Adlet: A Java-based Architecture Description Language

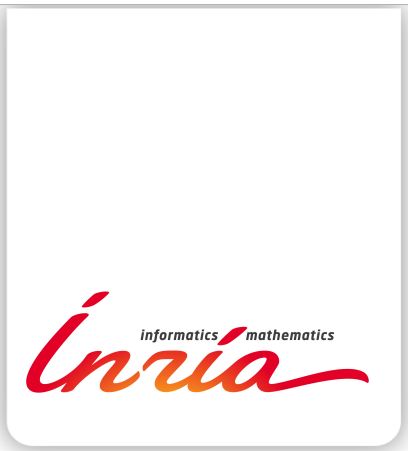
Lionel Seinturier

Adlet: A Java-based Architecture Description Language

Lionel Seinturier

RESEARCH REPORT
N° 9242
December 2018
Project-Teams Spirals

ISSN 0249-6399 ISRN INRIA/RR--9242--FR+ENG



Adlet: A Java-based Architecture Description Language

Lionel Seinturier

Project-Teams Spirals

Research Report n° 9242 — December 2018 — 16 pages

Abstract: Adlet is a Java-based architecture description language (ADL). Adlet enables to describe in pure Java syntax a software architecture. Two main patterns are supported: composite component, and typed binding. The composite component pattern enables to describe a hierarchy of parent and sub components. Typed bindings enable to describe the communication links between provided and required component interfaces. These two patterns are implemented in pure Java syntax. This means that an Adlet software architecture descriptor is a fully legal Java program with no particular language extension. This enables reusing the full stack of tools (unit testing, processing, editing, compiling, etc.) available for this language. As such Adlet can be seen a programming style for describing software architectures with the Java language.

Key-words: software, architecture, component, Java, Fractal.

RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Adlet : un language de description d'architecture en Java

Résumé : Adlet est un langage de description d'architecture (ADL) en Java. Adlet permet de décrire une architecture logicielle dans une syntaxe purement Java. Deux gabarits principaux sont supportés: composant composite, et liaison typée. Le gabarit composant composite permet de décrier une hiérarchie de composants parents et enfants. Les liaisons typées permettent de décrire les liens de communication entre les interfaces fournies et requises d'un composant. Ces deux gabarits sont mis en œuvre avec une syntaxe purement Java. Cela signifie qu'un descripteur Adlet d'architecture logicielle est un programme purement Java sans extension langage particulière. Cela permet d'utiliser la pile complète d'outils (test unitaire, traitement, édition, compilation, etc.) disponible pour ce langage. En tant que tel Adlet peut être vu comme un style de programmation pour décrire des architectures logicielles en Java.

Mots-clés : logiciel, architecture, composant, Java, Fractal.

1 Introduction

Architecture Description Languages (ADL) [5] are languages for describing the overall structure and interconnection between elements in a software system. Adlet is an ADL for the Fractal component framework [3]. Adlet extends the existing Fraclet annotation library [6] with support for composite components and typed bindings.

Sections 2 and 3 present respectively the notions of atomic and composite components. The notion of a typed binding is introduced in Section 4. Section 5 reports about the metamodel generation process. Typing rules are detailed in Section 6. Section 7 reports about additional functionalities defined by Adlet. The example of the Comanche web server programmed with Adlet is presented in Section 8. Finally, section 9 concludes this document.

2 Atomic Component

Atomic components are leaves in a component hierarchy. With Adlet a component is defined by a `@Component`-annotated class.

The structure of a component is composed of two main elements: provided interfaces, and required interfaces. A provided interface defines the services exported by a component. A required interface defines the services needed by the component during its functioning. Both required and provided interfaces are defined with annotations.

Figure 1 presents a code snippet with the example of the `HelloWorld` component that declares one provided interface (parameter `provides` associated to the `@Component` annotation) named `r` and whose signature is defined by the `Runnable` Java interface, and one required interface (annotation `@Requires` associated to the field `s`) named `s` and whose signature is defined by the `PrinterItf` Java interface. Components can declare as many provided and required interfaces as needed.

```
1 @Component(provides=@Interface(name="r",signature=Runnable.class))
2 class HelloWorld implements Runnable {
3   @Requires PrinterItf s;
4   // ...
5 }
```

Figure 1: Definition of an atomic component.

Provided and required interfaces are characterized with a so-called contingency that can take two values: mandatory or optional. An interface is said to be of contingency optional if this is not mandatory that the interface be bound for the system to function. Such an interface corresponds typically to an non-vital service. Conversely, an interface is said to be of contingency mandatory if this is mandatory that the interface be bound for the system to function. In other words, the system will not function if such an interface is not bound.

