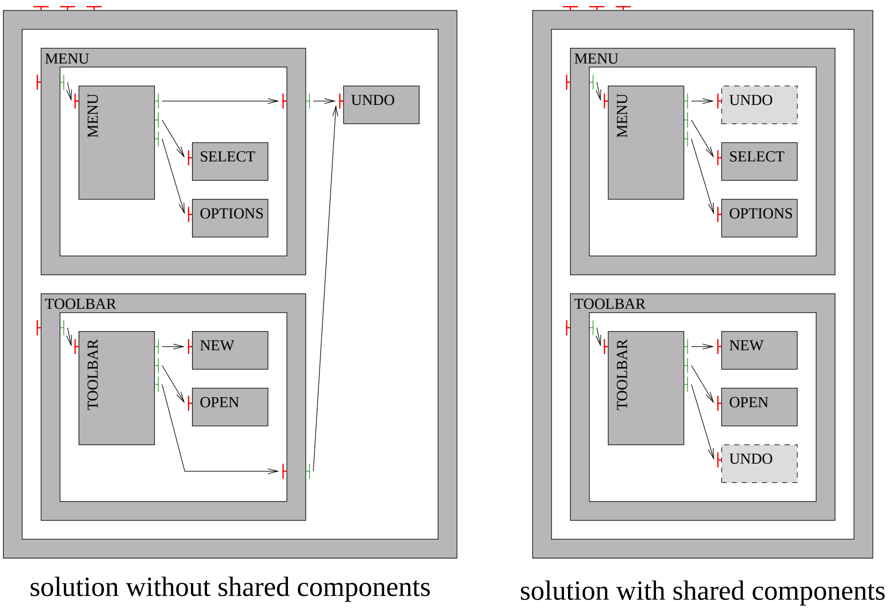


Figure 9: Advantages of shared components

Because of shared components, the structure of a Fractal component, in terms of direct and indirect sub components, is not necessarily a tree, but can be a directed acyclic graph (it cannot be an arbitrary graph, because a component cannot be added inside itself or inside one of its direct or indirect sub components). In terms of bindings, this structure can be arbitrary, provided it follows the constraints of section 4.1. In particular, bindings can form cycles.

The ContentController interface also specifies two operations to get the internal interfaces of the component, which are similar to the getFcInterface and getFcInterfaces operations. These operations are useful to bind the internal interfaces to sub components.

The content controller operations *may* throw a NoSuchInterfaceException exception if a specified client interface does not exist, an IllegalLifeCycleException exception when a component is not in an appropriate state to perform an operation and, in case of other errors related to content control, an org.objectweb.fractal.api.control.IllegalContentException exception.

A component interface implementing ContentController (resp. SuperController) must be named content-controller (resp. super-controller).

Note

In order to associate local names to the sub components of a component, similar to the local names of the interfaces of a component, a possibility is to ensure that all these sub components provide the NameController interface defined in Figure 10.

```
interface NameController {
   string getFcName ();
   void setFcName (string name);
}
```

Figure 10: Name control API

A component interface implementing NameController must be named name-controller.

4.5. Life cycle control

Changing an attribute or a binding, or removing a sub component, with the above control interfaces, and while components are executing, can be dangerous: messages can be lost, the application's state may become inconsistent, or the application may simply crash. In order to provide a minimal support to help implement such dynamic reconfigurations, a component can provide the LifeCycleController interface (see Figure 11).

```
package org.objectweb.fractal.api.control;

interface LifeCycleController {
   string getFcState ();
   void startFc () throws IllegalLifeCycleException;
   void stopFc () throws IllegalLifeCycleException;
}
```

Figure 11: Life cycle control API

This interface provides two operations named startFc and stopFc, to start and stop a component properly. As for the addFcSubComponent and removeFcSubComponent operations, the semantics of these operations is voluntarily as weak as possible, so that many implementations are possible: these operations may or may not be recursive, i.e. starting or stopping a component may or may not automatically start or stop all its direct and indirect sub components. Likewise, the effect of these operations on the component's state is voluntarily not specified (in fact it cannot be specified here, because the APIs defined in this document do not provide access to this state). In particular, the stopFc operation can be seen as a clean up operation invoked before the component is destroyed, or as a suspend operation. In the first case the component's state will be erased, while in the second case it will be left unchanged.

In addition to these operations, the LifeCycleController interface also provides a getFcState operation. This operation returns the current state of the component (in a strict sense, i.e. without taking into account its sub components, which can have a different execution state), as a string. The STARTED and STOPPED strings mean that the component is started or stopped, respectively.

In the STARTED state, i.e. just after successful completion of a call to startFc, a component can emit or accept operation invocations, which are guaranteed to execute "normally". Note that this does not prevent the unbindFc and removeFcSubComponent operations to throw the IllegalLifeCycleException if they are invoked while the component is in this state (in order to prevent a component from being reconfigured while it is in an unstable state).

In the STOPPED state, i.e. just after successful completion of a call to stopFc, a component cannot emit operation invocations, and can accept operation invocations only through control interfaces. *The result of operation invocations to the functional interfaces of a stopped component is undefined.* It may be a normal result, an exception, a suspension of the invocation until the component is restarted, or anything else.

The LifeCycleController interface corresponds to a minimal life cycle automaton, whose transitions are represented in the following table:

	STOPPED	STARTED
startFc	STARTED	STARTED
stopFc	STOPPED	STOPPED

However, some components may require very different life cycles. Of course, completely arbitrary life cycles can be specified by providing completely new interfaces, distinct from the LifeCycleController interface. More commonly, life cycles can be adapted from the basic one by extending the LifeCycleController interface to introduce new states and transitions or even to change the transitions of the basic life cycle. In this case, it is a requirement of this specification that the semantics associated to the STARTED and STOPPED states should be preserved.

The org.objectweb.fractal.api.control.IllegalLifeCycleException exception may be thrown when a requested transition, in a life cycle automaton, is not valid.

A component interface implementing LifeCycleController must be named lifecycle-controller.

5. Instantiation

The frameworks presented in the previous sections allows one to use, introspect, configure and reconfigure *existing* components. In order to be useful, they must be completed with a framework to create *new* components. This section defines such a framework, based on factories.

5.1. Factories

In the instantiation framework specified in this section, components are created by other components called component *factories*. The Fractal model distinguishes between generic component factories, which can create several kinds of components, and standard component factories, which can create only one kind of components, all with the same component type. Generic and standard component factories can provide the GenericFactory interface and the Factory interfaces, respectively (see Figure 12 - note that, in accordance with the rule defined in section 4.1, both interfaces are functional interfaces, and not control interfaces).

```
package org.objectweb.fractal.api.factory;

interface GenericFactory {
  Component newFcInstance (Type t, any controllerDesc, any contentDesc)
    throws InstantiationException;
}

interface Factory {
  Type getFcInstanceType ();
  any getFcControllerDesc ();
  any getFcContentDesc ();
  Component newFcInstance () throws InstantiationException;
}
```

Figure 12: Instantiation API

The GenericFactory interface provides only one operation named newFcInstance. This operation takes as parameter the type of the component to be created, a descriptor of its controller part, and a descriptor of its content part. This operation creates a component corresponding to the given description, and returns its Component interface.

