Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Petr Nevařil

Metadata management for Fractal component model

Department of Software Engineering

Supervisor: Mgr. Petr Hnětynka, PhD.

Study branch: Computer Science

Prague 2006

I would like to thank my supervisor Petr Hnětynka for his guidance of this thesis.

I do declare that I wrote this thesis on my own and that references include all the sources of information I have exploited. I agree with lending of this thesis.

Prague, April 2006 Petr Nevařil

Contents

Title: Metadata management for Fractal component model
Author: Petr Nevařil
Department: Department of software engineering
Supervisor: Mgr. Petr Hnětynka, PhD.
Supervisor's e-mail address: hnetynka@nenya.ms.mff.cuni.cz
Abstract: A metadata management is one of the key features of modern component based systems. The most contemporary used standard for metadata management is OMG Meta Object Facilities (MOF). Having the metadata in a MOF-based repository plays an important role for developing software using the Model Driven Architecture (MDA) approach, i.e. developing the system by sequence of transformations of its conceptual model. The thesis analyze the Objectweb's Fractal component model and designs its MOF-based metamodel. Designing of the metamodel is complicated by existence of several conformance levels to the Fractal component model. The Fractal metamodel proposed in this thesis is applicable for describing components system conforming to Fractal at an arbitrary level.
Keywords: Fractal, MDA, metadata, component-based systems

Název práce: Správa metadat pro komponentový model Fractal
Autor: Petr Nevařil
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Petr Hnětynka, PhD.
e-mail vedoucího: hnetynka@nenya.ms.mff.cuni.cz
Abstrakt: Správa metadat je jednou z klíčových oblastí v moderních komponentových systémech. V současné době je nejpoužívanějším standardem pro správu metadat OMG MOF (Meta Object Facilities). Metadata definovaná podle tohoto standardu hrají důležitou při vývoji software pomocí MDA (Model Driven Architecture), kde aplikace vzniká postupnými transformacemi jejího konceptuálního modelu. V této diplomové práci je analyzován komponentový systém Fractal a je navržen jeho metamodel pomocí MOF. Návrh metamodelu je obtížný kvůli existenci několika úrovní, na kterých může komponentový systém modelu Fractal vyhovovat. V této práci je navržen metamodel použitelný pro komponentový systém libovolné úrovně.
Klíčová slova: Fractal, MDA, metadata, komponentové systémy

Chapter 1

Introduction

With the continued, and growing, diversity of software systems, the interoperability will never be achieved by forcing all software development to be based on a single operating system, programming language, instruction set architecture, application server framework or any other choice. There are simply too many platforms in existence, and too many conflicting implementation requirements, to ever agree on a single choice in any of these fields. Thus the possibilities to facilitate and automatize the integration of application developed for different platforms are examined.

MDA[14] (Model-driven architecture) is an approach to software development which emphasizes the role of machine readable models in building software systems. It introduces designing a system in a platform independent way and consequential transformations of the platform independent model to a model for a particular platform. Such a design leads not only to systems that are easier to develop, integrate and maintain but also adds the ability to automate at least some of the construction.

Meta-Object Facility (MOF)[15] is an OMG standard for creating metamodels of software systems. MOF defines an abstract language and a framework for specifying, constructing, and managing technology neutral metamodels. Importantly, MOF defines a framework for implementing repositories that hold metadata described by a metamodel. To enable metadata interchange among repositories, OMG defined XML-based format named XMI (XML Metadata Interchange)[17].

By enforcing a strict separation of interface and implementation and by making software architecture explicit, component-based programming can fully take advantages of the MDA approach.

The Fractal component model[16] is a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure systems and applications, from operating systems to middleware platforms and to graphical user interfaces. The main features of the Fractal model are composite components (to have a uniform view of application at various abstraction levels), introspection capabilities (to monitor a running system), and configuration and reconfiguration capabilities (to deploy and dynamically reconfigure an application). The Fractal specification defines several conformance levels to the Fractal component model. A component system is conforming to a given Fractal level if it provides a defined set of features.

1.1 The goal of the thesis

The Fractal component model is not defined as a big, fixed specification that all Fractal components have to 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. To allow interoperability of different component systems based on the Fractal component model, conformance levels to the Fractal component model are defined.

Having the Fractal metadata stored in a MOF compliant repository is important not only for automatization of model transformations (and therefore allowing the MDA approach to software development) but also it provides a standardized interface for Fractal metadata and allows usage of MOF-based tools for managing them.

The goal of this thesis is to create an extensible metamodel of the Fractal component model (using the MOF standard) that can be used to describe Fractal component systems conforming to Fractal at all possible levels. The existence of several conformance levels and extreme modularity and extensibility of Fractal makes the creation of the metamodel difficult.

1.2 The structure of the thesis

To achieve the goal, the thesis is structured as follows. Chapter 2 provides a brief description of MDA, MOF and Fractal, Chapter 3 presents the proposed solution to fulfill the thesis goal, Chapter 4 describes projects and papers that deal with objectives related to this thesis, while Chapter 5 eveluates and concludes the thesis and suggests the possible future work.

Chapter 2

Background

This chapter provides a reader with information necessary to understand the solution described in Chapter 3. It describes MDA, MOF and Fractal.

2.1 MDA

MDA[14] is an approach to software development, which increases the power of models in that work. It is model-driven because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification. Using the MDA approach to software development, design has several stages using various kinds of models and transformations between them.

2.1.1 Model vs. metamodel

Comprehension of the distinction between the terms like model and metamodel, data and metadata, and understanding what does the *meta* prefix mean is very important for understanding the rest of this thesis. The term *model* is generally usually used to denote a description of something, typically something in the real world. The data that represent the model are called *metadata*. It is a general term for data that in some sense describes information (another data). The description of metadata is determined by a *metamodel*. In other words, a metamodel is a model of some kind of metadata (i.e., metamodel = model of model).

For example when modeling the inhabitation of people, information like John Smith (32 years old) lives in New York, Petr Novak (20) lives in Prague are the data. Figure 2.1 shows a possible model of this data in UML. In

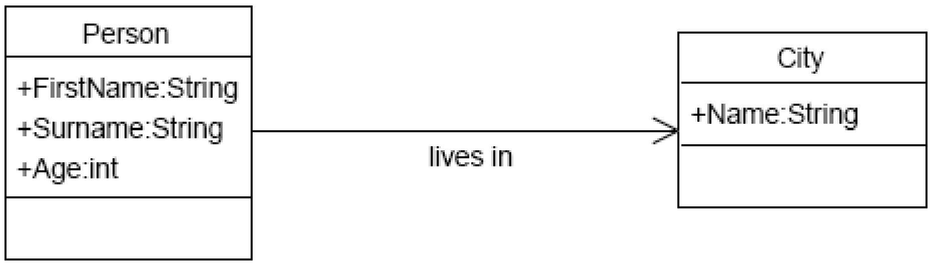


Figure 2.1: Model example

this case, metadata is the information that describes model elements in this model, i.e. that each person is described with an instance of the Person class, having the FirstName, the Surname and the Age attributes, with the relation to an instance of the City class. Constructs and elements, which can be used to build the model (i.e., classes, attributes, associations, ...), are specified by the UML metamodel. A part of the UML metamodel (very simplified) is depicted in Figure 2.2.

2.1.2 Computation independent model (CIM)

At first, the requirements for a system are modeled in a computation independent model describing the situation in which the system will be used. Such a model is sometimes called a domain model and a vocabulary that is familiar to the practitioners of the domain in question is used in its specification. Typically, such a model is independent of any implementation of the system.

It is assumed that the primary user of the CIM, the domain practitioner, has no knowledge about the models or artifacts used to realize the functionality for which the requirements are articulated in the CIM. The CIM plays an important role in bridging the gap between those that are experts about the domain and its requirements on the one hand, and those that are experts of

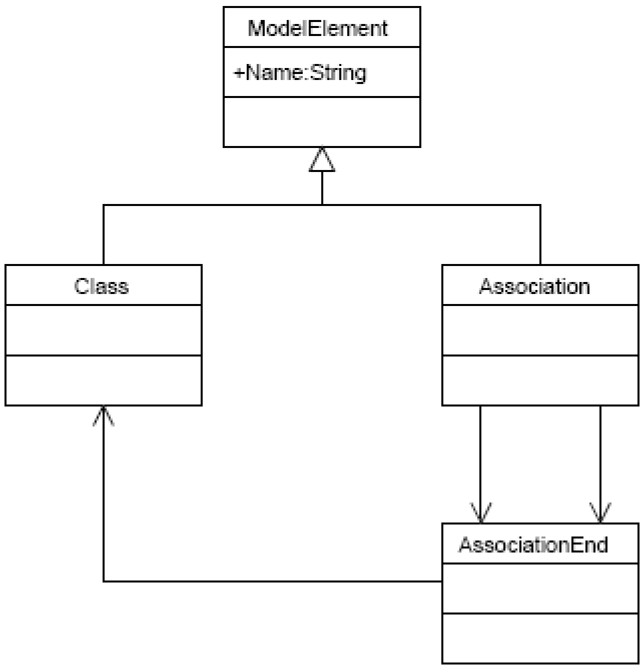


Figure 2.2: Very simplified UML metamodel

the design and construction of the artifacts that together satisfy the domain requirements, on the other.

2.1.3 Platform Independent Model (PIM)

Having the CIM, a platform independent model can be build. It describes the system, but not shows details of its use of its platform. A platform is defined as a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented. CORBA[13] is an example of platform that enables the remote invocation and event architectural styles. Another platform that enables a components and containers style is Java 2 Enterprise Edition (J2EE)[22].

2.1.4 Platform Specific Model (PSM)

A PSM combines the specifications in the PIM with the details that specify how that system uses a particular platform. It may provide more or less detail, depending on its purpose. A PSM will be an implementation, if it provides all the information needed to construct a system and to put it into operation, or may acts as a PIM that is used for further refinement to a PSM that can be directly implemented.

2.1.5 Transformation of PIM to PSM

Model transformation is a process of converting one model to another model of the same system. As illustrated in Figure 2.3 the transformation from PIM to PSM usually consists of a sequence of several transformations, starting with the generic model and resulting in a model specific for a particular platform. Even, an implementation can be the result of the transformation sequence, because implementation is, in a manner, a kind of model. There are many ways in which such a transformation may be done. Transformations can use different mixtures of manual and automatic transformation for adding the information necessary to translate a PIM to the PSM.

2.1.6 Metamodel Mappings

A *model type mapping* specifies a mapping from any model built using types specified in the PIM language to models expressed using types from a PSM language. A metamodel mapping is a specific example of a model type mapping, where the types of model elements in the PIM and the PSM are both specified as MOF metamodels. In this case the mapping gives rules and/or algorithms expressed in terms of all instances of types in the metamodel specifying the PIM language resulting in the generation of instances of types in the metamodel specifying the PSM language(s).

Using MDA, the development of a software system consists of these activities:

- A model is prepared using a platform independent language specified by a metamodel,

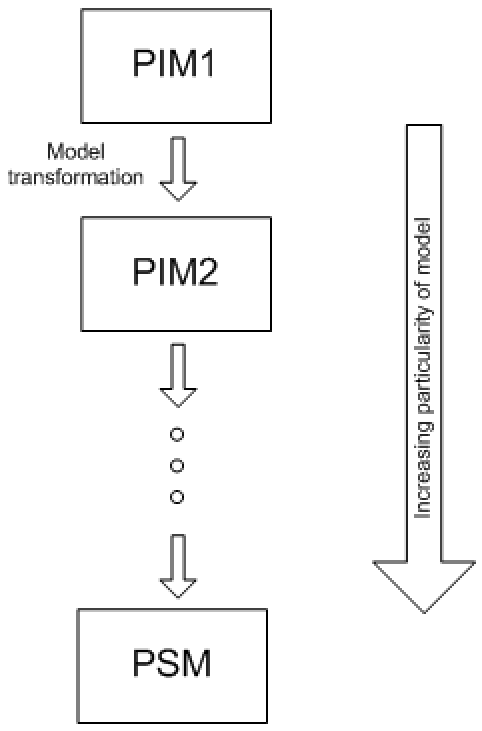


Figure 2.3: Sequence of PIM transformation

- a particular platform is chosen,
- a specification of a transformation for this platform is available or is prepared (this transformation specification is in terms of a mapping between metamodels),
- the mapping guides the transformation of the PIM to produce the PSM.