3 Composite Component

The Composite pattern [4] enables to compose elements in a tree structure. With Adlet, a composite component encapsulates one or several subcomponents that can themselves be composite or atomic components. The hierarchy of parent and sub components enables to explicitly characterize the decomposition of a large software system into smaller subsystems.

Adlet provides two alternative language constructs for declaring subcomponents: field and nested class. In both cases, the field and the nested class are annonated with `@Component`. Figure 2 illustrates the declaration of the `HelloWorld` composite components with two subcomponents: `client` of type `ClientImpl` (line 4) and `Container` (line 6).

4 Typed Binding

A binding is a communication link between two interfaces. Three kinds of bindings can be declared: normal, export, and import. Not all bindings between any pairs of interfaces can be defined. To be

```
1 @Component(provides=...))
2 class HelloWorld {
3
4   @Component ClientImpl client;
5
6   @Component(provides=...) static class Container { ... }
7 }
```

Figure 2: Definition of a composite component with two subcomponents.

legal a binding must enforce some typing rules, hence the name typed binding. The typing rules follow.

Rule 1: Normal binding. A normal binding is a binding between a required interface and a provided one. In this case, the two bound components need to be direct subcomponents of the same parent component. This means that a binding cannot cross the boundaries of a parent component, and that a binding binds components that belong to the same level of a component hierarchy.

Rule 2: Export binding. An export binding corresponds to the case where a subcomponent exports a provided interface to its parent component. The interface is then also provided by the parent component. An export binding is then a binding between an interface provided by a parent component and an interface provided by a direct subcomponent of this parent component.

Rule 3: Import binding. An import binding corresponds to the case where a parent component imports a required interface from one of its direct subcomponents. The interface is then also required by the subcomponent.

Rule 4: Services inclusion. The target interface of a binding needs to provide at least as many service operations as are required by the source interface of the binding. In terms of the Java programming language for a normal binding, this means that the Java type of the target/provided interface must be a subtype of the Java type of the source/required interface.

Rule 5: Contingency. The source and target interfaces of a binding can be either optional (0) or mandatory (M). This leads to four case for the pairs of interfaces: O-O, O-M, M-O, M-M. The three cases O-O, O-M, and M-M, correspond to legal bindings. The idea is that two interfaces with the same contingency, either optional, or mandatory, are compatible and can be bound. Also, if a required interface is optional, binding it to a mandatory provided interface will of course lead to a functioning system. Nevertheless, binding a mandatory required interface to an optional provided one should not be considered. Indeed, since the provided one is optional, no guarantee can be obtained regarding the fact that the corresponding services will effectively be provided. Then, since the required mandatory interface needs an implementation, the functioning of the system would not be guarantee.

Figure 3 illustrates the declaration with Adlet of source and target interfaces that can be used to define the three types of bindings. A normal binding can be defined between the required `normal_source` required interface of component `Sub1` and the provided `normal_target` provided interface of component `Sub2`. An export binding can be defined between the provided interface `export_source` of component `Parent` and the provided interface `export_target` of component `Sub1`. An import binding can be defined between the required interface `import_source` of component `Sub2` and the required interface `import_target` of component `Parent`.

The previous paragraphs defined in English the rules associated with bindings. The idea with Adlet is to provide an encoding of these rules in pure Java. That is, we want to be able to let the Java type system ensure that only the bindings enforcing the rules can be defined, and report a typing error if this is not the case. The advantage of this solution compared to other approaches that rely on language extensions or domain specific languages, is that we do not need to provide a specific implementation of the type checker. The checking comes for free with the Java type checker.

The next section describes the metamodel-based solution that we designed to support the encoding of the binding rules.

```
1  @Component(provides=@Interface(name="export_source",signature=Runnable.class))
2  class Parent implements Runnable {
3
4    @Component(provides=@Interface(name="export_target",signature=Runnable.class))
5    static class Sub1 implements Runnable {
6      @Requires Runnable normal_source;
7      // ...
8    }
9
10   @Component(provides=@Interface(name="normal_target",signature=Runnable.class))
11   static class Sub2 implements Runnable {
12     @Requires Runnable import_source;
13     // ...
14   }
15
16   @Requires Runnable import_target;
17   // ...
18 }
```

Figure 3: Source and target interfaces for normal, export, and import bindings.