The Factory interface also provides a newFcInstance operation, but this operation does not take any parameter, which reflects the fact that all the components created by this operation have the same type, and the same controller and content descriptors. These information can be retrieved with the three other operations of this interface, named getFcInstanceType, getFcControllerDesc and getFcContentDesc.

In both interfaces, the component type must describe only the functional interfaces of the components to be created. The control interfaces of the components to be created must indeed be specified in the controller descriptors. The exact semantics of the controller and content descriptors, in both interfaces, is however left unspecified in this version of the Fractal component model specification.

Note that, in both interfaces, the newFcInstance operation does not necessarily create a new component instance each time it is invoked. It can also, for example, always return the same instance (this is the singleton pattern). The components created by a factory must be created in the same address space as the factory component. But the exact location of the created components, in this address space, is voluntarily *not* specified. In particular, it is not ensured that the components created by a factory are automatically added to the parent component(s) of the factory component.

The org.objectweb.fractal.api.factory.InstantiationException exception must be thrown when a component cannot be created, in the newFcInstance operations of the Factory and GenericFactory interfaces.

A component interface implementing GenericFactory must be named generic-factory. A component interface implementing Factory must be named factory.

5.2. Templates

A *template* is a special kind of standard factory component that creates components that are quasi "isomorphic" to itself. More precisely, the components created by a template component *must* have the same functional client and server interfaces as the template component (except for the Factory interface, which is provided by the template, but not necessarily by its instances), but can have arbitrary control interfaces. The components created by a template component also have the same attributes as the template. A template component may contain several sub template components, bound together through bindings. The components created by such a template component are components that contain as many sub components as sub templates in the template, bound together as the sub templates are bound in the template (see Figure 13). If some sub templates are shared, the corresponding sub components in the components created by the template will also be shared.

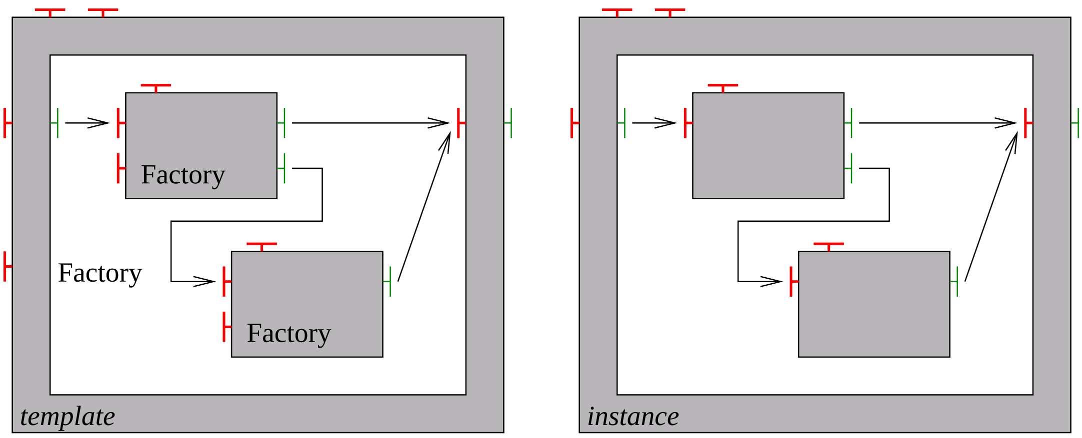


Figure 13: A sample template component and a component created from it

If a generic factory component is able to create template components, then it must be possible to create a template component with a operation invocation, on this generic factory, of the form newFcInstance(type, templateControllerDesc, {controllerDesc, contentDesc}), where type describes the functional client and server interfaces of the components that the template will create, templateControllerDesc is the descriptor of the controller part of the template component to be created, and controllerDesc and contentDesc are the descriptors of the controller and content parts of the components that the template will create (the brackets denote an array).

Template components are useful in only one case, namely when several identical components must be created from a textual representation, such as an Architecture Description Language definition. In this case, instead of parsing the textual representation each time an instance must be created, it can be more efficient to parse the text file(s) and to create a corresponding template only once, and then to instantiate the template each time an instance is needed. In all other cases, using templates is equivalent, but generally less efficient, than not using them.

5.3. Bootstrap

According to the above framework, components are created from component factories. But how are created component factories? They can be created from other component factories, but this leads to an infinite recursion. In order to stop it, a bootstrap component factory, which does not need to be created explicitly, and which is accessible from a “well-known” name, is necessary. This bootstrap component factory must be able to create several kinds of components, including component factories. In other words, it must provide the GenericFactory interface.

Note

In Java, this bootstrap component must be accessible from the getBootstrapComponent static method, defined in the org.objectweb.fractal.api.Fractal class. This method must not take any parameter, and must return the Component interface of the bootstrap component.

6. Typing

This section defines a simple type system for components and component interfaces. This type system reflects the main characteristics of component interfaces, introduced in section 3, i.e. their name, their language type, and their *role* (client or server). It also introduces two new characteristics named *contingency* and *cardinality*.

6.1. Contingency and cardinality

The *contingency* of an interface indicates if the functionality corresponding to this interface is guaranteed to be available or not, while the component is running:

- the operations of a *mandatory* interface are guaranteed to be available when the component is running. This semantic is obvious for a server interface. For a client interface, which does not have a functionality of its own, it means that the interface *must* be bound, and that it must be bound to a mandatory interface. As a consequence, a component with mandatory client interfaces cannot be started until all these interfaces are bound to other mandatory interfaces.
- the operations of an *optional* interface are *not* guaranteed to be available. This can happen, for a server interface, when the complementary internal interface is not bound to a sub component. This can also happen, for a client interface, when this interface is not bound.

The *cardinality* of an interface **type** T indicates how many interfaces of type T a given component may have:

- the *singleton* cardinality means that a given component must have exactly one interface of type T.
- the *collection* cardinality means that a given component may have an arbitrary number of interfaces of type T. All these interfaces must have a name that *begins* with the name specified in T (see next section). Since there is a priori an infinite number of such interfaces, these interfaces cannot all be created at the same time: they must be created lazily, during invocations of the getFcInterface and bindFc operations. For example, if the name specified in T is listener, then an invocation to getFcInterface("listener11") or to bindFc("listener11", s) will create an interface named listener11, if it does not already exist. This interface may be removed automatically when it is no longer used by any binding.

Mandatory and optional interfaces are useful for components that absolutely require other components to work, and which may also use other components, if they are present. For example, a parser component absolutely needs a lexer component, but can work with or without a logger component. Collection interfaces are useful for components with a variable number of required components of the same type, such as a menu component and its associated menu item components, a model component and its listener components (in the MVC model), and so on.