2.2 MOF

The Meta-Object Facility (MOF)[15] defines an abstract language and a framework for specifying, constructing, and managing technology neutral metamodels. In addition, the MOF defines a framework for implementing repositories that hold metadata (e.g., models) described by the metamodels. This framework uses standard technology mappings to transform MOF metamodels into metadata APIs.

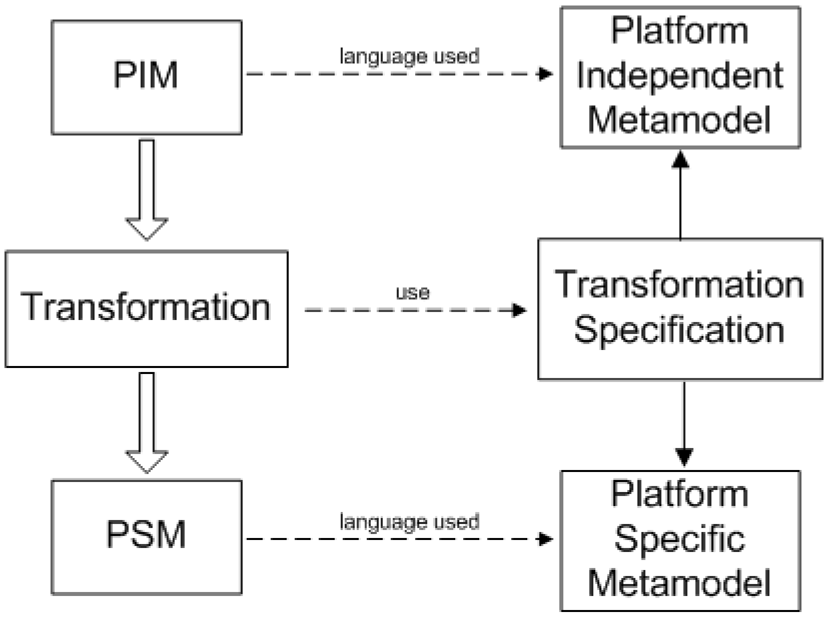

PIM
language used
Platform Independent Metamodel
Transformation
use
Transformation Specification
PSM
language used
Platform Specific Metamodel

Figure 2.4: Usage of metamodels in transformation

Metadata itself is a kind of information, and can accordingly be described by other metadata. In MOF terminology, metadata that describes metadata is called meta-metadata, and a model that consists of a meta-metadata is called a metamodel. The MOF metamodel defines the abstract syntax of the metadata in the MOF representation of a model. Since there are many possible kinds of metadata in a typical system, the MOF framework needs to support many different MOF metamodels. The MOF integrates these metamodels by defining a common abstract syntax for defining metamodels. This abstract syntax is called the MOF Model and is model for metamodels (i.e., a meta-metamodel). The MOF metadata framework is typically depicted as a four layer architecture as shown in Figure 2.5.

The MOF specification defines *MOF Reflective Interfaces* which provide generic interfaces to manipulate all metadata. It works like the Java reflection feature of J2EE and allows information to be accessed dynamically.

Because of the absence of MOF graphical representation, UML is used to depict MOF metamodels. The main reason for using the UML is the similarity of the UML metamodel with the MOF metamodel. The similarity is

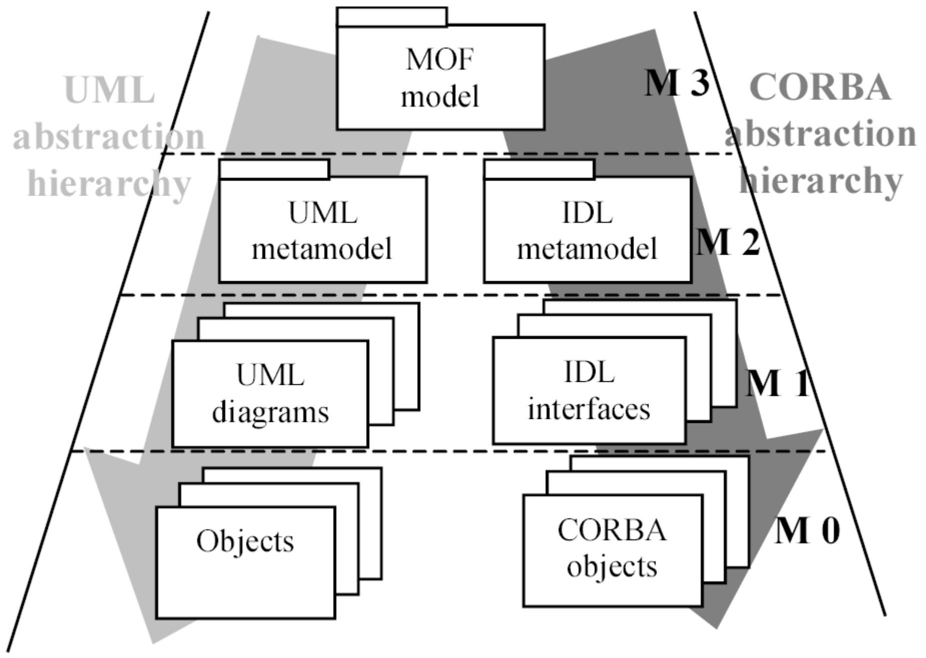


Figure 2.5: Two examples of MOF architecture layers

even so high, that in the latest version of MOF and UML (version 2.0), the core part of MOF and UML metamodel is common for both of them. However, there is no sufficient implementation of the latest MOF version, hence the version 1.4 is used for the purpose of this thesis. The further upgrade to version 2.0 would not be a problem, because there is a straightforward mapping between versions 1.4 and 2.0.

XMI

To enable metadata interchange among repositories, OMG defined a XML-based format named XMI (XML Metadata Interchange)[17]. It defines technology mappings from MOF metamodels to XML DTDs and XML documents. These mappings can be used to define an interchange format for metadata conforming to a given MOF metamodel.

2.3 The Fractal Component Model

2.3.1 Fractal objectives

Existing component-based frameworks and architecture description languages provide only limited support for extension and adaptation. This limitation has several important drawbacks: it prevents the easy and possible 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 usage of these frameworks and languages in different environments, including embedded systems.

The Fractal component model alleviates all above mentioned 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.

The Fractal component model heavily uses the *separation of concerns* design principle[10]. The idea of this principle is to separate into distinct pieces of code or runtime entities the various concerns or aspects of an application: implementing the service provided by the application, but also making the application configurable, secure, available, ... In particular, the Fractal component model uses three specific cases of separation of concerns principle: namely *separation of interface and implementation, component oriented programming,* and *inversion of control.* The first pattern, also called the bridge pattern, corresponds to the separation of the design and the implementation concerns. The second pattern corresponds to the separation of the implementation concern into several composable, smaller concerns, implemented in well separated entities called components. The last pattern corresponds to the separation of the functional and configuration concerns: instead of finding and configuring necessary components and resources themselves, Fractal components are configured and deployed by an external separated entity.

2.3.2 External component structure

Depending on the level of observation, i.e. *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 2.6). Each interface has a name and implements a language interface. One may distinguish two kinds of interfaces: a *client* (or *required*) interface emits operation invocations, while a *server* (or *provided*) interface receives them.

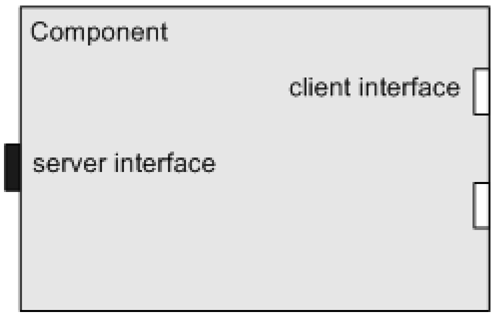


Figure 2.6: External component structure

2.3.3 Internal component structure

Internally a Fractal component is formed from two parts: a *controller* (also called membrane), and a *content* (see Figure 2.7). The content of a component is composed of (a finite number of) other components, called *subcomponents*, 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., 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 (the definition of control interface is stated bellow in this section), is called a *primitive* component. A component without any control interface is called a base component.

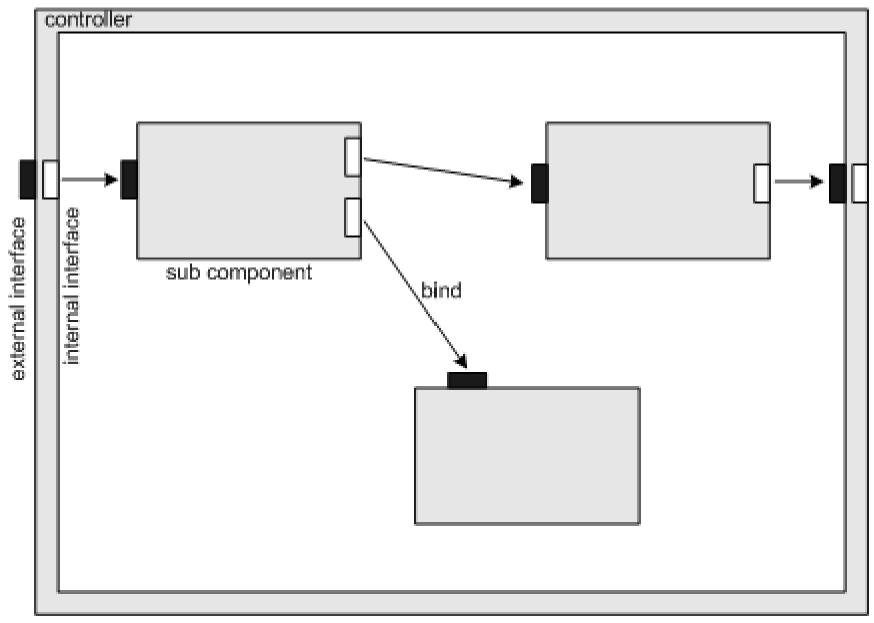


Figure 2.7: Internal component structure

A given component can be added to several other components. Such a component is *shared* between these components. The main application of shared components is to model resources. Because of shared components, the structure of Fractal component, in terms of direct and indirect subcomponents, is not necessarily a tree, but can be a directed acyclic graph.

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 subcomponents. 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. A *binding* is a communication path between component interfaces.

2.3.4 Standardized Fractal control interfaces

Commonly used features of component systems are in Fractal defined as control interfaces. A component can provide the AttributeController to read and write its attributes from outside of the component. The BindingController interface is a standardized interface for binding and unbinding client interfaces of the component providing it. By providing the ContentController interface a component can manage its subcomponents.

Changing an attribute or a binding, or removing a subcomponent, 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 reconfiguration a component can provide the LifeCycleController interface.

2.3.5 Type system

The definition of a simple type system is also a part of the Fractal specification. A component type is just a set of component interface types. A component is represented by the ComponentType interface. This interface defines operations which return types of component interfaces.

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 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 by connected to any server interface). The role indicates if component interfaces of this type are client or server interfaces. The contingency denotes 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.