5 Metamodel Generation

For each component, Adlet generates a so-called metamodel. This metamodel reifies two architectural elements: interfaces and component hierarchy. These pieces of information are then used to define bindings. These pieces of information of course already exist in an Adlet program, but cannot be accessed and manipulated as data. The metamodel enables such a feature. The metamodel class is updated each time the source class changes.

Figure 4 extends the code example of Figure 3. The `@StaticMetamodel` annotation (line 2) declares the link with the metamodel class, here `Parent_`. This metamodel class defines six fields, one for each of the required and provided interfaces of the three components, `Parent`, `Sub1`, and `Sub2`. The naming of these fields follows the nesting of components in the hierarchy. For example, the `import_source` required interface of component `Sub2` is named `Parent_.Sub2.import_source`.

```
1  @Component(provides=@Interface(name="export_source",signature=Runnable.class))
2  @StaticMetamodel("Parent_")
3  class Parent implements Runnable {
4
5    @Component(provides=@Interface(name="export_target",signature=Runnable.class))
6    static class Sub1 implements Runnable {
7      @Requires Runnable normal_source;
8      // ...
9    }
10
11   @Component(provides=@Interface(name="normal_target",signature=Runnable.class))
12   static class Sub2 implements Runnable {
13     @Requires Runnable import_source;
14     // ...
15   }
16
17   @Requires Runnable import_target;
18   // ...
19
20   @Binding
21   public static Binder binder() {
22     Binder b = new Binder();
23     b.export(Parent_.export_source, Parent_.Sub1.export_target);
24     b.normal(Parent_.Sub1.normal_source, Parent_.Sub2.normal_target);
25     b.impor(Parent_.Sub2.import_source, Parent_.import_target);
26     return b;
27   }
28 }
```

Figure 4: Binding definitions.

In Figure 4, lines 23, 24, and 25, declare three bindings with the methods, respectively,

`export`, `normal`, and `impor`: an export binding between interfaces `Parent_.export_source` and `Parent_.Sub1.export_target`, a normal binding between `Parent_.Sub1.normal_source` and `Parent_.Sub2,normal_target`, and an import binding between `Parent_.Sub2.import_source` and `Parent_.import_target`.

Figure 5 presents a part of the code generated for the `Parent_` metamodel class. The six fields mentioned above, each corresponding to an interface, for defining the three bindings, are illustrated. The complete definition of the fields are provided in Section 6.

```
1  class Parent_ {
2    static Provided export_source = ...;
3    static class Sub1 {
4      static Provided export_target = ...;
5      static Required normal_source = ...;
6      static class _Bottom extends Parent_{}
7    }
8    static class Sub2 {
9      static Provided normal_target = ...;
10     static Required import_source = ...;
11     static class _Bottom extends Parent_{}
12   }
13   static Required import_target = ...;
14   static class _Bottom {}
15 }
```

Figure 5: Generated metamodel for the program of Figure 4.

6 Typing Rules

The definition of the Adlet typing rules rely on the following four Java types (`Required`, `Provided`, `Optional`, `Mandatory`), and three Java methods (`normal`, `export`, `import`). The four Java types take the form of Java interfaces with generic parameters.

Definition 6.1 *Let* `Required` *be the Java type of required component interfaces.* `Required` *is defined as:*

```
interface Required<TYPE,PARENT,CURRENT,BOTTOM,CTGY> {}
```

where:

- `TYPE` is the Java type of the current required component interface,
- `PARENT` is the Java type of the parent component containing the component that defines the current required component interface,
- `CURRENT` is the Java type of the component that defines this required component interface,
- `BOTTOM` is a subtype of `PARENT` that is generated and defined in the metamodel,
- `CTGY` is the contingency of this required component interface.

Definition 6.2 *Let* `Provided` *be the Java type of provided component interfaces.* `Provided` *is defined as:*

```
interface Provided<TYPE,PARENT,CURRENT,BOTTOM,CTGY> {}
```

where the five generic parameters `TYPE`, `PARENT`, `CURRENT`, `BOTTOM`, and `CTGY`, follow the same definition as for the `Required` type.