6.2. Type system

In the type system specified here, a component type is just a set of component interface types. A component type is represented by the ComponentType interface (see Figure 14). This interface defines a getFcInterfaceTypes operation, which returns the set of component interface types in this component type, as an array. It also defines a getFcInterfaceType operation, which returns the component interface type whose name is given as parameter (this operation must throw the NoSuchInterfaceException if the requested interface type does not exist).

A component interface type is represented by the InterfaceType interface. Such a type is made of a name, a signature, a role, a contingency and a cardinality. The name is the name of component interfaces of this type. The signature is the name of the language interface type that is implemented by component interfaces of this type (for a client interface, an empty signature means that this client interface can be connected to any server interface). The role indicates if component interfaces of this type are client or server interfaces. The contingency indicates if the functionality of interfaces of this type is guaranteed to be available or not. Finally, the cardinality indicates how many interfaces of this type a component may have.

Component and component interface types can be created by using a type factory, represented by the TypeFactory interface. Indeed this interface provides two operations to create component interface types and component types. A component interface implementing TypeFactory must be named type-factory.

```
package org.objectweb.fractal.api.type;

interface ComponentType extends Type {
  InterfaceType[] getFcInterfaceTypes ();
  InterfaceType getFcInterfaceType (string itfName) throws NoSuchInterfaceException;
}

interface InterfaceType extends Type {
  string getFcItfName ();
  string getFcItfSignature ();
  boolean isFcClientItf ();
  boolean isFcOptionalItf ();
  boolean isFcCollectionItf ();
}

interface TypeFactory {
  InterfaceType createFcItfType (string name, string signature, boolean isClient,
    boolean isOptional, boolean isCollection) throws InstantiationException;
  ComponentType createFcType (InterfaceType[] itfTypes) throws InstantiationException;
}
```

Figure 14: Typing API

A component of type T must have as many external interfaces as described in T (and, in particular, in the interface cardinalities), and all these interfaces must have the name, language type and role described in the corresponding component interface type. Likewise, if this component also exposes its content, and in particular its internal interfaces, then it must also have, *at most*, as many internal *functional* interfaces as described in T, and each of these interfaces must have the name, language type and role described in the corresponding component interface type. This implies that each internal functional interface has a *complementary* external interface of the same name, signature, contingency and cardinality, and of opposite role (but the converse is not necessarily true). Note that this property is ensured by the type system specified in this section: in the general case, nothing more than what is explicitely stated in section 4.1 is ensured (and so an internal interface may not have a complementary external interface).

Note that if, in general, the number of interfaces of a Fractal component may change during its life time, the number of interfaces of a Fractal component *that uses the type system presented here* cannot change during its lifetime (except for interface collections). Indeed the ComponentType and InterfaceType interfaces do not offer any operations to modify an existing type, and the other interfaces specified in this document do not offer a operation to change the type of a component or of an interface. But a Fractal component may perfectly provide a setFcType operation, if needed, since the Fractal model is extensible.

6.3. Sub typing relation

This section defines a sub typing relation for component and component interface types, based on *substitutability*. This relation provides a sufficient (but not necessary) condition such that if a component type T_1 is a sub type of T_2, then a component of type T_1 can replace a component of type T_2 in any environment, this environment (other components and bindings) being left unchanged, and *both components being seen as black boxes*.

An interface type I_1 is a sub type of a *server* interface type I_2 if the following conditions are satisfied: I_1 has the same name and the same role as I_2; the language interface corresponding to I_1 is a sub interface of the language interface corresponding to I_2; if the contingency of I_2 is mandatory, then the contingency of I_1 is mandatory too; if the cardinality of I_2 is collection, then the cardinality of I_1 is collection too.

An interface type I_1 is a sub type of a *client* interface type I_2 if the following conditions are satisfied: I_1 has the same name and the same role as I_2; the language interface corresponding to I_1 is a super interface of the language interface corresponding to I_2; if the contingency of I_2 is optional, then the contingency of I_1 is optional too; if the cardinality of I_2 is collection, then the cardinality of I_1 is collection too.

A component type T_1 is a sub type of a component type T_2 if and only if each client interface type defined in T_1 is a sub type of an interface type defined in T_2, and each server interface type defined in T_2 is a super type of an interface type defined in T_1.

7. Options

As said in section 1.2, in the Fractal component model, everything is optional. For example, a Fractal component may provide or not the Component interface, it may support or not the Interface interface, it may provide or not the control interfaces defined in section 4, it may use or not the type system defined in section 6, and so on.

In addition, a Fractal component may provide or use new or alternative control interfaces, type systems, or even component semantics. For example, a Fractal component may provide a new ConcurrencyController interface to control concurrent accesses to the component. It may also provide an alternative BindingController interface, named for example InternalBindingController, to control the bindings between sub components directly from the enclosing component. It can also use an empty type system, with a unique type, sub type of itself, used for all components and component interfaces. A Fractal component may even define a new semantic for the communication between its sub components: instead of specifying that operation invocations follow bindings, as defined in section 4.1, it can for example specify that operation invocations are broadcasted to all the sub components, in order to model an asynchronous, reactive "space". Bindings are then useless (another possiblity is to define parallel components, where all the sub components have the same type as the enclosing component, and where each operation invocation received on this component is executed in parallel by all its sub components). A Fractal component may also refine the internal component structure defined in section 4.1, by specifying that the component's controller can, like the component's content, contain sub components. Such a Fractal component can then provide new control interfaces to introspect and reconfigure the sub components of its controller part.

The advantage of this extreme modularity and extensibility is that the Fractal component model can be applied to many situations. The drawback is that two arbitrary Fractal components will generally not be able to work together, because they will generally use very different, and potentially incompatible, options or extensions of the Fractal model. In order to reduce this problem, this section defines some set of options, and gives them a symbolic name called a conformance *level*. The goal is to be able to say, or even certify, that a given Fractal application or tool is conform to the Fractal model of level X. It will then be easy to know which Fractal applications and tools can work together, by comparing their conformance level to the Fractal model.

7.1. Conformance levels

This specification defines the following conformance levels (new conformance levels can of course be defined as needed):

- level 0: at this level nothing is mandatory. Fractal components are like simple objects. A Java object, a Java Bean, or an Enterprise Java Bean, for example, are conform to the Fractal component model of level 0.
  - level 0.1: same as level 0, with the additional requirements that all components with configurable attributes must provide the AttributeController interface, that all components with client interfaces must provide the BindingController interface, that all

components that expose their content must provide the ContentController interface, and that all components that expose their life cycle must provide the LifeCycleController interface. Of course, these requirements do not prevent components from providing additional control interfaces, including extensions and alternatives of the previous interfaces.