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

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();
}
```

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.

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

Together with the interfaces for the type system, Fractal specification also defines a subtyping relation for the component and interface types, based on *substitutability.* This relation provides a sufficient (but not necessary) condition such that if a component type T_1 is a subtype of T_2, then a component of type T_1 can replace component of type T_2 in any environment, this environment (other component and bindings) being left unchanged, and both components being seen as black boxes.

2.3.6 Conformance levels

In Fractal component model, most features are optional. For example a Fractal component may define a new semantic for the communication between its subcomponents: instead of specifying that operation invocations follow bindings, it can for example specify that operation invocations are broadcasted to all the subcomponents, in order to model an asynchronous "reactive space". A Fractal component may also refine the internal component structure, by specifying that the component's controller can, like the component's content, contain subcomponents. Such a Fractal component can then provide new control interfaces to introspect and reconfigure the subcomponents of its controller part.

The advantage of such 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 not be generally able to work together, because they will use different, and potentially incompatible, options or extensions of the Fractal model. In order to reduce this problem, named sets of options called *conformance levels* are defined. The goal is to be able to say, or even certify, that a given Fractal application or tool is conforming to the Fractal model of level X. Then it will be easy to know, which Fractal applications and tools can work together, by comparing their conformance level to the Fractal model.

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

Conforming at level 0.1 requires following features: all components with configurable attributes have to provide the AttributeController interface, all components with client interfaces have to provide the BindingController interface, all components that expose their content have to provide the ContentController interface, and all components that expose their life cycle have to provide the LifeCycleController interface. Of course, these requirements do not prevent components from providing additional control interfaces, including extension and alternatives of the previous interfaces.

Level 1

To conform at level 1 to Fractal component model, all components have to provide, at least, the Component interface, which specifies the introspection capability of the component. Implementation of this interface enables to examine the component interfaces.

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

Level 1.1

The level 1.1 components have to provide the Component interface, with the same additional requirements as for the level 0.1, concerning the control interfaces.

Level 2

This level extends the level 1, by adding the necessity to implement the Interface interface by all component interfaces. Implementation of this interface extends component introspection capabilities with the possibility to discover interface properties on the fly.

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

Level 2.1

For conforming at level 2.1, there are the same requirements as for level 2, with the same additional requirements as for levels 0.1, and 1.1 concerning the control interfaces.

Level 3

All Fractal level 3 components have to implemenent the reflective interfaces defined in levels 2 and 1, and they have to use (an extension of) the Fractal type system.

Level 3.1

The additional requirements for conforming at level 3.1, are the same as for levels 0.1, 1.1 and 2.1.

Level 3.2

A level 3.1 component system is a level 3.2 component system if a bootstrap component is accessible from a "well-known" name. This bootstrap component have to provide a GenericFactory and a TypeFactory interface. Moreover, the GenericFactory interface has to be able to create components with any control interfaces in the set of Fractal standardized control interfaces (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.

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

Level 3.3

The additional requirement, that the GenericFactory interface of the bootstrap component must be able to create primitive and composite template components, is required from the level 3.2 components to be the level 3.3 components.

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.

2.3.7 Fractal ADL

The Fractal Architecture Description Language (ADL) is an open and extensible language to define component architectures for the Fractal component model. Fractal ADL is a XML based ADL that can be used to describe Fractal component configuration. An example of Fractal ADL describing the component in Figure 2.8 is stated bellow:

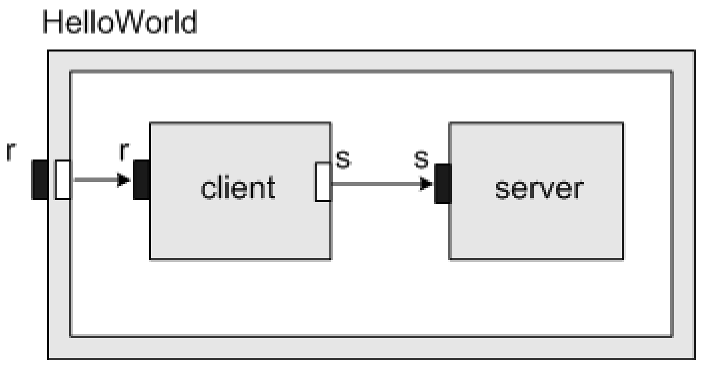


Figure 2.8: Hello world component

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
    "classpath://org/objectweb/fractal/adl/xml/basic.dtd">

<definition name="HelloWorld">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <component name="client">
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <interface name="s" role="client" signature="Service"/>
    <content class="ClientImpl"/>
  </component>
  <component name="server">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
  </component>
  <binding client="this.r" server="client.r"/>
  <binding client="client.s" server="server.s"/>
</definition>
```

2.3.8 Julia

Julia[7] is the reference implementation of Fractal component model in Java conforming at level 3.3 to the Fractal specification. Julia implements an extensible set of control objects, from which the user can freely choose and assemble the controller objects he or she wants, in order to build the controller part of a Fractal component. The set of control objects allows user to instantiate non reconfigurable but very efficient or, on the contrary, completely reconfigurable but less efficient components. It also allows the user to mix these components, in order to use different optimization levels for the different parts of an application.

A Fractal component is generally represented by a set of Java objects, which can be separated into three groups (see Figure 2.9):

- the objects that implement the component interfaces (one object per component interface; each object has an "impl" link to an object that really implements the Java interface, and to which all methods calls are delegated; this reference is null for client interfaces),
- the objects that implement the controller part of the component (a controller object can implement zero or more control interfaces),
- and the objects that implement the content part of the component (not shown in the figure).

The objects that represent the controller part of a component can be separated into two groups: the objects that implement the control interfaces, and (optional) *interceptor* objects that intercepts incoming and/or outgoing method calls on functional (or business, or user) interfaces. Each controller object can contain references to other controller objects. The objects that implement the content part of a component can be subcomponents (for composite components), or user objects (for primitive components).

2.4 Goals revisited

As described in previous sections the possibility to adopt only some of the Fractal concepts (and to extend them in an arbitrary way) enables the variety of Fractal components system to be really huge. The goal of this thesis is to create a common Fractal metamodel, which can be applicable and easily extended for component systems conforming to Fractal at an arbitrary level defined in the Fractal component model specification (see Section 2.3.6). Then, through the use of the Fractal metamodel, to create a basic support for development of Fractal components using the MDA approach to software development: implement a MOF-based Fractal metadata repository, and a transformation of the Fractal metadata from the Fractal standard for describing metadata — the Fractal ADL, to the MDA standard — MOF (i.e., loading of metadata defined in ADL to the MOF-based repository). Together with the Fractal metamodel the extension of this metamodel for Julia component system will be created, as a demonstration of the extensibility and applicability of the metamodel.

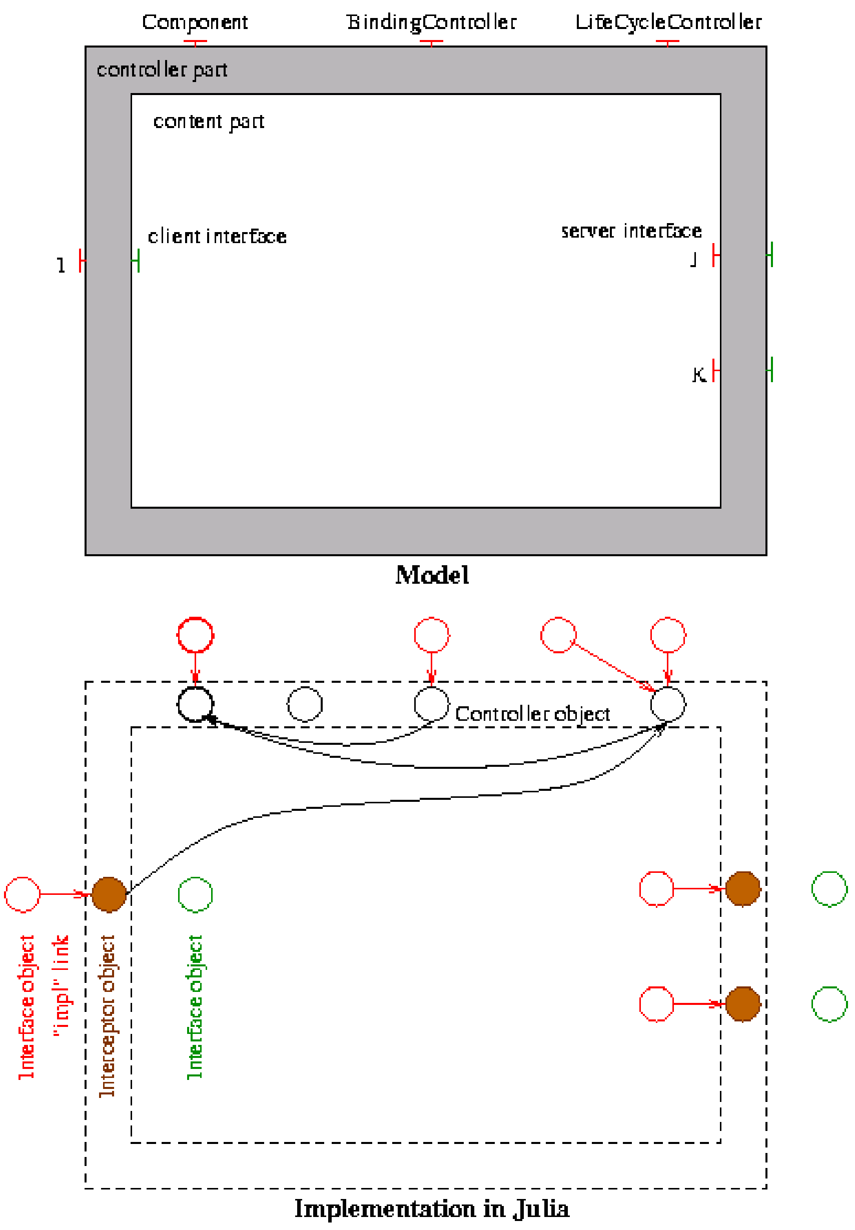


Figure 2.9: An abstract component and a possible implementation in Julia

Chapter 3

Solution

This chapter presents the proposed metamodels of Fractal and Julia and describes the implementation of the Fractal metadata repository and related tools.

3.1 Fractal metamodel

In following sections, the particular segments of the Fractal metamodel are described in detail, together with the presentation of the alternative possibilities for modeling relations between given concepts, and reasoning why the selected one is the most suitable. The whole Fractal metamodel is presented in Appendix A. As it was explained in Section 2.2, UML class diagrams are used to describe the Fractal metamodel.

3.1.1 Components

As Fractal is a component model, the component is the essential concept of the Fractal metamodel. As described in Section 2.3.3, the Fractal component model specification defines three different kinds of components: a base component, a primitive component, and a composite component; according to increasing level of control it provides. Figure 3.1 shows straightforward capturing of this relation in the inheritance hierarchy. This relation among component kinds corresponds with seeing the base component as a black box with no control interfaces; the primitive component as a black box with distinguishing between the functional and control part of it; and the composite component as a white box, with the ability to see its content (subcomponents).

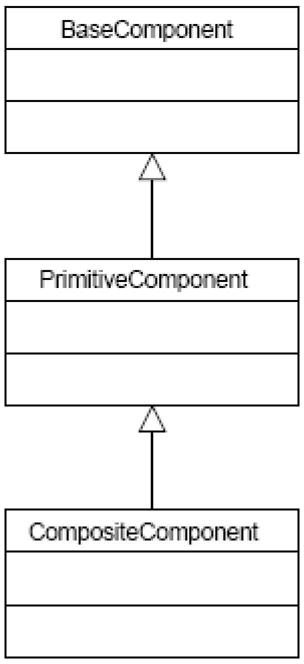


Figure 3.1: Possible components metamodel

From another point of view the base component, the primitive component, and the composite component are three distinct kinds of components, as illustrated in Figure 3.2. Common for all of them is being a kind of component, but there is no direct relation between them. This approacha is more appropriate than the previous one, because composite component is surely not a special kind of primitive component, as it is denoted by the inheritance in Figure 3.1.