Definition 6.3 *Let* `Optional` *and* `Mandatory` *be the types for, respectively, mandatory and optional component interfaces. These types are used in relation with the* `CTGY` *generic parameter mentioned above. The definition of* `Optional` *and* `Mandatory` *follows.*

```
interface Optional {}
interface Mandatory extends Optional {}
```

Definition 6.4 *Let* `normal` *be the Java method for defining normal bindings.* `normal` *is defined as follows:*

```
<RTYPE, PTYPE extends RTYPE, PARENT, RCTGY, PCTGY extends RCTGY>
void normal(
  Required<RTYPE,PARENT,?,?,RCTGY> r,
  Provided<PTYPE,PARENT,?,?,PCTGY> p
)
```

As specified in this definition, a normal binding binds a required interface with a provided one (Rule 1 in Section 4). Five generic parameters are associated with the method: `RTPYE`, `PTYPE`, `PARENT`, `RCTGY`, and `PCTGY`. The definition `PTYPE extends RTYPE` ensures that the Java type of the provided interface is a subtype of the Java type of the required interface (Rule 4). The type `PARENT` that is declared, both for the required and the provided interfaces, ensures that the bound components belong to the same parent component (Rule 1). The definition `PCTGY extends RCTGY` ensures that the contingency of the provided interface is a subtype of the contingency of the required interface. Given that the contingency is, either `Optional`, or `Mandatory` that is a subtype of `Optional` (Rule 5).

Definition 6.5 *Let* `export` *be the Java method for defining export bindings.* `export` *is defined as follows:*

```
<RTYPE, PTYPE extends RTYPE, RCURRENT, PBOTTOM extends RCURRENT,
 RCTGY, PCTGY extends RCTGY>
void export(
  Provided<RTYPE,?,RCURRENT,?,RCTGY> r,
  Provided<PTYPE,?,?,PBOTTOM,PCTGY> p
)
```

As specified in this definition, an export binding binds two provided interfaces (Rule 2 in Section 4). Six generic parameters are associated with the method: `RTPYE`, `PTYPE`, `RCURRENT`, `PBOTTOM`, `RCTGY`, and `PCTGY`. The definition `PTYPE extends RTYPE` ensures that the Java type of the provided interface is a subtype of the Java type of the required interface (Rule 4). The definition `PBOTTOM extends RCURRENT` ensures that the component that defines the target interface is a direct subcomponent of the component that defines the source interface (Rule 2). As this is the case for normal bindings, the definition `PCTGY extends RCTGY` enforces Rule 5 about the contingencies of bound interfaces.

Definition 6.6 *Let* `impor` *be the Java method for defining import bindings.* `impor` *is defined as follows:*

```
<RTYPE, PTYPE extends RTYPE, RBOTTOM extends PCURRENT, PCURRENT,
 RCTGY, PCTGY extends RCTGY>
void impor(
  Required<RTYPE,?,?,RBOTTOM,RCTGY> r,
  Required<PTYPE,?,PCURRENT,?,PCTGY> p
)
```

As specified in this definition, an import binding binds two required interfaces (Rule 3 in Section 4). Six generic parameters are associated with the method: `RTPYE`, `PTYPE`, `RBOTTOM`, `PCURRENT`, `RCTGY`, and `PCTGY`. The definition `PTYPE extends RTYPE` ensures that the Java type of the provided interface is a subtype of the Java type of the required interface (Rule 4). The definition `RBOTTOM extends PCURRENT` ensures that the component that defines the source interface is a direct subcomponent of the component that defines the target interface (Rule 3). As this is the case for normal and export bindings, the definition `PCTGY extends RCTGY` enforces Rule 5 about the contingencies of bound interfaces.

Example

We now present the application of these typing rules to the example of Figure 4 and of its associated metamodel introduced in Figure 5.

Normal binding

In Figure 4, a normal binding is defined between the required interface `normal_source` (metamodel field on line 5) and the provided interface `normal_target` (metamodel field on line 9).

The definition of `normal_source` follows. The generic parameters `TYPE`, `PARENT`, `CURRENT`, `BOTTOM`, and `CTGY`, are mapped respectively to `Runnable`, `Parent_`, `Sub1`, `Sub1._Bottom`, and `Mandatory`.

```
static Required<Runnable, Parent_, Sub1, Sub1._Bottom, Mandatory>
  normal_source = new ComponentInterface<>( Sub1.class, "normal_source" );
```

The definition of `normal_target` follows. The generic parameters `TYPE`, `PARENT`, `CURRENT`, `BOTTOM`, and `CTGY`, are mapped respectively to `Runnable`, `Parent_`, `Sub1`, `Sub2._Bottom`, and `Mandatory`.

```
static Provided<Runnable, Parent_, Sub2, Sub2._Bottom, Mandatory>
  normal_target = new ComponentInterface<>( Sub2.class, "normal_target" );
```

The application of Definition 6.4 to the binding between `normal_source` and `normal_target` ensures that this is a correct normal binding. Indeed, `RTYPE` and `PTYPE` are both mapped to `Runnable` and thus `PTYPE extends RTYPE` holds; `PARENT` is mapped to `Parent_` in both cases; and finally `RTCGY` and `PCTGY` are both mapped to `Mandatory` and thus `PCTGY extends RCTGY` holds.