- level 1: same as level 0, with the additional requirement that all components must provide, at least, the Component interface.
  - level 1.1: same as level 1, with the same additional requirements as for level 0.1, concerning the control interfaces.
- level 2: same as level 1, with the additional requirement that all component interface references must be castable to Interface.
  - level 2.1: same as level 2, with the same additional requirements as for levels 0.1 and 1.1, concerning the control interfaces.
- level 3: same as level 2, with the additional requirement that all the components must use (an extension of) the type system defined in section 6.
  - level 3.1: same as level 3, with the same additional requirements as for levels 0.1, 1.1 and 2.1, concerning the control interfaces.
  - level 3.2: same as level 3.1, with the additional requirement that a bootstrap component must be accessible from a "well-known" name. This bootstrap component must provide a GenericFactory and a TypeFactory interface. Moreover, the GenericFactory interface must be able to create components with any control interfaces in the set of control interfaces defined in section 4 (and, in particular, primitive and composite components). Finally, this interface must also be able to create (3.2 level) primitive components encapsulating 0.1 level components (see below for more details).
  - level 3.3: same as level 3.2, with the additional requirement that the GenericFactory interface of the bootstrap component must be able to create primitive and composite template components.

These conformance levels can summarized as shown in Figure 15, where C, I, CT, IT, AC, BC, CC, LC, F and T represent the component, interface, component type, interface type, attribute controller, binding controller, life cycle controller, factory and "template" interfaces, respectively, and where an x denotes a mandatory feature.

Note that a level 3 component is also a level 2, 1 or 0 component, a level 2 component is also a level 1 or 0 component, a level 3.3 component is also a level 3.2, 3.1 or 3 component, but a level 3, 2 or 1 component is *not* a level 0.1, 1.1 or 2.1 component. More generally, if l_1 is greater than l_2 in alphabetical order, a level l_2 component is *not* necessarily also a level l_1 component (this desirable rule cannot always be respected, because the alphabetical order is a total order, while the set inclusion order is only a partial order).

	C	I	CT, IT	AC, BC, CC, LC	F	T
0						
0.1				x		
1	x					
1.1	x			x		
2	x	x				
2.1	x	x		x		
3	x	x	x			
3.1	x	x	x	x		
3.2	x	x	x	x	x	
3.3	x	x	x	x	x	x

Figure 15: Conformance levels to the Fractal component model

Encapsulated components

As specified above, at the 3.2 level, the bootstrap generic factory must be able to create 3.2 primitive components that encapsulate 0.1 level components. These encapsulating components (which do not have a ContentController interface, and therefore are *not* composite components) must "delegate" all the operation invocations they receive on their functional and control interfaces to their encapsulated component, if the encapsulated component has a corresponding interface. For example, if the bindFc or startFc operation is invoked on an encapsulating component, this component must in turn invoke this operation on its encapsulated component, if it provides the BindingController or LifeCycleController interface. The encapsulating component can of course do some pre and post computations before and after calling its encapsulated component.

If the encapsulated component provides a BindingController interface, then the encapsulating component must invoke, during its initialization, the bindFc operation of this interface with, as arguments, the component name and the reference of its Component interface, so that the encapsulated component can get a reference to the Component interface of its encapsulating component.

7.2. Extensions

The above conformance levels may not be sufficient to fully compare two Fractal systems. In particular, for Fractal systems dedicated to a specific language, such as C or Java, the language is as important as the conformance level. And, even for Fractal systems based on the same language, other "details" (such as, in Java, the class loading policy - one class loader per component vs a single class loader for all the components), unspecified here, may cause incompatibilities. However, if needed, these other features can be captured in new conformance levels, such as $C.x.y$ or $J.x.y$, for C and Java respectively.

8. Example

This section shows how a 3.3 level, Java Fractal platform can be used, in order to illustrate how the APIs defined in this specification can be used to create, assemble and reconfigure component configurations.

The example used throughout this section is a very simple application made of two primitive components inside a composite component (see Figure 16). The first primitive component is a "server" component that provides an interface s of type S. The other primitive component is a "client" component, bound to the previous server interface.

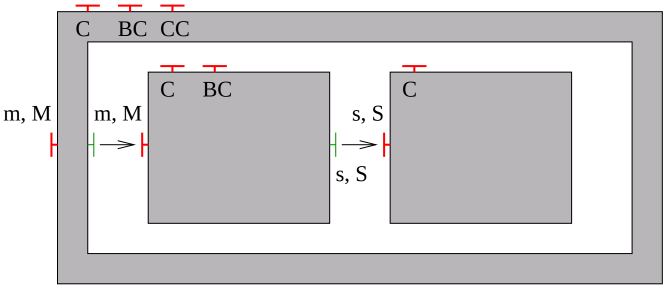


Figure 16: A sample component based application

8.1. Instantiation

The above components can be instantiated as follows. The first step is to create the component and component interface types. In order to do this, we get a reference to the bootstrap component, and then to its TypeFactory interface:

```
Component boot = Fractal.getBootstrapComponent();
TypeFactory tf = (TypeFactory)boot.getFcInterface("type-factory");
```

We can now create the types of the root, client and server components as follows:

```
ComponentType rType = tf.createFcType(new InterfaceType[] {
  tf.createFcItfType("m", "M", false, false, false)
});
```

```
ComponentType cType = tf.createFcType(new InterfaceType[] {
  tf.createFcItfType("m", "M", false, false, false),
  tf.createFcItfType("s", "S", true, false, false)
});
```

```
ComponentType sType = tf.createFcType(new InterfaceType[] {
  tf.createFcItfType("s", "S", false, false, false)
});
```

We could now create the components directly, but we will use intermediate template components here, in order to show how they can be used. These component templates can be created as follows:

```
GenericFactory gf = (GenericFactory)boot.getFcInterface("generic-factory");
```

```
Component rTmpl = gf.newFcInstance(
  rType, "compositeTemplate", new Object[] {"composite", null});
```

```
Component cTmpl = gf.newFcInstance(
  cType, "template", new Object[] {"primitive", "CImpl"});
```

```
Component sTmpl = gf.newFcInstance(
  sType, "template", new Object[] {"primitive", "SImpl"});
```

Here the template (resp. compositeTemplate) descriptor is supposed to describe components with a BindingController interface (resp. with a BindingController and a ContentController interfaces). The primitive and composite descriptors are supposed to describe similar components, but with an additional LifeCycleController interface. Finally, CImpl and SImpl are the names of the Java classes of the 0.1 level Fractal components that will be *encapsulated* in the client and server components (see end of section 7.1). The CImpl class, for example, has the following form:

```
public class CImpl implements M, BindingController {
  private S s;
  public String[] listFc () { return new String[] { "s" }; }
  public Object lookupFc (String name) {
    if (name.equals("s")) return s;
    return null;
  }
  public Object bindFc (String name, Object itf) {
    if (name.equals("s")) s = (S)itf;
  }
  public Object unbindFc (String name) {
    if (name.equals("s")) s = null;
  }
  // ...
}
```

We can then either instantiate each template one by one, put the resulting primitive components inside the composite component, connect all these components, and finally start them. But we can also put the primitive templates inside the composite template, connect these templates together, and then instantiate the whole application by just instantiating the composite template component. This is what we do here.