However, the second modeling approach to modeling the component kinds has several drawbacks. An extension of this metamodel (to could describe a particular Fractal component system) demands too much effort when adding properties or behavior common for all kinds of component. Subclasses of all component kinds have to be created, when the extension is done without

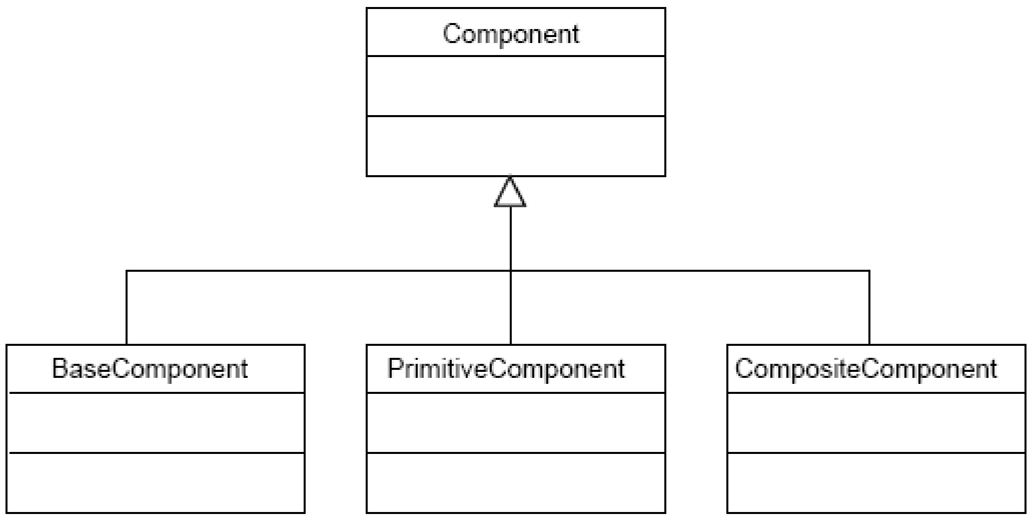


Figure 3.2: Possible components metamodel

changing the common Fractal metamodel. The inheritance hierarchy of the component kinds is unnecessarily complicated. The only difference between the base and the primitive component is in containing a control interface, and thus the distinction between them can be done by testing of the control interface presence in the component. The properties and the behavior of them are not different, and therefore there is no need to model them as separated component subclasses. Similarly, the difference between the primitive and the composite component is only in the presence or absence of a subcomponent. Analogous to precedent reasoning, there is no need to create a special component subclass for composite component. For further extension of the metamodel, it is preferable to keep the metamodel as simple as possible. The resulting metamodel has the only one class for all component kinds with the attribute `name` necessary for the component identification.

Internally, a Fractal component is composed of two parts — a content and a controller (see Section 2.3.3); and the component content is formed out of component subcomponents. Figure 3.3 illustrates straightforward capturing of these relations. However this model is unnecessarily complicated. It is useless to create a special class (the `Content` class) for modeling component containment hierarchy; better solution is to capture the subcomponent re-

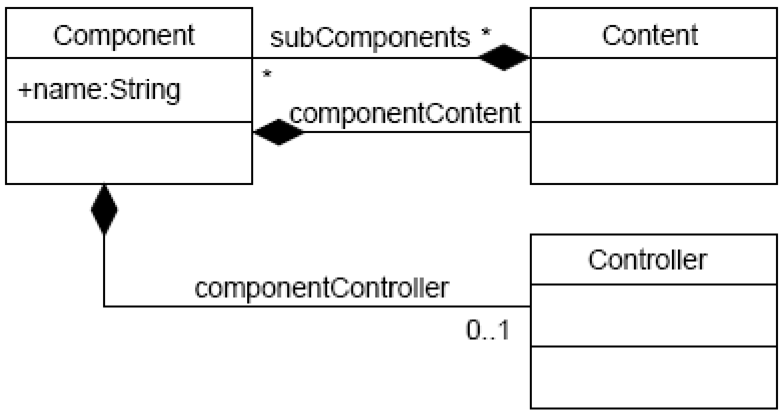


Figure 3.3: Possible component internal structure metamodel

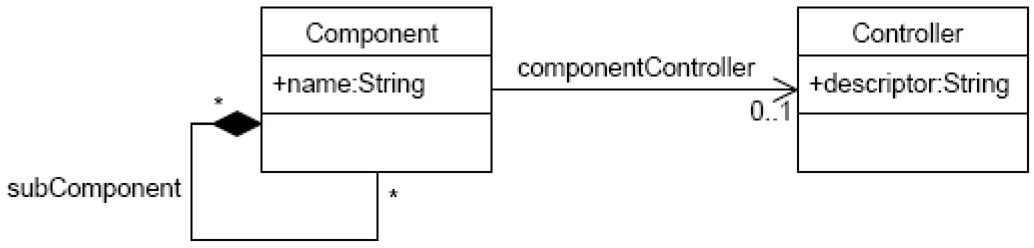


Figure 3.4: Resulting component structure metamodel

lation with a composition association. Multiplicity of both association ends of this composition is $0..n$, because a component can contain an arbitrary number of subcomponents and can be shared among an arbitrary number of components.

The controller of a component embodies the control behavior associated with the component. These control aspects of the component are described by ist control interfaces. Hence, the metamodel element for representing the controller is not necessary, it is fully described by a set of control interfaces. However, the Fractal metamodel contains the Controller element for representing the entity that implements the control interfaces. The componentController association is not a composition but rather a reference from the Component element to the controller. Because of the optionality of having a controller, the multiplicity of the relation is 0..1. The resuling metamodel of the internal component structure is depicted in Figure 3.4.

Alternative metamodel of component controller

As it was described in Section 2.3.3, component can have a control part — a controller. We can think of it as a membrane that enables to control its content. However, a component with a controller must be accessible in the same way as a component without it (to preserve the uniform view principle). Probably the best way of implementing it, is using the *Decorator* [5] design pattern (see Figure 3.5).

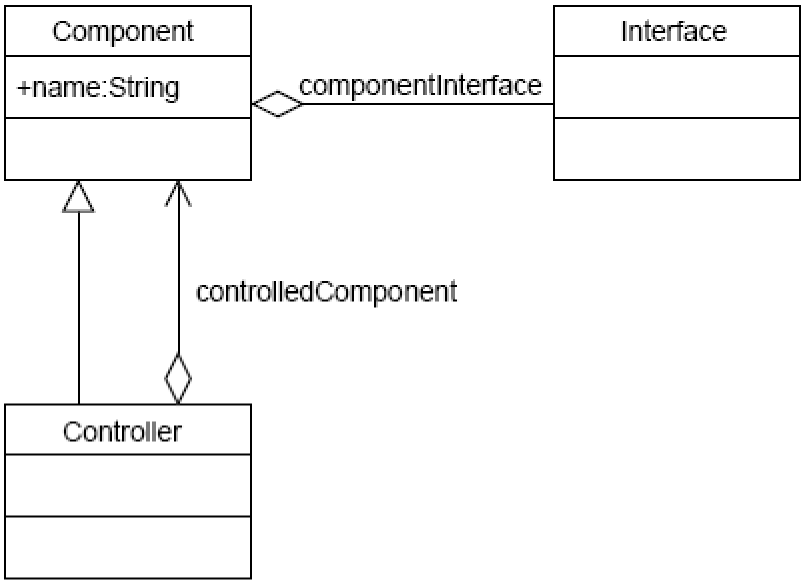


Figure 3.5: Possible controller implementation

By inheriting from the Component, the semantics of the Controller element in the metamodel would be extended beyond the concepts of the Fractal specification. For example, it would allow the Controller to have subcomponents, it would require from the Controller to have a type, etc.

3.1.2 Interfaces and bindings

An interface represents either a required or provided service of a component. Every interface is identified by a name within its component. It is necessary to distinguish between the terms *component interface* and *language interface*: a component has an interface, which implements a language interface, e.g. a component has an interface with name "runComponent", which implements the java.lang.Runnable language interface (in this case the Java interface). The relation between a component and its interfaces is depicted in Figure 3.6. The Attribute name of the Interface class represent name of the component interface. The distinction between the functional and the control interfaces is done via the value of the boolean control attribute. Additional properties of interfaces such as cardinality, contingency, language interface it implements etc. are discussed in Section 3.1.3.

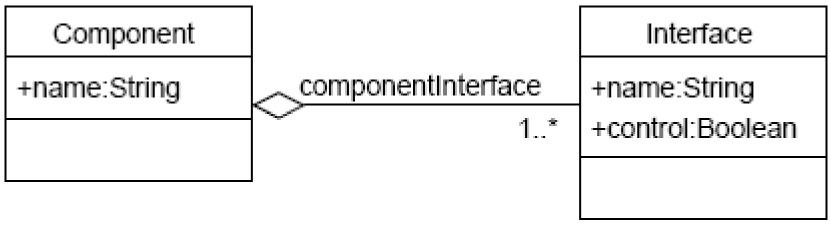


Figure 3.6: Component interfaces

The semantics of binding creation between a client interface of a component and a server interface of another component is defined in the Fractal specification as a control interface BindingController. Providing of this interface for every client interface is one of the requirements for conforming to the Fractal component model at the levels $X.1$ (see Section 2.3.6). Figure 3.7 presents a separate class for bindings. Representation of the binding as

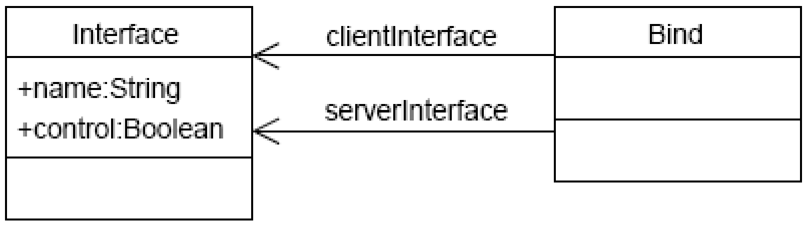


Figure 3.7: Interface bindings

a stand-alone class has several advantages. It allows simple extension of the binding concept by adding various attributes, such as a binding validity, a durability etc. The next advantage is that the Interface class is not "polluted" by the reference to connected interface and the part of metamodel, which represents the standardized Fractal bindings (i.e., bindings usings the control interface BindingController), can be easily eliminated from the metamodel. This is useful in case that a Fractal component system uses its own way to solve communication between component interfaces .

Figure 3.8 shows the alternative way (the simplest one) of representing the interface bindings. This model is not very convenient, because the Binding association is meaningful for client interfaces only and is declared for all interface kinds and it also makes difficult to create extension of binding (associations can not be extended).

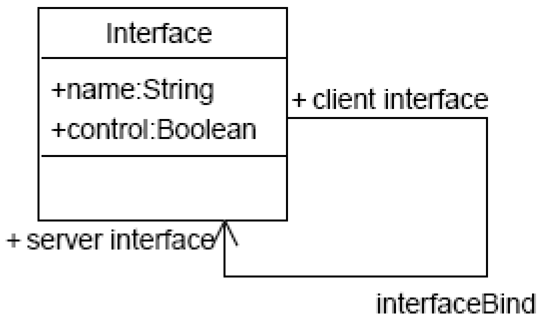


Figure 3.8: Possible interface bindings metamodel

3.1.3 Type system

Fractal defines a type system to allow checking interface bindings and to be able to decide if a component can be replaced by another component in its environment (other components, bindings). A simple typing is required for all components conforming to Fractal at level 1 and higher and for all interfaces of components at level 2. Modeling of this simple typing is quite straightforward and follows the definition of Fractal type interfaces. Figure 3.9 presents model of the simple Fractal type system.