Export binding

In Figure 4, an export binding is defined between the provided interface `export_source` (metamodel field on line 2) and the provided interface `export_target` (metamodel field on line 4).

The definition of `export_source` follows. The generic parameters `TYPE`, `PARENT`, `CURRENT`, `BOTTOM`, and `CTGY`, are mapped respectively to `Runnable`, `Object`, `Parent_`, `Parent_._Bottom`, and `Mandatory`.

```
static Provided<Runnable, Object, Parent_, Parent_._Bottom, Mandatory>
  export_source = new ComponentInterface<>( Parent_.class, "export_source" );
```

The definition of `export_target` follows. The generic parameters `TYPE`, `PARENT`, `CURRENT`, `BOTTOM`, and `CTGY`, are mapped respectively to `Runnable`, `Parent_`, `Sub1`, `Sub1._Bottom`, and `Mandatory`.

```
static Provided<Runnable, Parent_, Sub1, Sub1._Bottom, Mandatory>
  export_target = new ComponentInterface<>( Sub1.class, "export_target" );
```

The application of Definition 6.5 to the binding between `export_source` and `export_target` ensures that this is a correct normal binding. Indeed, `RTYPE` and `PTYPE` are both mapped to `Runnable` and thus `PTYPE extends RTYPE` holds; `RCURRENT` is mapped to `Parent_`, `PBOTTOM` is mapped to `Sub1._Bottom`, `Sub1._Bottom` extends `Parent_` (see line 6 in Figure 5), and thus `PBOTTOM extends RCURRENT` holds; finally `RTCGY` and `PCTGY` are both mapped to `Mandatory` and thus `PCTGY extends RCTGY` holds.

Import binding

In Figure 4, an import binding is defined between the required interface `import_source` (metamodel field on line 10) and the required interface `import_target` (metamodel field on line 13).

The definition of `import_source` follows. The generic parameters `TYPE`, `PARENT`, `CURRENT`, `BOTTOM`, and `CTGY`, are mapped respectively to `Runnable`, `Parent_`, `Sub2`, `Sub2._Bottom`, and `Mandatory`.

```
static Required<Runnable, Parent_, Sub2, Sub2._Bottom, Mandatory>
  import_source = new ComponentInterface<>( Sub2.class, "import_source" );
```

The definition of `import_target` follows. The generic parameters `TYPE`, `PARENT`, `CURRENT`, `BOTTOM`, and `CTGY`, are mapped respectively to `Runnable`, `Object`, `Parent_`, `Parent_._Bottom`, and `Mandatory`.

```
static Required<Runnable, Object, Parent_, Parent_._Bottom, Mandatory>
  import_target = new ComponentInterface<>( Sub1.class, "import_target" );
```

The application of Definition 6.6 to the binding between `import_source` and `import_target` ensures that this is a correct import binding. Indeed, `RTYPE` and `PTYPE` are both mapped to `Runnable` and thus `PTYPE extends RTYPE` holds; `RBOTTOM` is mapped to `Sub2_._Bottom`, `PCURRENT` is mapped to `Parent_`, `Sub2._Bottom` extends `Parent_` (see line 11 in Figure 5), and thus `RBOTTOM extends PCURRENT` holds; finally `RTCGY` and `PCTGY` are both mapped to `Mandatory` and thus `PCTGY extends RCTGY` holds.

7 Additional Functionalities

This section presents seven additional functionalities provided by Adlet. The first three, cardinality, contingency, and attribute, correspond to characteristics that can be defined on components. The fourth one, component factory, provides a way to instantiate and launch components. The next two, membrane, and mixin, enable to customize the operational semantics of components. The last one, Fractal control interfaces, enable a component to access the metalevel that is composed of the controllers defined in its membrane.

7.1 Cardinality

Interfaces are by default endpoints for a point-to-point communication channel. They are then said to be associated with a so-called cardinality singleton. When multi-point communication is needed, interfaces can be associated with a so-called cardinality collection. Figure 6 illustrates such a case with the `printers` interface (line 4) of type `PrinterItf`.

```
1 @Component(provides=...))
2 @StaticMetamodel("HelloWorld_")
3 class HelloWorld {
4   @Requires(cardinality=Cardinality.COLLECTION) Map<String,PrinterItf> printers;
5 }
```

Figure 6: Definition of a collection interface.