We begin by putting the primitive template components inside the composite one (here we assume a strong semantic for the addFcSubComponent method, i.e. we assume that $C' = C \cup \{c\}$ - see section 4.4):

```
ContentController cc = (ContentController)rTmpl.getFcInterface("content-controller");
cc.addFcSubComponent(cTmpl);
cc.addFcSubComponent(sTmpl);
```

We then bind the internal client interface m of the composite template to the server interface m of the client template:

```
((BindingController)rTmpl.getFcInterface("binding-controller"))
  .bindFc("m", cTmpl.getFcInterface("m"));
```

Finally, we bind the client interface s of the client template to the server interface s of the server template:

```
((BindingController)cTmpl.getFcInterface("binding-controller"))
  .bindFc("s", sTmpl.getFcInterface("s"));
```

At this stage the template components are like the components depicted in Figure 16, with just an additional Factory interface. Now that the template components have been created and bound to each other, the application components can be instantiated and bound to each other automatically, by just calling the newFcInstance method on the root template component:

```
Component r = ((Factory)rTmpl.getFcInterface("factory")).newFcInstance();
```

The result is depicted in Figure 17. As can be seen, the 0.1 level components CImpl and SImpl have been encapsulated in 3.2 level components, which provide them component and interface introspection functions.

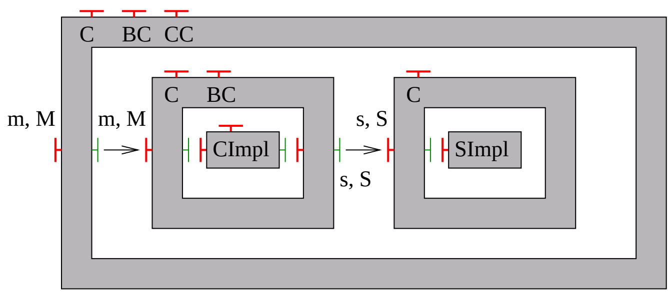


Figure 17: Result of the instantiation of the application depicted in Figure 16

All the application components can now be started automatically by just calling the startFc method on the root application component (here we assume a stronger semantic than the default one for the startFc method, i.e. we assume it to be recursive - see section 4.5):

```
((LifeCycleController)r.getFcInterface("lifecycle-controller")).startFc();
```

8.2. Reconfiguration

Let us suppose we want to dynamically change the server component. In order to do this, we need to unbind the client component, remove the server component, create a new server component, add the server component in the composite component, and finally bind the client component to the new server. But the binding and component removals cannot be done while the client and the composite component, respectively, are not stopped. So we must first stop these components (here again we assume this method to be recursive; we also assume that it does not change the states of the components, and that method calls to functional interfaces while the components are stopped are only suspended until the components are restarted):

```
((LifeCycleController)r.getFcInterface("lifecycle-controller")).stopFc();
```

We then retrieve the references of the client and server components (more precisely of the 3.2 level components that encapsulate the 0.1 level components CImpl and SImpl):

```
Component c = ((Interface)((BindingController)r.
  getFcInterface("binding-controller")).lookupFc("m")).getFcItfOwner();
Component s = ((Interface)((BindingController)c.
  getFcInterface("binding-controller")).lookupFc("s")).getFcItfOwner();
```

We can now unbind the client and server components, and remove the server component from the composite component (we assume a strong semantic for removeFcSubComponent):

```
((BindingController)c.getFcInterface("binding-controller")).unbindFc("s");
((ContentController)r.getFcInterface("content-controller")).removeFcSubComponent(s);
```

We can now create the new server component, i.e. a new 3.2 level component encapsulating a new 0.1 level component. Instead of using a template component for doing that, as in the previous section, we use here the bootstrap generic factory directly:

```
Component newS = gf.newFcInstance(sType, "primitive", "NewSImpl");
```

We can now add this new component in the composite component, bind it to the client component, and finally resume the application's execution (we make the same semantic hypotheses as in the previous section for the addFcSubComponent and startFc methods):

```
((ContentController)r.getFcInterface("content-controller")).addFcSubComponent(newS);
((BindingController)c.getFcInterface("binding-controller")).bindFc("s", newS);
((LifeCycleController)r.getFcInterface("lifecycle-controller")).startFc();
```

A. Fractal APIs

This section defines the Fractal API in Java, C and OMG IDL. These definitions can be used as "standard" definitions to provide interoperability between Java components only, between C components only, and between any components (respectively). They are the result of straightforward transformations of the pseudo IDL interface definitions from section 2 to 6.

Java API

```
package org.objectweb.naming;
public interface Name {
  NamingContext getNamingContext ();
  byte[] encode () throws NamingException;
}
public interface NamingContext {
  Name export (Object o, Object hints) throws NamingException;
  Name decode (byte[] b) throws NamingException;
}
public interface Binder extends NamingContext {
  Object bind (Name n, Object hints) throws NamingException;
}
public class NamingException extends Exception {
  public NamingException (String msg) { super(msg); }
}
package org.objectweb.fractal.api;
import org.objectweb.fractal.api.factory.InstantiationException;
public interface Component {
  Type getFcType ();
  Object[] getFcInterfaces ();
  Object getFcInterface (String interfaceName) throws NoSuchInterfaceException;
}
public interface Interface {
  Component getFcItfOwner ();
  String getFcItfName ();
  Type getFcItfType ();
  boolean isFcInternalItf ();
}
public interface Type {
  boolean isFcSubTypeOf (Type type);
}
public class Fractal {
  public static Component getBootstrapComponent () throws InstantiationException;
}
public class NoSuchInterfaceException extends Exception {
  public NoSuchInterfaceException (String itfName) { super(itfName); }
}
package org.objectweb.fractal.api.control;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.NoSuchInterfaceException;
public interface AttributeController { }
public interface BindingController {
  String[] listFc ();
  Object lookupFc (String clientItfName) throws NoSuchInterfaceException;
  void bindFc (String clientItfName, Object serverItf) throws
    NoSuchInterfaceException, IllegalBindingException, IllegalLifeCycleException;
```

```
  void unbindFc (String clientItfName) throws
    NoSuchInterfaceException, IllegalBindingException, IllegalLifeCycleException;
}
public interface ContentController {
  Object[] getFcInternalInterfaces ();
  Object getFcInternalInterface (String interfaceName) throws NoSuchInterfaceException;
  Component[] getFcSubComponents ();
  void addFcSubComponent (Component subComponent)
    throws IllegalContentException, IllegalLifeCycleException;
  void removeFcSubComponent (Component subComponent)
    throws IllegalContentException, IllegalLifeCycleException;
}
public interface SuperController {
  Component[] getFcSuperComponents ();
}
public interface LifeCycleController {
  String getFcState ();
  void startFc () throws IllegalLifeCycleException;
  void stopFc () throws IllegalLifeCycleException;
}
public interface NameController {
  String getFcName ();
  void setFcName (String name);
}
public class IllegalBindingException extends Exception {
  public IllegalBindingException (String msg) { super(msg); }
}
public class IllegalContentException extends Exception {
  public IllegalContentException (String msg) { super(msg); }
}
public class IllegalLifeCycleException extends Exception {
  public IllegalLifeCycleException (String msg) { super(msg); }
}
package org.objectweb.fractal.api.factory;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.Type;
public interface Factory {
  Type getFcInstanceType ();
  Object getFcControllerDesc ();
  Object getFcContentDesc ();
  Component newFcInstance () throws InstantiationException;
}
public interface GenericFactory {
  Component newFcInstance (Type type, Object controllerDesc, Object contentDesc)
    throws InstantiationException;
}
public class InstantiationException extends Exception {
  public InstantiationException (String msg) { super(msg); }
}
package org.objectweb.fractal.api.type;
import org.objectweb.fractal.api.NoSuchInterfaceException;
public interface ComponentType extends org.objectweb.fractal.api.Type {
  InterfaceType[] getFcInterfaceTypes ();
  InterfaceType getFcInterfaceType (String name) throws NoSuchInterfaceException;
}
public interface InterfaceType extends org.objectweb.fractal.api.Type {
  String getFcItfName ();
```

```
  String getFcItfSignature ();
  boolean isFcClientItf ();
  boolean isFcOptionalItf ();
  boolean isFcCollectionItf ();
}
public interface TypeFactory {
  InterfaceType createFcItfType (
    String name, String signature, boolean isClient, boolean isOptional, boolean isCollection)
      throws org.objectweb.fractal.api.factory.InstantiationException;
  ComponentType createFcType (InterfaceType[] interfaceTypes)
    throws org.objectweb.fractal.api.factory.InstantiationException;
}
```