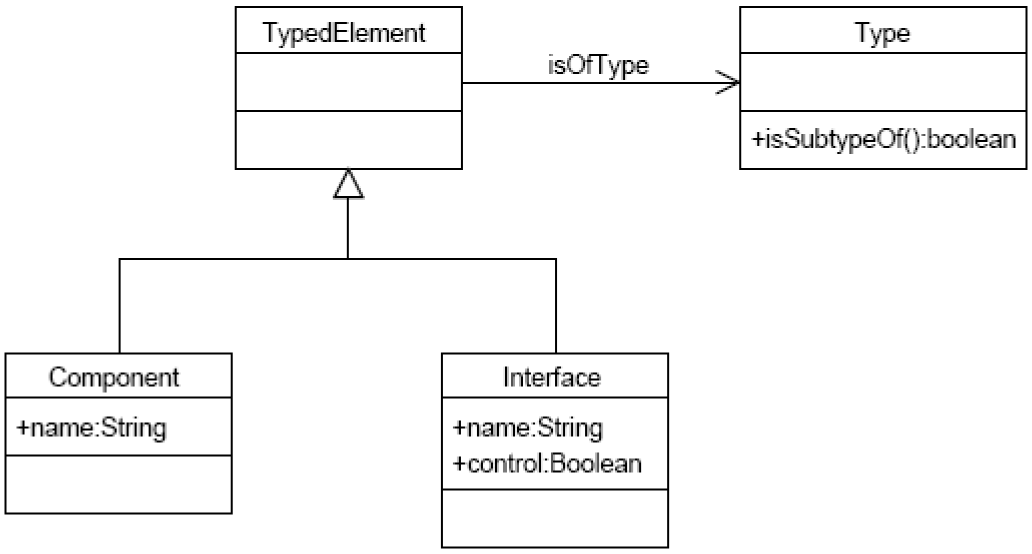


Figure 3.9: Abstract type classes

Interface type

An interface has several properties such as cardinality, contingency, signature etc. Distinction between properties, which are interface attributes and which are attributes of the interface type, depends on the role of the property when resolving interface substitutability by another interface (substitutability is the base of subtype relation definition in the Fractal specification). According to the substitutability definition in the Fractal specification and in compliance with the interfaces defined in the Fractal type system, properties of the interface type are:

- a signature representing language type of interface (e.g., name of a Java class),
- a boolean attribute client denoting if the interface is a client or server interface,
- an attribute optional describing interface contingency,
- and the cardinality represented as a boolean attribute collection (see Figure 3.10).

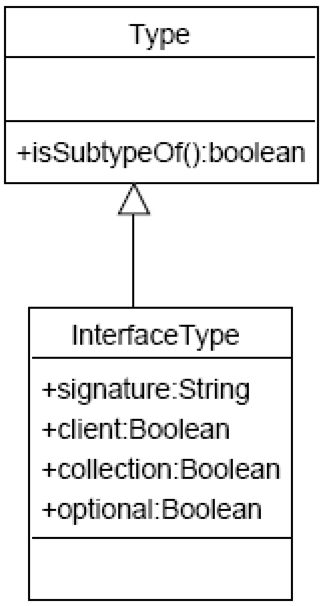


Figure 3.10: Interface type

In addition to above mentioned properties, an interface can be either internal or external (see Section 2.3.3). In this case, the distinction if it is an Interface property, or it is a property of the InterfaceType, is not as clear as in the previous cases. In accordance to definition of the Interface interface (it contains an isFcInternalItf method) it is more suitable to model it as an attribute of the Interface class (see Figure 3.11).

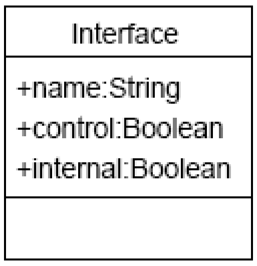


Figure 3.11: Interface

Component type

In the Fractal specification, component type is defined as a set of interface types. In the metamodel, component type is represented as a ComponentType class. It contains an operation for getting a particular interface type as depicted in Figure 3.12.

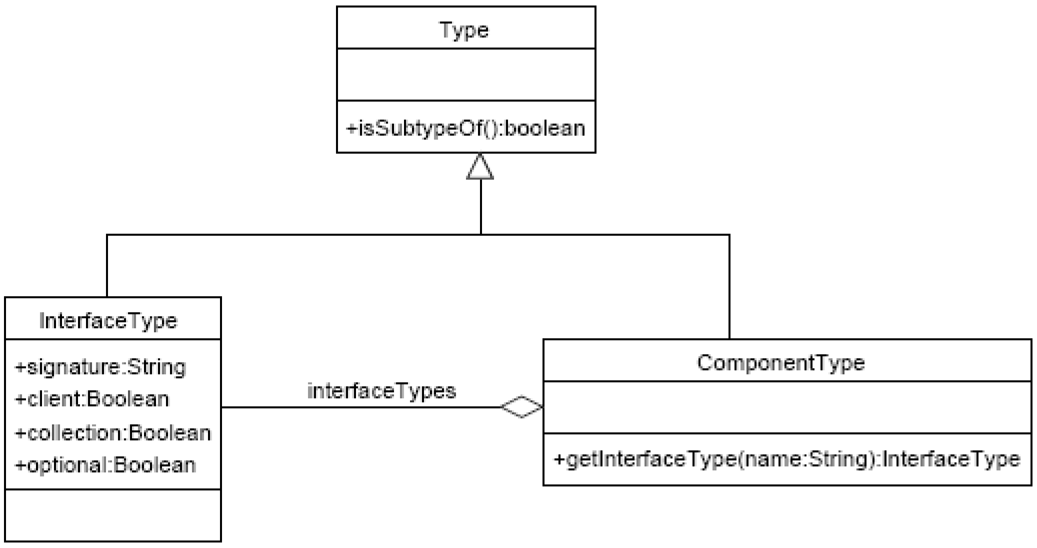


Figure 3.12: Component type

3.1.4 Component attributes

In the Fractal specification, management of component attributes is defined as the AttributeController interface. To conform to Fractal level $X.1$ and higher (see Section 2.3.6), all configurable component properties have to be accessed via this interface. The containment of attributes is modeled as an aggregating association allowing a component to have $0..n$ attributes (see Figure 3.13)

3.1.5 Component implementation

In component oriented programming, the separation of concerns principle is heavily used. The components together with their interfaces define architec-

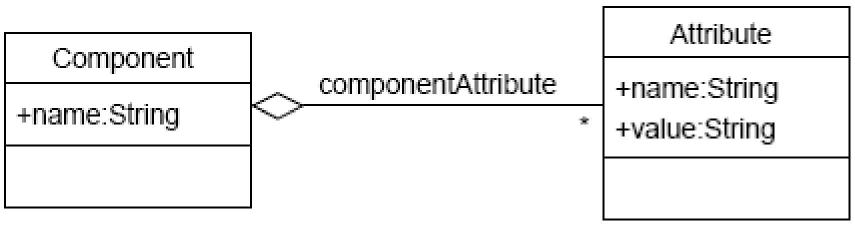


Figure 3.13: Component attributes

ture of the system and this architecture can be than implemented in various ways. Implementation of component is represented as a stand-alone element with a signature attribute, which defines the object (e.g., Java class) that implements it (see Figure 3.14).

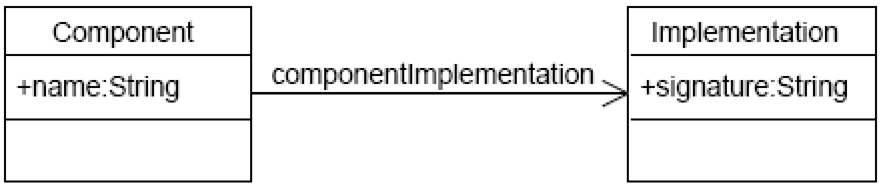


Figure 3.14: Component implementation

3.2 Extensibility of the metamodel

This section details the applicability of the proposed Fractal metamodel for describing the Fractal component systems that conform to the Fractal component model at an arbitrary level.

It is necessary to realize, that the conformance levels to the Fractal component model do not prescribe structure of the components, but rather define the control and reflective capabilities required from the components. In order to provide some information about the component attributes and about the internal component structure, the data describing this information has to be available. The proposed Fractal metamodel is the suggestion of these metadata structure. It consists of the part common for all conformance levels and of the part which is for some levels optional.

The optional part of the metamodel serves for modeling advanced Fractal features, such as the Fractal type system. However, it can be used to describe Fractal component systems at the lowest conformance levels too. It is important to distinguish the necessity of the run-time availability of the component metadata from the semantic structure of components in a component system. Component system can adopt the concepts defined in Fractal specification to describe its components, but because of limited resources, it does not need to support the Fractal reflective interfaces.

3.2.1 Level 0

The core concepts of Fractal — nesting of components with the uniform view of component and composite component; and the approach to accessing component via well defined access points — the interfaces; are common for all conformance levels. The Component class, Interface class, subComponent association, and componentInterface association are sufficient to represent the core concepts. Because of the compatibility with the metamodel of higher Fractal conformance levels, the inheritance from the TypedElement class is necessary. Since the level 0 components do not use any type information a singleton instance of the Type class — the UnknownType is referenced by all typed elements.

3.2.2 Levels $X.1$

To conform to level $X.1$ (i.e., 0.1, 1.1, 2.1 or 3.1) all components must provide a set of control interfaces. Each control interface is represented as an instance of the Interface class with the attribute control set to true. It is necessary to realize, that the metamodel does not describe semantics of the control capabilities, but only register their presence or absence. This is the reason why any special constructs for the control interfaces are not required.

The AttributeController and the BindingController are special cases of control interfaces. To could use these interfaces, additional information is necessary. The AttributeController interface provides access to component attributes. There is a separate class (the Attribute class) for representing com-

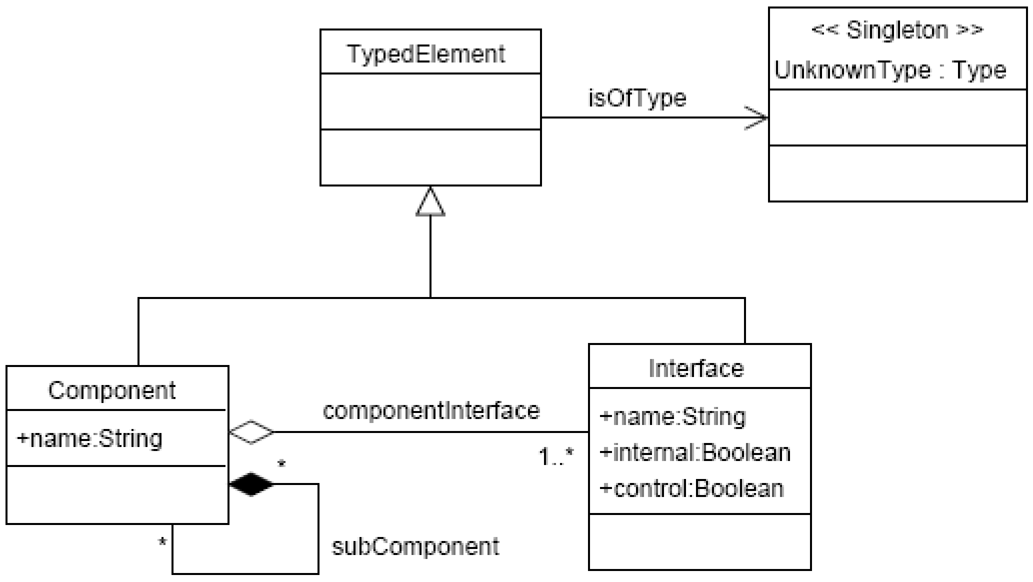


Figure 3.15: Core metamodel

ponent attributes. Usage of this class is necessary for all levels that use AttributeController for accessing its attributes and optionally for the other levels.

The Bind class is required for modeling interface bindings according to definition of the BindingController interface. All interface bindings are represented by this class. Similarly to other Fractal metamodel elements, even the components that do not implement BindingController interface (and thus there are not $X.1$ components) can use the Bind class. For components implementing this control interface it is a mandatory part of their metamodel, for the other it is a optional part.

3.2.3 Level 1

At this conformance level, component reflection capabilities are required. Every component has to provide access to its interfaces and simple type information at runtime. Moreover, the component type must be comparable with the type of another component by the subtype relation. The decision if a component type is a subtype of another component type can not be done in a universal way in the Fractal metamodel, because Level 1 components are not forced to use the Fractal type system, and thus the subtype relation is not constrained by any rules from the Fractal component model specification. In general, the subtype relation is specific for each particular Fractal-compliant component system.