As illustrated in Figure 6, required collection interfaces are injected in a Java field of type `Map`. The idea is to store the reference of these interfaces indexed by a unique name. This unique name is built with a prefix and a suffix. The prefix is shared by all the instances of this required collection interface, and is by default the name of the Java field, here `printers`. The suffix differs for each instance. The value of this suffix is left to the developer choice when she defines a binding that originates from this required collection interface. The metamodel provides the `setSuffix` method for defining this suffix. As an example, the following expression can be used for defining a binding that originates from the instance of the `printers` interface associated with suffix `1`: `HelloWorld_.printers.setSuffix("1")`.

7.2 Contingency

As mentioned in Section 2, interfaces can be associated with a contingency that is either mandatory of optional. By default, the contingency is optional. Figure 7 illustrates the definition of a provided optional interface (line 2, parameter `contingency`), and of a required optional interface (line 4, parameter `contingency`).

7.3 Attribute

Components can define attributes. An attribute is a piece of data exported by a component and that can, by default, be read and written. With Adlet, an attribute is a field annotated with

```
1  @Component(provides=
2    @Interface(name="r",signature=Runnable.class,contingency=Contingency.OPTIONAL))
3  class HelloWorld {
4    @Requires(contingency=Contingency.OPTIONAL) PrinterItf s;
5  }
```

Figure 7: Definition of optional interfaces.

`@Attribute`. Figure 8 illustrates the declaration of an attribute named `id` (line 3) and of type `String`.

```
1  @Component(...))
2  class HelloWorld {
3    @Attribute String id;
4  }
```

Figure 8: Definition of an attribute.

7.4 Component Factory

For every component, Adlet generates a Factory class [4] that provides methods for creating instances of the component. The name of the generated factory class is the concatenation of the component name and of the `Factory` suffix. For example, the factory class for the component `HelloWord` is named `HelloWorldFactory`.

The factory class generated by Adlet implements the interface `Factory` defined in the Fractal API. In particular, this interface defines the method `newFcInstance` that returns a new instance of the corresponding component. In addition, if the component provides an interface named `r` of type `Runnable`, the factory class defines a public static method `main`. This method instantiates the component, starts it, and invokes the method `run` of the interface `r`.

7.5 Membrane

The concept of a membrane defines the way a component is managed and executed by the Fractal framework. Fractal is an open component framework in the sense that a membrane is a fully programmable piece of code. By this way, developers can customize and define their own operational semantics for components. A Fractal membrane is defined by a name, a set of controllers, and a set of code generators for interceptors.

A controller is defined by the name of a control interface, the type of this control interface, and the class implementing this control interface. A controller implements a specific management function for the component it is associated with. Typical management functions include binding management, management of the component hierarchy (parent and child components), management of the component life cycle. This list is not restrictive, and the developer can provide her own controller for the management functions she needs. A specific controller associated with the control interface name `component` and the control interface type `Component` must be provided by every membrane. This controller defines the reference of the current component (much like `this` defines the reference of the current object in Java), and provides methods to introspect the component, especially methods to discover its interfaces. The last constituents of a membrane are the code generators for interceptors. Every interface of a component can be associated with interceptors. As their name suggests it, interceptors enable to execute code whenever a call is issued on one of the interfaces of the component. Typical interception code includes access control, concurrency management, threading policy, auditing. This list is not restrictive, and the developer can provide her own interceptor code generators for the management functions she needs. Interceptor code generators are divided in two parts: interceptor class generator, and source code generators. The first one generates the skeleton of the interceptor class, and the former ones generate blocks of code implementing the interception logic for each method of the component interfaces.

Figure 9 illustrates the definition of the primitive membrane for atomic components. A membrane definition is a `@Membrane`-annotated class (line 1). The name of the membrane is defined