C API

```
typedef unsigned char jboolean;
typedef unsigned short jchar;
typedef signed char jbyte;
typedef signed short jshort;
typedef signed int jint;
typedef signed long long jlong;
typedef float jfloat;
typedef double jdouble;
struct Morg_objectweb_naming_Name {
  Rorg_objectweb_naming_NamingContext* (*getNamingContext)(void *_this);
  jbyte* (*encode)(void *_this);
};
struct Morg_objectweb_naming_NamingContext {
  Rorg_objectweb_naming_Name* (*export)(void *_this, void* o, void* hints);
  Rorg_objectweb_naming_Name* (*decode)(void *_this, jbyte* b);
};
struct Morg_objectweb_naming_Binder {
  Rorg_objectweb_naming_Name* (*export)(void *_this, void* o, void* hints);
  Rorg_objectweb_naming_Name* (*decode)(void *_this, jbyte* b);
  void* (*bind)(void *_this, Rorg_objectweb_naming_Name* n, void* hints);
};
struct Morg_objectweb_fractal_api_Component {
  Rorg_objectweb_fractal_api_Type* (*getFcType)(void *_this);
  void** (*getFcInterfaces)(void *_this);
  void* (*getFcInterface)(void *_this, char* interfaceName);
};
struct Morg_objectweb_fractal_api_Interface {
  Rorg_objectweb_fractal_api_Component* (*getFcItfOwner)(void *_this);
  char* (*getFcItfName)(void *_this);
  Rorg_objectweb_fractal_api_Type* (*getFcItfType)(void *_this);
  jboolean (*isFcInternalItf)(void *_this);
};
struct Morg_objectweb_fractal_api_Type {
  jboolean (*isFcSubTypeOf)(void *_this, Rorg_objectweb_fractal_api_Type* type);
};
struct Morg_objectweb_fractal_api_control_AttributeController { };
struct Morg_objectweb_fractal_api_control_BindingController {
  char** (*listFc)(void *_this);
  void* (*lookupFc)(void *_this, char* clientItfName);
  void (*bindFc)(void *_this, char* clientItfName, void* serverItf);
  void (*unbindFc)(void *_this, char* clientItfName);
};
```

```
struct Morg_objectweb_fractal_api_control_ContentController {
  void** (*getFcInternalInterfaces)(void *_this);
  void* (*getFcInternalInterface)(void *_this, char* interfaceName);
  Rorg_objectweb_fractal_api_Component** (*getFcSubComponents)(void *_this);
  void (*addFcSubComponent)(
    void *_this, Rorg_objectweb_fractal_api_Component* subComponent);
  void (*removeFcSubComponent)(
    void *_this, Rorg_objectweb_fractal_api_Component* subComponent);
};
struct Morg_objectweb_fractal_api_control_SuperController {
  Rorg_objectweb_fractal_api_Component** (*getFcSuperComponents)(void *_this);
};
struct Morg_objectweb_fractal_api_control_LifeCycleController {
  char* (*getFcState)(void *_this);
  void (*startFc)(void *_this);
  void (*stopFc)(void *_this);
};
struct Morg_objectweb_fractal_api_control_NameController {
  char* (*getFcName)(void *_this);
  void (*setFcName)(void *_this, char* name);
};
struct Morg_objectweb_fractal_api_factory_Factory {
  Rorg_objectweb_fractal_api_Type* (*getFcInstanceType)(void *_this);
  void* (*getFcControllerDesc)(void *_this);
  void* (*getFcContentDesc)(void *_this);
  Rorg_objectweb_fractal_api_Component* (*newFcInstance)(void *_this);
};
struct Morg_objectweb_fractal_api_factory_GenericFactory {
  Rorg_objectweb_fractal_api_Component* (*newFcInstance)(
    void *_this,  Rorg_objectweb_fractal_api_Type* type, void* ctrlDesc, void* cntntDesc);
};
struct Morg_objectweb_fractal_api_type_ComponentType {
  jboolean (*isFcSubTypeOf)(void *_this, Rorg_objectweb_fractal_api_Type* type);
  Rorg_objectweb_fractal_api_type_InterfaceType** (*getFcInterfaceTypes)(void *_this);
  Rorg_objectweb_fractal_api_type_InterfaceType* (*getFcInterfaceType)(void *_this, char* name);
};
struct Morg_objectweb_fractal_api_type_InterfaceType {
  jboolean (*isFcSubTypeOf)(void *_this, Rorg_objectweb_fractal_api_Type* type);
  char* (*getFcItfName)(void *_this);
  char* (*getFcItfSignature)(void *_this);
  jboolean (*isFcClientItf)(void *_this);
  jboolean (*isFcOptionalItf)(void *_this);
  jboolean (*isFcCollectionItf)(void *_this);
};
struct Morg_objectweb_fractal_api_type_TypeFactory {
  Rorg_objectweb_fractal_api_type_InterfaceType* (*createFcItfType)(
    void *_this, char* name, char* signature,
    jboolean isClient, jboolean isOptional, jboolean isCollection);
  Rorg_objectweb_fractal_api_type_ComponentType* (*createFcType)(
    void *_this, Rorg_objectweb_fractal_api_type_InterfaceType** interfaceTypes);
};
// where Rxyz types are defined by:
// typedef struct {
//   struct Mxyz *meth;
//   void *selfdata;
// } Rxyz;
```