It is not necessary to alter the level 0 metamodel (see Figure 3.15), as it already holds all required data to satisfy the needs of implementation of reflection capabilities required from level 1 components. The ComponentInterface association supports the implementation of obtaining the component interfaces and the Type class represents the component type. Equally to level 0, the UnknownType singleton instance of the Type class represents type for all level 1 components. The isSubtypeOf operation of the Type class is left unimplemented; for using the metamodel for a particular level 1 Fractal component system, the Type class has to be extended and the component system specific isSubtypeOf operation has to be implemented.

3.2.4 Level 2

With conforming to Fractal at level 2, the reflection capabilities are required not only from components, but also from their interfaces. Component interfaces must provide values of its basic properties, reference to the owner component, and the simple type information. The requirements for the interface typing are the same as for level 1 component type — implementation of the isSubtypeOf operation. Analogically to level 1, the level 2 components do not have to use the Fractal type system and can use its own. Hence, the isSubtypeOf operation is left unspecified in the common Fractal metamodel and ready for implementation in a particular Fractal component system.

The level 2 Fractal component systems can use the same metamodel as the levels 0 and 1 do. It contains all necessary data to support the interface introspection. For using the metamodel for a particular Fractal level 2 component system, extension of the Type class has to be developed with the implementation of the isSubtypeOf method.

3.2.5 Level 3

Level 3 components must support the Fractal type system. For the type information representation the ComponentType and InterfaceType classes are used (subtypes of the Type with implemented isSubtypeOf operation). Structure of the InterfaceType follows the definition of the Interface interface. The component type is defined as a set of the component interface types and thus it can be derived from the ComponentInterface association and from the InterfaceType. The ComponentType class is present in the Fractal metamodel because of the simplification of metamodel extension; it is not only an extension point for customization of the Fractal type system, but also utilizes using of the metadata repository.

3.2.6 Levels 3.2 and 3.3

The conformation to Fractal at these levels does not affect the metamodel structure. It just requires presence of two factory interfaces and prescribes the semantics of them. These interfaces can be modeled similarly to another control interface, as was described for levels $X.1$.

3.2.7 Usage of the Implementation and the Controller class

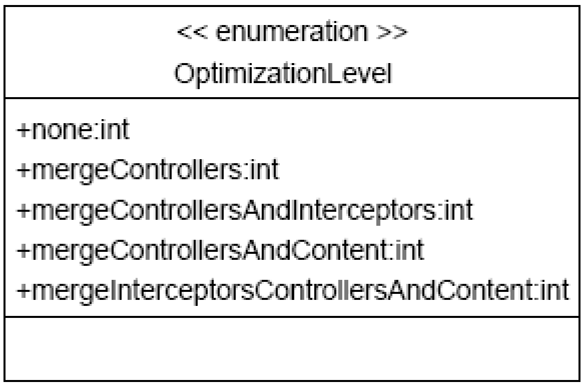
In previous overview that discusses the usage of the Fractal metamodel, there is no mention about the Implementation and the Controller class. It is necessary to distinguish between describing architecture and describing a particular implementation of a software system. The component together with its interfaces describes the architecture of the system. This architecture can be then implemented in various ways using different control abilities. For all conformance levels the Implementation class can be used to represent a particular implementation of the component, likewise the Controller class stores the information about the particular component controller.

3.3 Julia metamodel

Metamodel presented in this section is an extension of the Fractal metamodel for Julia, which is a Java implementation of 3.3 level Fractal system. Similar to the Fractal metamodel, particular parts of the Julia metamodel are describe in the following sections. The whole Julia metamodel (as an extension of Fractal metamodel) can be found in appendix A.

3.3.1 Julia components

One of the Julia goals is to provide a support for choosing between highly configurable components and less configurable, but very efficient ones. User can make the speed/memory tradeoffs he or she want. For this purpose Julia use class generators to merge classes and remove the indirections when invoking methods. Figure 3.16 illustrates optimization levels supported by Julia.



<< enumeration >>
OptimizationLevel
+none:int
+mergeControllers:int
+mergeControllersAndInterceptors:int
+mergeControllersAndContent:int
+mergeInterceptorsControllersAndContent:int

Figure 3.16: Optimization levels

Each Julia component has its optimization level and each optimization level have its own class generator for merging class. Incoming and outgoing methods calls on functional interface can be intercepted by so called *interceptors* to do for example some conversions or to add exception handling specific for a particular environment. These interceptors are generated automatically by a class generator.

As it was mentioned above, Julia conforms at level 3.3 to the Fractal component model, hence all component interfaces must be castable to the Interface interface. In Julia, support of this interface is generated automatically by a class generator for all component interfaces. Figure 3.17 depicts the resulting metamodel for Julia components.

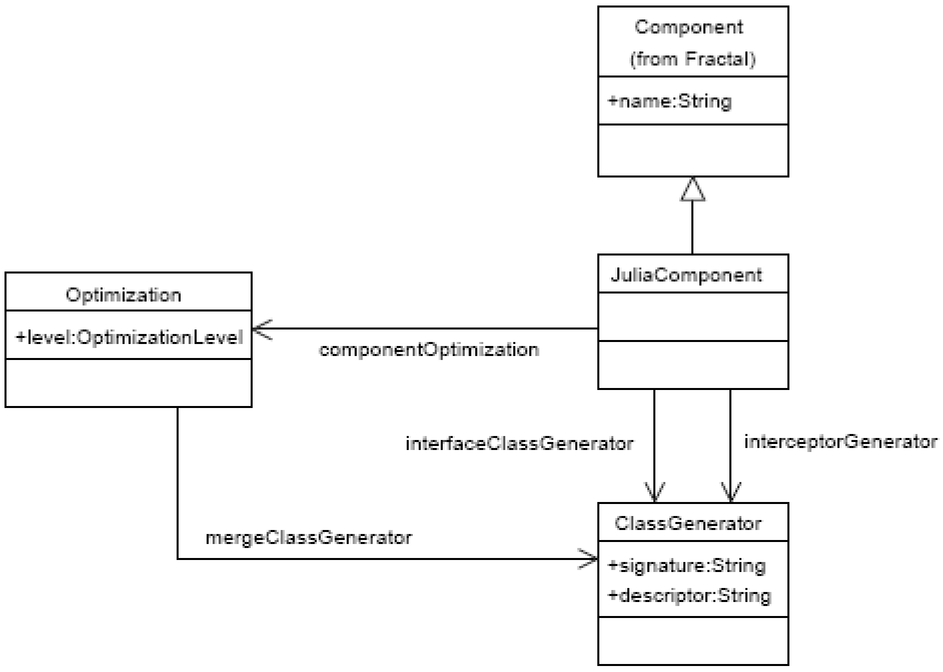


Figure 3.17: Julia component

3.3.2 Interfaces

Each component interface is represented by its own Java object. It is because of the requirement to implement both the Interface interface and the Java interface corresponding to this interface (getFcItfName() method of the Interface implementation has to return interface name and thus distinct implementation for all component interfaces has to exist). Figure 3.18 shows the relation of the Julia interface to its implementation.

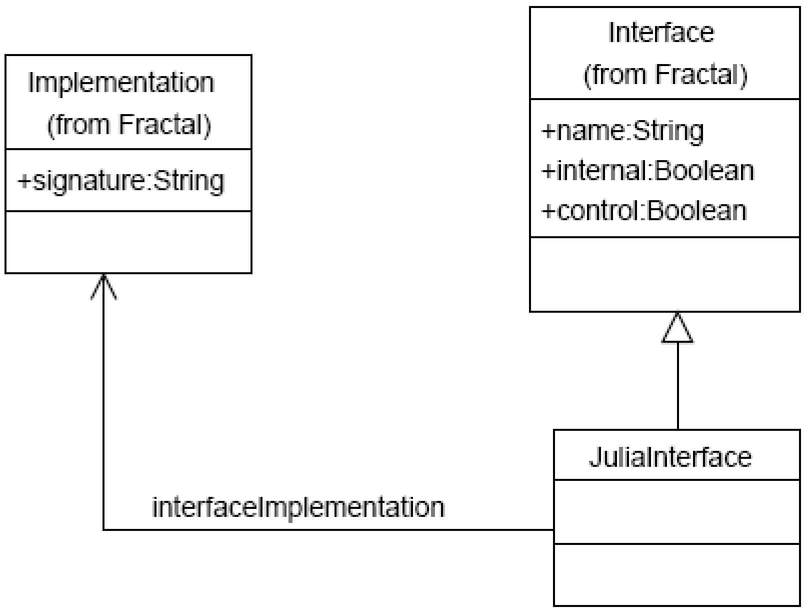


Figure 3.18: Julia interface

3.3.3 Control objects

The main goal of Julia is to implement a framework to program component controllers. It provides an extensible set of control objects, from which the user can freely choose and assemble controller part of component. Julia use *mixin classes* for this purpose. Mixin classes can be mixed, resulting in normal classes. Figure 3.19 captures the mixin concept and relates it to the Fractal metamodel.

3.4 Implementation

Together with the creation of Fractal metamodel, we develop a basic support for developing Fractal component systems, using the MDA approach to software development. We implemented a MOF-based repository for managing Fractal metadata together with loading of Fractal metadata specified via Fractal ADL and storing it to the repository. Entire implementation is written in Java and particular parts of it are designed as Fractal components.

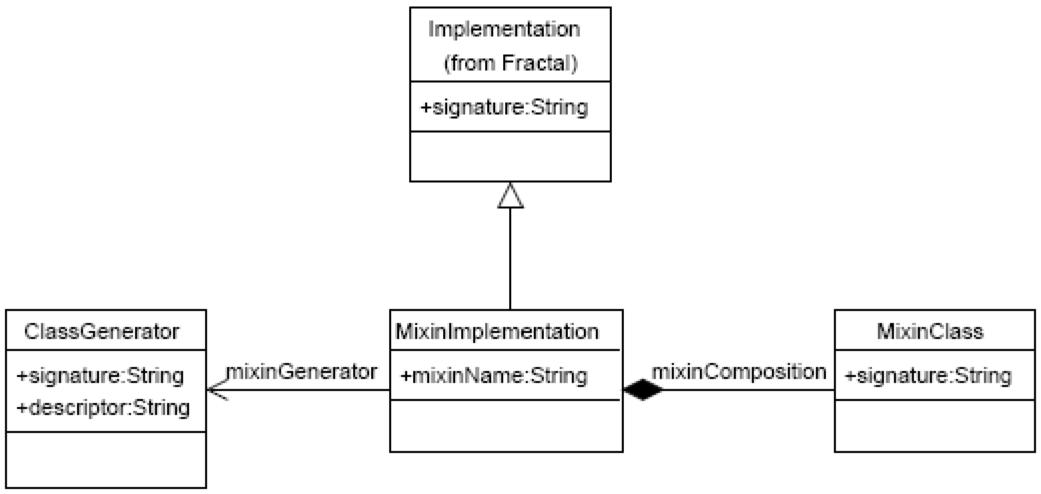


Figure 3.19: Julia mixin implementation

The attached CD contains the above mentioned implementation together with a demonstration of the repository usage. The installation instruction for runnig the example are described in Appendix B.

3.4.1 Metadata repository

Repository was generated from MOF metamodel using the generation service of Metadata repository (MDR) [11], which generates repository accessible through the Java interfaces compliant with the Java Metadata Interface (JMI) [9]. With regard to mentioned similarity of MOF and UML metamodel, the MOF metamodel was prepared in the Poseidon for UML[20] modeling tool. Then, it was converted via a MDR command-line tool uml2mof from the UML to the MOF metamodel. The process of repository creation is illustrated in Figure 3.20.