by the parameter `name` (line 2). The set of controllers is the set of `@Controller`-annotated fields (line 10 to 23) of the class. Here, five controllers are defined: `component`, `bc`, `sc`, `lc`, `nc`. For example, in lines 13 and 14, the controller named `bc` is defined with the control interface name `binding-controller`, the control interface type `BindingController`, and the implementing class `ContainerBindingControllerDef`. The code generators for interceptors are defined with the parameters `interceptorClassGenerator` (line 3), and `interceptorSourceCodeGenerators` (line 4).

```
1  @Membrane(
2    name = Primitive.NAME,
3    interceptorClassGenerator = InterceptorClassGenerator.class,
4    interceptorSourceCodeGenerators = LifeCycleSourceCodeGenerator.class
5  )
6  public class Primitive {
7
8    final public static String NAME = "primitive";
9
10   @Controller(impl=ComponentControllerDef.class)
11   private Component component;
12
13   @Controller(name="binding-controller",impl=ContainerBindingControllerDef.class)
14   private BindingController bc;
15
16   @Controller(name="super-controller",impl=SuperControllerDef.class)
17   private SuperControllerNotifier sc;
18
19   @Controller(name="lifecycle-controller",impl=LifeCycleControllerDef.class)
20   private LifeCycleCoordinator lc;
21
22   @Controller(name="name-controller",impl=NameControllerDef.class)
23   private NameController nc;
24 }
```

Figure 9: Definition of the Primitive membrane.

7.6 Mixin

The notion of a mixin, as proposed by [2], is an inheritance mechanism based on the composition of abstract classes. This mechanism enables the fields and methods of source classes to be composed and mixed into a target class. This mechanism is used in single inheritance languages such as Java to provide a substitute for the absence of multiple inheritance. This mechanism is implemented for example in the JAM [1] extension to the Java language. The Julia framework, that is the reference implementation of the Fractal component framework, is using mixins for reusing code in membrane and controller implementations.

Adlet provides an implementation of the mixin mechanism. Figure 10 illustrates its usage. The annotation `Mixin` (line 1) defines the list of sources classes (parameter `layers` from lines 3 to 14) to be composed into the target class whose name is defined by the parameter `impl` (line 2). Note that the class `ContainerBindingControllerDef` (line 16) is empty, and is just a placeholder for hosting the definition of the annotation `Mixin`.

7.7 Fractal Control Interfaces

The control interfaces defined in a membrane can be accessed with the `@Controller` annotation. Figure 11 illustrates the definition of a component that accesses two control interfaces. Line 3 declares the field named `self` that stores the reference to the control interface `component` of type `Component`. Line 4 declares the field name `sc` that stores the reference to the super controller that is the controller that enables to retrieve the parents of the current component. The parameter `name` of the annotation `Controller` specifies the control interface name that the developer wants to retrieve. If omitted, the value `"component"` is assumed. The type of the field must correspond to the type of the control interface defined in the membrane of the component (see Section 7.5).

```
1  @Mixin(
2    impl="juliac.generated.ContainerBindingControllerImpl",
3    layers={
4      BasicControllerMixin.class,
5      ContainerBindingControllerMixin.class,
6      FracletBindingControllerMixin.class,
7      UseComponentMixin.class,
8      InterceptorBindingMixin.class,
9      CheckBindingMixin.class,
10     TypeBindingMixin.class,
11     UseSuperControllerMixin.class,
12     ContentBindingMixin.class,
13     UseLifeCycleControllerMixin.class,
14     LifeCycleBindingMixin.class
15   })
16 public class ContainerBindingControllerDef {}
```

Figure 10: Definition of a mixin class.

```
1  @Component(...))
2  class HelloWorld {
3    @Controller org.objectweb.fractal.api.Component self;
4    @Controller(name="super-controller") SuperController sc;
5  }
```

Figure 11: Access to Fractal control interfaces.

8 Example: The Comanche Web Server

This section illustrates the usage of Adlet with the so-called Comanche web server. As any other web server, Comanche handles HTTP requests and serves the corresponding content. For the sake of simplicity, only GET request are supported. The program can of course be extended to support other types of requests and functionalities. The architecture of the Comanche web server is illustrated in Figure 12. This architecture defines 11 components, including 4 composite ones and 7 atomic ones, in a 4-level hierarchy. The Comanche composite component is the top-level component of this hierarchy.