OMG IDL API

```
typedef sequence<Object> ObjectArray;
typedef sequence<string> stringArray;
typedef sequence<octet> octetArray;
module org_objectweb_naming {
  exception NamingException { };
  interface NamingContext;
  interface Name {
    NamingContext getNamingContext ();
    octetArray encode () raises(NamingException);
  };
  interface NamingContext {
    Name export (in Object o, in Object hints) raises(NamingException);
    Name decode (in octetArray b) raises(NamingException);
  };
  interface Binder : NamingContext {
    Object bind (in Name n, in Object hints) raises(NamingException);
  };
};
module org_objectweb_fractal_api {
  exception NoSuchInterfaceException { };
  interface Type {
    boolean isFcSubTypeOf (in Type type);
  };
  interface Component {
    Type getFcType ();
    ObjectArray getFcInterfaces ();
    Object getFcInterface (in string interfaceName) raises(NoSuchInterfaceException);
  };
  typedef sequence<Component> ComponentArray;
  interface Interface {
    Component getFcItfOwner ();
    string getFcItfName ();
    Type getFcItfType ();
    boolean isFcInternalItf ();
  };
};
module org_objectweb_fractal_api_control {
  exception IllegalBindingException { };
  exception IllegalContentException { };
  exception IllegalLifeCycleException { };
  interface AttributeController { };
  interface BindingController {
    stringArray listFc ();
    Object lookupFc (in string clientItfName)
      raises(org_objectweb_fractal_api::NoSuchInterfaceException);
    void bindFc (in string clientItfName, in Object serverItf) raises(IllegalBindingException,
      IllegalLifeCycleException, org_objectweb_fractal_api::NoSuchInterfaceException);
    void unbindFc (in string clientItfName) raises(IllegalBindingException,
      IllegalLifeCycleException, org_objectweb_fractal_api::NoSuchInterfaceException);
  };
  interface ContentController {
    ObjectArray getFcInternalInterfaces ();
    Object getFcInternalInterface (in string interfaceName)
      raises(org_objectweb_fractal_api::NoSuchInterfaceException);
    org_objectweb_fractal_api::ComponentArray getFcSubComponents ();
    void addFcSubComponent (in org_objectweb_fractal_api::Component subComponent)
```

```
      raises(IllegalContentException, IllegalLifeCycleException);
    void removeFcSubComponent (in org_objectweb_fractal_api::Component subComponent)
      raises(IllegalContentException, IllegalLifeCycleException);
  };
  interface SuperController {
    org_objectweb_fractal_api::ComponentArray getFcSuperComponents ();
  };
  interface LifeCycleController {
    string getFcState ();
    void startFc () raises(IllegalLifeCycleException);
    void stopFc () raises(IllegalLifeCycleException);
  };
  interface NameController {
    string getFcName ();
    void setFcName (in string name);
  };
};
module org_objectweb_fractal_api_factory {
  exception InstantiationException { };
  interface GenericFactory {
    org_objectweb_fractal_api::Component newFcInstance (
      in org_objectweb_fractal_api::Type type, in Object controllerDesc, in Object contentDesc)
        raises(InstantiationException);
  };
  interface Factory {
    org_objectweb_fractal_api::Type getFcInstanceType ();
    Object getFcControllerDesc ();
    Object getFcContentDesc ();
    org_objectweb_fractal_api::Component newFcInstance () raises(InstantiationException);
  };
};
module org_objectweb_fractal_api_type {
  interface InterfaceType : org_objectweb_fractal_api::Type {
    string getFcItfName ();
    string getFcItfSignature ();
    boolean isFcClientItf ();
    boolean isFcOptionalItf ();
    boolean isFcCollectionItf ();
  };
  typedef sequence<InterfaceType> InterfaceTypeArray;
  interface ComponentType : org_objectweb_fractal_api::Type {
    InterfaceTypeArray getFcInterfaceTypes ();
    InterfaceType getFcInterfaceType (in string name)
      raises(org_objectweb_fractal_api::NoSuchInterfaceException);
  };
  interface TypeFactory {
    InterfaceType createFcItfType (
      in string name, in string signature,
      in boolean isClient, in boolean isOptional, in boolean isCollection)
        raises(org_objectweb_fractal_api_factory::InstantiationException);
    ComponentType createFcType (in InterfaceTypeArray interfaceTypes)
      raises(org_objectweb_fractal_api_factory::InstantiationException);
  };
};
```

B. Glossary

This section defines the core concepts of the Fractal model.

binder: a naming context that can also give access (reference) to the interfaces designated by the names it manages.

binding: a primitive, local communication path between a client and a server interface. More complex "bindings" are made of bindings and of binding components.

concept: an abstract representation composed of the properties common to a set of concrete representations of directly observable entities.

cardinality: a property of an interface type that indicates how many interfaces of this type a given component may have. The cardinality is either singleton or collection.

component: a runtime entity exhibiting a recursive structure and reflexive capabilities. A component is composed of a controller and a content. A component has well defined access points called interfaces, and provides more or less introspection and control capabilities (intercession) to other components.

- **base component**: a component that does not have any control interface.
- **primitive component**: a component with some control interfaces, but that does not expose its content.
- **composite component**: a component that exposes its content.
- **sub component**: a component that is contained in another component.
- **super component**: relatively to a (sub) component: a component that contains this (sub) component. Due to component sharing, a component may have several super components.
- **shared component**: a component that is contained in several super components.
- **binding component**: a component dedicated to the communication between other components. Similar to a connector in other component models.

conformance level : a symbolic name that designates a set of options or extensions of the Fractal component model. A Fractal system is conform to a given conformance level if it supports all the options designated by this level.

content: one of the two parts of a component, the other one being its controller. A content is an abstract entity controlled by a controller. The content of a component is (recursively) made of sub components and bindings.

contingency: a property of an interface indicates if the functionality of this interface is guaranteed to be available or not, while its component is running. The contingency is either optional or mandatory.

controller: one of the two parts of a component, the other one being its content. A controller is an abstract entity that embodies the control behavior associated with a particular component. A controller can exercise an arbitrary control over the content of the component it is part of (intercept incoming and outgoing operation invocations for instance).

entity: anything having existence.

factory: a component that can create other components. Generic factories can create several kinds of components, while standard component factories create only one kind of components.

fractal: a property that characterizes entities (objects in nature; sets, functions in mathematics; software components in the case of Fractal) which exhibit a structure at all scales or at least at numerous scales, i.e. whose structure depends explicitly on the resolution at which they are being observed. Some fractal systems are scale invariants - which means they exhibit in fact the same structure at all scales. Fractal software systems are scale invariant. They are modelled as interacting Fractal components which are self similar: they are not identical (of course!) but exhibit the same structure expressed in terms of interfaces, bindings, attributes and controllers at any resolution they are being observed.

intercession: the ability of a component (seen as a program) to modify its own execution state; or to alter its own interpretation or semantics.

interface: an access point to a component, also called a component interface; or a language interface, i.e. a type made of several operation declarations.