Repository generated by the MDR can be accessed using both reflective (see Section 2.2), or metamodel specific (generated) APIs. For each class in the Fractal metamodel MDR generates two interfaces: one for creating new instances of given class; and the second one representing the instances of that class with the getter and setter methods for class attributes. The second interface also contains methods for navigating according to class associations

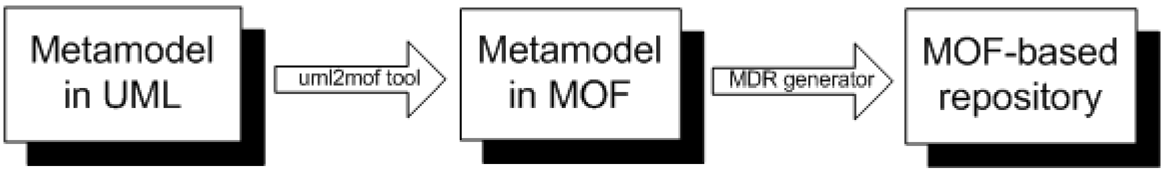


Figure 3.20: Repository implementation

(for association's ends that are navigable). There is also generated an interface for each association. The interface allows creating, removing and testing of existence of association instances between class instances. Together with these interfaces their implementation is also generated by the MDR. The repository is accessed through the package interface created for every package. It offers methods for obtaining the interfaces for creating new instances of metamodel elements.

For example for the class Component from the Fractal metamodel two interfaces — ComponentClass and Component — are generated. The ComponentClass interface allows creation of new component instances, while the Component interface serves for setting of component attributes and navigating between associated classes. The Component interface contains methods setName(String), getInterfaces() etc. The interface for association between the Component and the Interface classes involve methods add(Component, Interface), exists(Component, Interface) etc. Implementation of the ComponentClass interface is obtained by invoking getComponentClass() on Fractal-Package object.

However the generated API for accessing repository is not intuitive much for those unfamiliar with the JMI, and also requires detailed knowledge of the Fractal metamodel to use it properly. For example, if you want to add an interface to a component properly, sequence of methods has to be called (see bellow).

```
Interface iface = fractalPackage.getInterface().create("ifcName");
iface.setInterfaceOwner(component);
fractalPackage.getComponentInterface().add(component, iface);
UnknownType ut = fractalPackage.getUnknownType().createUnknownType();
iface.setType(ut);
fractalPackage().getIsOfType().add(iface, ut);
```

To provide more comfortable access to repository, we developed a *facade*[1] interface for the repository. It hides the complexity of the JMI and offers the intuitive interface of the repository. Instead of writing multiple line of code, the interface can be simply added by calling one method from the facade interface:

```
facade.addInterface("ifcName", component, additionalParams);
```

Because of the equality of the Fractal metamodel core structures, the facade interface is common for all conformance levels. Data for the optional parts of the metamodel are passed to methods using the associative array with additional parameters (a parameter name associated with a parameter value). Having the additional parameters in the associative array enables using the repository facade for the Fractal metamodel extensions.

We have implemented the facade interface for all Fractal conformance levels, i.e. levels 0,1,2 and 3. The implementation for sub levels such as 0.1, 1.1 etc. is not necessary because the differences between levels and its sub levels is only in necessity of implement some control interfaces. Repository facade for a particular component level stores the metadata according to requirements for a given level.

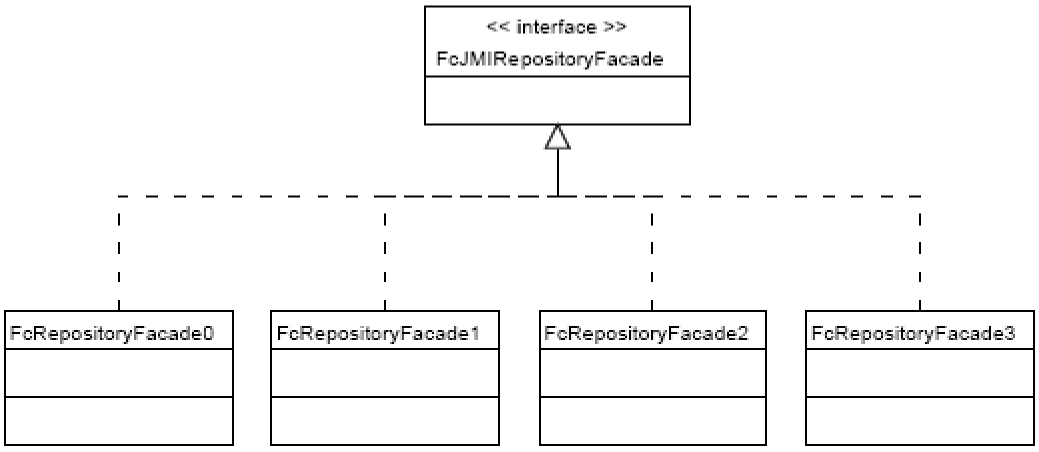


Figure 3.21: Repository facades

[1]Facade is a design pattern from [5]

The facade for Julia metadata repository is an extension of the Fractal repository facade. Implementation of the facade interface for Fractal level 3 is reused by inheriting it by JuliaRepositoryFacade.

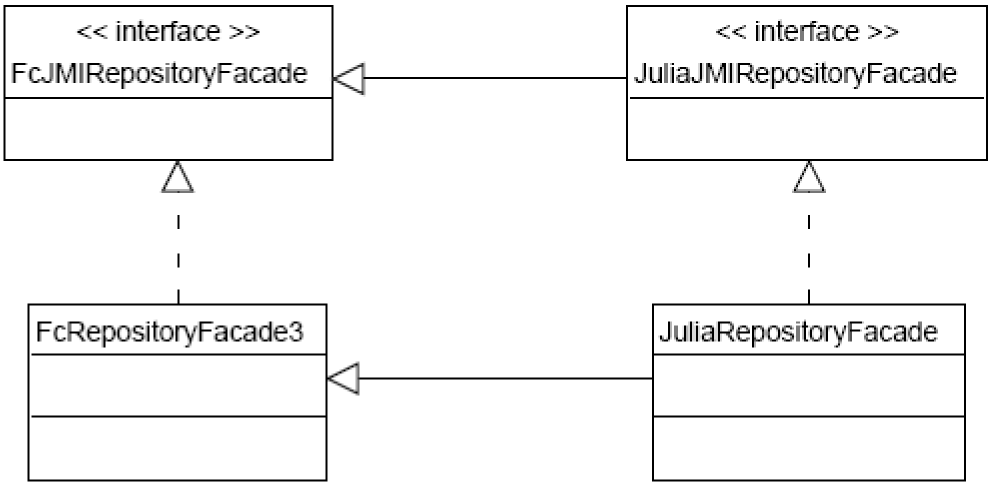


Figure 3.22: Julia repository facade

3.4.2 ADL loading

Implementation of the metadata loading from the Fractal ADL to the MOF-based repository fully benefits from the current Fractal implementation. We have implemented the loading as a Fractal composite component with the usage of existing Fractal components. Data from XML-based Fractal ADL are loaded using the ADLLoader component, which is a composite component from the standard Fractal distribution. The loader component is used as a member of the composite component that performs loading. Such a reuse of an existing Fractal component as a building block for a new composite component is an illustration of the Fractal component model strength and benefits for software development.

Architecture of composite component for loading ADL to JMI repository is depicted in Figure 3.23. Data loaded from the ADL using the ADLLoader component are processed by the RepositoryWriter component. The writer does not work with the repository directly, but use the RepositoryFacade component for accessing the JMI repository.

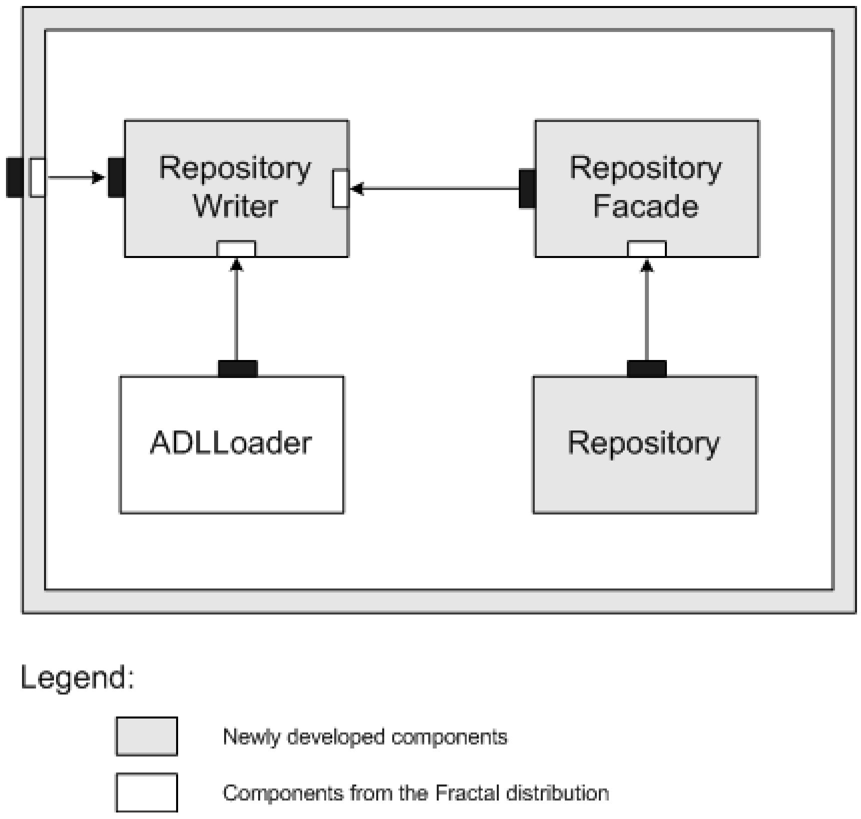

Repository Writer
Repository Facade
ADLLoader
Repository
Legend:
Newly developed components
Components from the Fractal distribution

Figure 3.23: Design of ADL loading

3.5 MDA development with the proposed metamodel

Having the MOF-compliant Fractal metamodel together with the metadata repository, and the implementation of transformation from the Fractal ADL to MOF, developers can benefit from MDA-based development of Fractal applications and can use tools supporting MDA (see 4).

Transformations between the Fractalmetamodel and metamodels of another platforms can be developed and used for guiding the transformations between PIM and PSM, as it was described in Section 2.1.6.

Chapter 4

Related work

In paper[4], the importance of a metadata management in complex software systems (such as middleware platforms) is discussed.

The paper [18] compares two approaches to building a metadata repository for component-based distributed environment. It compares a hand-written repository and MOF-based one with respect to easiness of building and accessing them. Authors of this paper recommend using MOF-based repositories, because it allows faster development, better extensibility and it also complies with the OMG standards. The results are based on implementations of the both repositories for a real component model.

The Fractal concepts are implemented in several other projects: Think[19] is a component-based programming framework for building modular and customizable operating systems and applications, which can be considered as an implementation of Fractal component model that enables component-based programming in the C language; ProActive[21] is a Fractal level 3.2 GRID Java library for parallel, distributed, and concurrent computing; SOFA[23] is Fractal level 2 compliant componet system which allows for an application to be composed of a set of dynamically hierarchical updatable components.

AndroMDA[1] is a generation framework that follows the MDA paradigm. It enables the developer to create an UML model of the application in a CASE-tool and then generates classes and deployable components. The resulting files are directly deployable; all the developer has to do is to implement the application specific business logic. AndroMDA comes with templates prepared for generating J2EE web applications from UML models; it generates EJB components, Struts classes and HTML pages. But AndroMDA is not limited to EJB-based applications only. It provides mechanisms for creating templates and subsystems to generate applications for other platforms than J2EE.

Jamda[8] is a similar project to AndroMDA, it is an open-source framework for building application generators which create Java code from a model of the business domain. An application generator build using Jamda would perform the role of a model compiler in the Object Management Group's Model Driven Architecture specification.

Chapter 5

Conclusion and future work

The goal of this thesis was to design a metamodel applicable for describing component systems conforming to the Fractal component model at an arbitrary level. The proposed metamodel satisfies this requirement and allows an easy extensibility of the metamodel for modeling arbitrary features of a particular Fractal component system.

As a proof of the concept, the MOF-based metadata repository was developed together with the metamodel. It provides a basic support for implementing Fractal components using the MDA approach to software development. The facade interface allows an intuitive use of the repository. At this time, the facade for adding objects to the repository exists. The future implementation of a facade interface for navigating and updating data in the repository would enable more comfortable usage of the repository.

The metadata loading from the Fractal ADL to the MOF-based repository enables using both the Fractal metadata tools and the standardized MOF-based tools for managing the metadata. It also allows using the application from the Fractal GUI project[1] to design the architecture of Fractal systems and then to translate them to a metamodels conforming to the MDA.

Implementation of the reverse metadata transformation, i.e. transformation from the Fractal MOF-based repository to the Fractal ADL, could be a task of a future work.

[1]Fracral GUI is one of the OMG's Fractal projects. It provides a graphical tool for editing Fractal component configurations.

In this thesis the metamodel was designed for the reference implementation of the Fractal component model Julia. The possible future work is the creation of the Fractal metamodel extensions for another Fractal-based systems such as ProActive[21], Think[19], SOFA[23], or FROGi[6] to enable MDA-based development of their applications.

Bibliography

[1] *AndroMDA*, http://www.andromda.org/

[2] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J. B.: *An Open Component Model and Its Support in Java*, Proceedings of CBSE 2004, Edinburgh, May 2004

[3] Brunneton, E., Coupaye, T., Stefani, J.B.: *Recursive and Dynamic Software Composition with Sharing*, Proceedings of WCOP'02, Malaga, Spain, June 2002

[4] Costa, F., Blair, G.S.: *The Role of Meta-Information Management in Reflective Middleware*, Proceedings of the ECOOP' 00 Workshop on Workshop on Reflection and Metalevel Architectures, Sophia Antipolis and Canes, France, Springer-Verlag, June 2000.

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, March 1995

[6] *FROGi*, http://www-adele.imag.fr/frogi/

[7] France Telecom: *Julia documentation*, http://fractal.objectweb.org/current/doc/javadoc/julia/overview-summary.html

[8] *Jamda*, http://jamda.sourceforge.net

[9] *Java Metadata Interface*, JSR-40, http://java.sun.com/products/jmi/

[10] Lopes C. V. and Hursch W. L.: *Separation of Concerns*, College of Computer Science, Northeastern University, Boston, February 1995, http://citeseer.ist.psu.edu/lopes95separation.html

[11] *Metadata Repository*, http://mdr.netbeans.org/

[12] OMG: CORBA Components, version 3.0, OMG document formal/02-06-65

[13] OMG: *Common Object Request Broker Architecture*, http://www.omg.org/technology/documents/formal/corba_iiop.htm

[14] OMG: *MDA Guide*, version 1.0.1, OMG document omg/03-06-01

[15] OMG: *Meta Object Facility Specification*, version 1.4, OMG document formal/02-04-03

[16] OMG: *The Fractal Component Model*, version 2.0-3

[17] OMG: *XML Metadata Interchange Specification*, version 1.2, OMG document formal/02-01-01

[18] Petr Hnětynka, Michal Píše: *Hand-written vs. MOF-based Metadadata Repositories: The SOFA Experience*, Proceedings of ECBS 2004, Brno, Czech Republic, IEEE CS, ISBN 0-7695-2125-8, pp. 329-336, May 2004

[19] J.-Ph. Fassino, J.-B. Stefani, J. Lawall, G. Muller. *THINK: A Software Framework for Component-based Operating System Kernels*, in: Proceedings of Usenix Annual Technical Conference, Monterey (USA), June 10th-15th 2002

[20] *Posseidon for UML* tool, http://www.gentleware.com/

[21] *ProActive*, http://www-sop.inria.fr/oasis/ProActive/

[22] Sun Microsystems: *Java 2 Enterprise Edition (J2EE)*, http://java.sun.com/j2ee/

[23] Plasil F., Balek D., Janecek R.: *SOFA/DCUP, Architecture for Component Trading and Dynamic Updating*, Proceedings of ICCDS 1998, Annapolis, USA, IEEE CS, 1998.

Appendix A

The Fractal and Julia metamodels

The following figures depict the Fractal and Julia metamodels from which the MOF-based repository is generated. All elements of the metamodels have been described in Chapter 3.

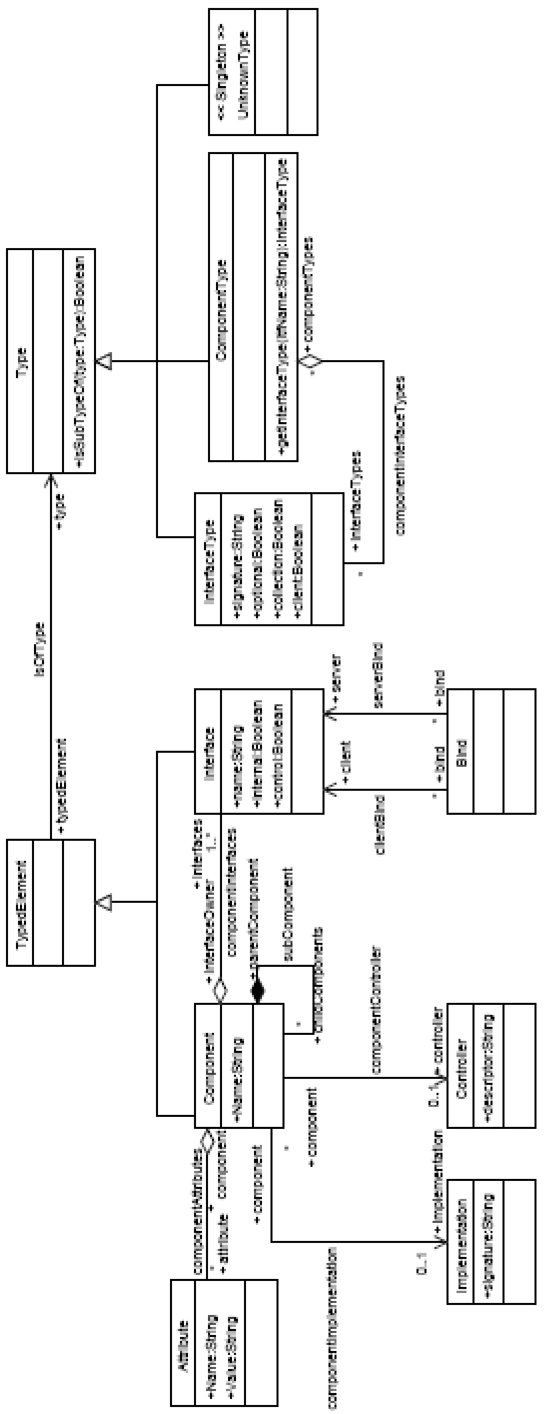


Figure A.1: The Fractal metamodel

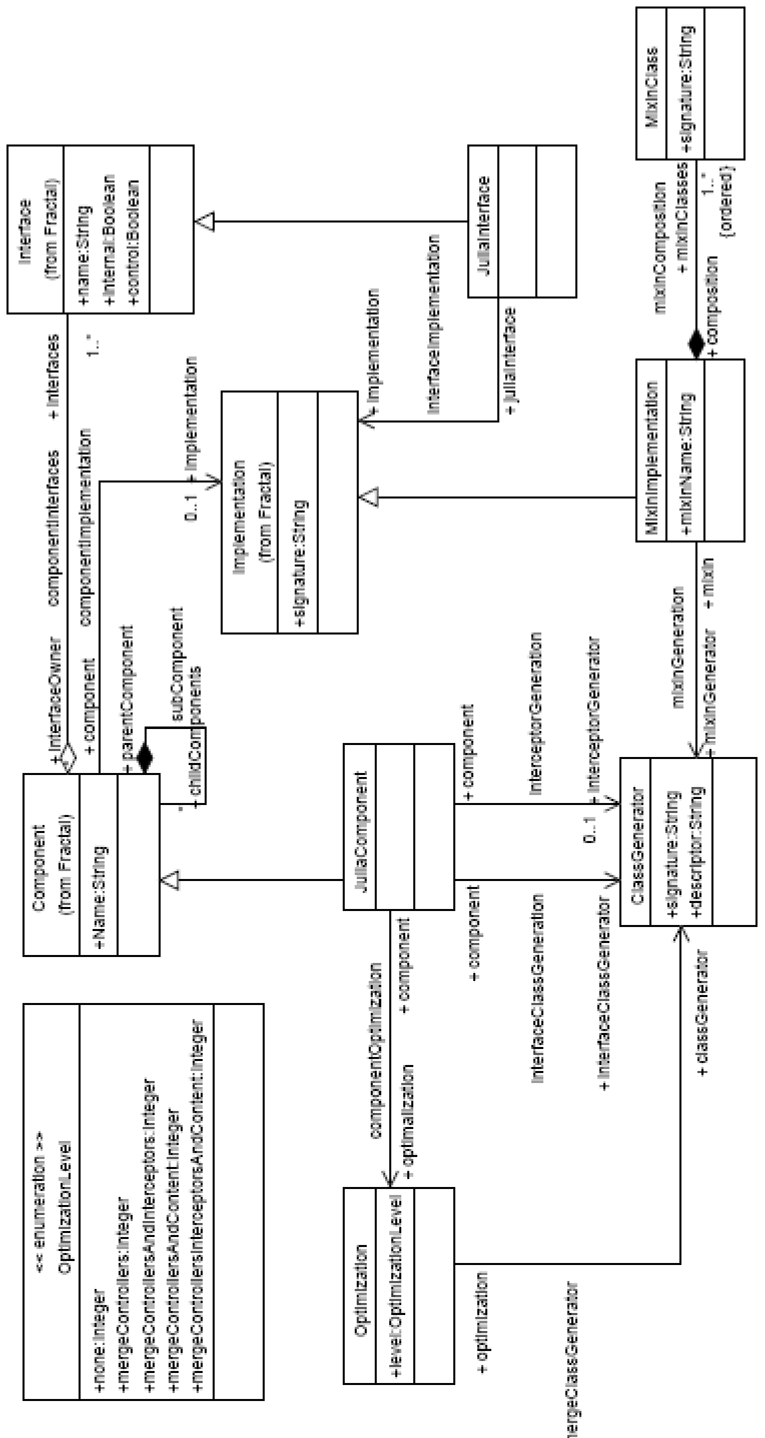


Figure A.2: The Julia metamodel

Appendix B

Installation instructions

The attached CD contains four directories:

- text contains the text of this thesis,
- src contains source codes of the implementation described in Section 3.4 (i.e., Fractal metadata repository and the loading of metadata from ADL to the repository),
- build contains build of the application and the example of a Fractal ADL definition and its equivalent in XMI,
- install contains Linux and Windows self-installing executables of the demo application, i.e. loading of an ADL definition to repository and them printing the repository content to a file.

B.1 Installing the demo application

Running of the demo application requires Java Runtime Environment at least at version 1.5. In Linux, use the install-linux.bin installer, Windows installation is performed using the install-windows.exe executable. For installing the demo, follow the instructions of the installer.

B.2 Running of the demo application

The executables of the demo are located in the bin subdirectory of the installation destination folder. In case of the Linux installation, the name of the demo script is runDemo.sh, in Windows, it is runDemo.bat. The script expects two parameters: the fully qualified name of an Fractal ADL definition (without the .fractal extension) and the name of the output XMI file. The ADL name is specified similarly to names of Java classes — a sequence of identifiers separated by the "." token (e.g., org.objectweb.fractal.adl.BasicCompiler). The name is relative to a classpath folder of the demo script (e.g., the build directory). The metadata are loaded from ADL to the repository and then the content of the repository is written to the output file.

Running the demo script without any parameters performs the transformation of the HelloWorld ADL definition, which is located in build subdirectory of the installation:

```
runDemo.sh
```

Example of running the metadata transformation of a component from the Fractal distribution:

```
runDemo.sh org.objectweb.fractal.adl.BasicCompiler BasicCompiler.xmi
```

It is also possible to rebuild (compile) the application using the compile.sh (or complile.bat) script.