- **component interface**: an access point to a component, i.e., a place where operation invocations can be emitted or received.
- **language interface**: a type made of several operation declarations.
- **client interface**: a component interface that emits operation invocations.
- **server interface**: a component interface that receives operation invocations.
- **optional interface**: a component interface whose functionality is *not* guaranteed to be available, while the component is running.
- **mandatory interface**: a component interface whose functionality is guaranteed to be available, while the component is running.
- **external interface**: a component interface that is only accessible from outside the component.
- **internal interface**: a component interface that is only accessible from inside the component, i.e. from its sub components.
- **complementary interface**: of an interface I, a component interface with the same name, signature, contingency, and cardinality as I, but with opposite role and visibility (external or internal).
- **functional interface**: a component interface that corresponds to a provided or required functionality of a component, as opposed to a control interface.

- **control interface**: a component interface that manages a "non functional aspect" of a component, such as introspection, configuration or reconfiguration, and so on. By convention, control interfaces are server interfaces whose name ends with -controller, or is equal to component.

introspection: the ability of a component (seen as a program) to observe and reason about its own execution state.

model: a system of relations between selected concepts - built explicitly at ends of description, explanation or forecast.

name: a value that designates a component interface, but that does not necessarily give access to (reference) it.

naming context: a entity that creates and manages names. Naming contexts can be nested and overlapping, allowing names to be valid in different naming contexts. A component controller constitutes a primitive naming context.

reflection (reflective capabilities): the ability of a component (seen as a program) to manipulate as data the entities that represent its execution state during its own execution. This manipulation can take two forms: introspection and intercession.

role: a property of a component interface, indicates if this interface is a client or server interface.

signature: of a component interface, is the name of the language interface type corresponding to this component interface.

template: a special kind of factory that creates components that are "isomorphic" to itself.

type: a set of structural properties common to a set of entities (components and interfaces for instance).

C. Change History

Changes from version 2.0

Changes in the document:

- The Rationale (section 1.1) has been rewritten.
- Section 2.1 and Appendix A have been rewritten (the IDL is now clearly presented as a pseudo, non normative IDL, used only for documentation purposes).

Changes from version 1.0

The document has been completely rewritten. It is now focused exclusively on the Fractal component model, i.e. the considerations about the Fractal framework and the associated roles have been removed. All features of the Fractal model have been made optional, so that the model can be used in many situations, from embedded software to application servers. Finally the Fractal model itself has been revised, in order to clarify and/or simplify some points (binding control, interface collections...).

Changes in the document:

- Some sections have been completely removed: Requirements (2.2), General Model (3.1), Framework Increments (4.2), Roles and Responsibilities (6), Framework Packaging (7), and Bibliography (Appendix B). The material of the remaining sections has been completely reorganized to improve clarity.
- Some completely new sections have been added: Interface Definition Language (2.1), Options (7), Glossary (Appendix B).

Major changes in the API:

- The API is now presented with an abstract IDL, not strictly dedicated to Java.
- The naming and binding framework which was defined in version 0.7.3, and removed in version 0.8, has been added back.
- ComponentIdentity and InterfaceReference have been renamed and decoupled, and both interfaces are now optional. As a consequence, InterfaceReference has been replaced with **any** in the signature of the getFc[Internal]Interface[s], lookupFc, and bindFc operations.
- The BindingController interface has been modified so that UserBindingController is no longer necessary. This latter interface has then been removed.
- Attributes can now be readable, writable or both: attribute controllers are not forced anymore to define a pair of accessors (getters and setters).
- Addition of the SuperController and NameController interfaces.

- Templates are now optional. The Template interface has been slightly modified and renamed into Factory. TemplateFactory has been simplified and renamed into GenericFactory.
- The type system semantic has been slightly modified, but not its API.
- All exceptions are now checked exceptions.

Changes from version 0.9

Changes in the document:

- Minor typographic changes.
- Sections 4.14 and 5 have been updated to take into account the API changes (see below).
- Addition of a "Rationale" paragraph in Section 4.2.1.
- Addition of an appendix "Features Deferred to Future Releases".

Major changes in the API:

- Removal of setFCControllerDesc and setFCContentDesc operations in Template. These operations have been replaced with two new parameters in TemplateFactory.createTemplate.

Minor changes in the API:

- The ...FC... pattern has been replaced with ...Fc...

Changes from version 0.8-0

Changes in the document:

- Section "Roles and Contracts" has moved from Section 3 to Section 6. The role Framework Provider has been refined (and the section has been renamed "Roles and Responsibilities").
- Section "Framework Increments" has been developed - especially the section "Programming Support Increments".
- Section "Programming Support Increments" has been removed for it was mostly Julia-specific. As such it should be available in the fore coming Julia documentation. This point is linked to the two points above.

Major changes in the API:

- Removal of the naming system introduced in the previous version of Fractal. This point is linked to discussions about a potential ObjectWeb naming system.
- Introduction of the interface UserBindingController. It is very close from the interface LocalBindingController of version 0.7.2. Its purpose is to ease call-backs implementation by programmers in "user components", i.e. Java objects.
- Refinement of the exceptions management: introduction of Fractal specific exceptions:
  - NoSuchInterfaceException, IllegalBindingException, IllegalContentException, IllegalLifeCycleException, InstantiationException.
- Refinement of the instantiation phase of components: introduction of an additional parameter in the template creation operation createFCTemplate in TemplateFactory and of two operations setFCControllerDesc and setFCContentDesc in Template.

Minor changes in the API:

- The Fractal API previously defined in the org.objectweb.fractal package is now defined in the org.objectweb.fractal.api package. Fractal implementations will be defined in org.objectweb.fractal.*xxx* packages: the Fractal Reference Implementation for instance is defined in org.objectweb.fractal.julia.
- Systematization on the usage of a naming convention to prevent name clashes. All method names are built using the naming convention: verb+"FC"+noun as in getFCInterfaces.

Changes from version 0.7.3-0

Changes in the document:

- A separate section is now devoted to the running example.
- Removal of the appendix "Features Deferred To Future Releases" (which was empty anyway).
- Addition of the appendix "Bibliography" and bibliographic references in text.

Major changes in the API:

- Introduction of a naming framework that defines interfaces Name and NamingContext. Interface InterfaceReference (defined in package org.objectweb.fractal) now inherits from Name (defined in org.objectweb.naming).
- Typed interfaces references: an interface reference (InterfaceReference) now has the Java type of the interface it references. This is a major change with respect to the Fractal programming model. The major consequences are: 1) explicit binding operations (bind) can be avoided (but cast operations are needed instead) and 2) the bootstrap process is simpler (a bootstrap component does not have to be a container anymore).

- Binding controllers now only deal with local bindings (same address space, same enclosing component). The interface LocalBindingController has been suppressed. A method check has been introduced (see below).

Minor changes in the API:

- Name of the java package are changed from org.objectweb.compfw to org.objectweb.fractal.
- Replacement of methods create*XXX*InterfaceType with one single method createFCItfType with Boolean parameters (isClient, isOptional, isCollection) in interface TypeFactory.
- Addition of a method check (which checks that bindings between components are really local) in interface ContentController.