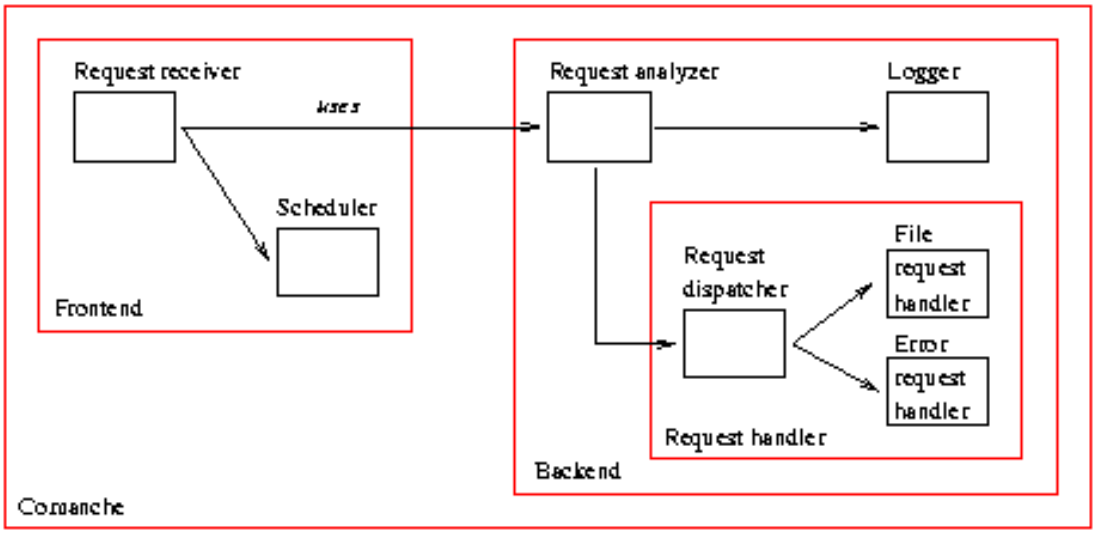


Figure 12: The architecture of the Comanche web server.

The code of the Comanche composite component is provided in Figure 13. The component provides the interface named `r` of type `Runnable` (line 1). The metamodel is generated in the class named `Comanche_` (line 2). Two subcomponents, `fe` and `be` respectively of types `FrontEnd` and

`BackEnd`, are defined (lines 5 and 6). Two bindings, one exporting the interface `r` of component `fe` to component `Comanche` (line 11), and one between the interface `rh` of component `fe` and the interface `rh` of component `be` (line 12), are defined.

```
1  @Component(provides=@Interface(name="r",signature=Runnable.class))
2  @StaticMetamodel("Comanche_")
3  public class Comanche {
4
5    @Component private FrontEnd fe;
6    @Component private BackEnd be;
7
8    @Binding
9    public static Binder binder() {
10     Binder binder = new Binder();
11     binder.export(Comanche_.r, Comanche_.fe.r);
12     binder.normal(Comanche_.fe.rh, Comanche_.be.rh);
13     return binder;
14 } }
```

Figure 13: Definition of the Comanche composite component.

The complete code of the Comanche web server can be found in Appendix A.

9 Conclusion

This document defines the Adlet Architecture Description Language for the Fractal component framework [3]. Adlet defines a programming style that enables to define component hierarchies and bindings between components directly in the Java language. Compared to the existing solutions where the architecture is defined in XML and component code is defined in Java, this is a clear advantage since the entire solution stays within the boundaries of only one language, the solution is better integrated with existing development environments and tools, and maintenance operations such as refactoring and bug fixing is eased. For that, Adlet provides a Java library that defines annotations, types, and methods. The rules associated with bindings originate from the specification of the Fractal component framework. They are enforced in Adlet thanks to two main elements: metamodels automatically generated from Adlet architectural descriptors, and generic typing rules defined in the API of the library.

An implementation and some examples for Adlet are available for download at https://gitlab.ow2.org/fractal/juliac/tree/master/extension/adlet.

References

[1] D. Ancona, G. Lagorio, and E. Zucca. A smooth extension of Java with mixins. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'00)*, pages 154–178, Sophia Antipolis and Cannes, France, June 2000.

[2] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (ECOOP/OOPSLA'90)*, volume 25 of *SIGPLAN Notices*, pages 303–311. ACM Press, October 1990.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Software Practice and Experience (SPE)*, 36(11-12):1257–1284, 2006.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[6] R. Rouvoy and P. Merle. Leveraging Component-Based Software Engineering with Fraclet. *Annals of Telecommunications - annales des télécommunications*, 64(1-2), January 2009.

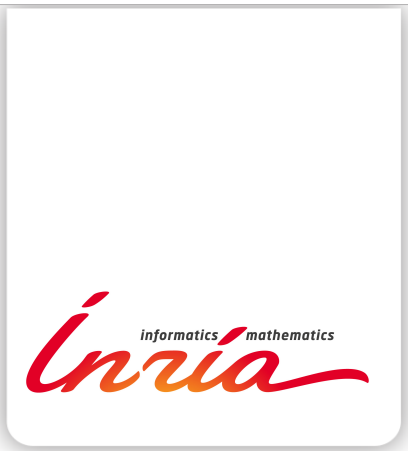
A Code of the Comanche web server

This appendix provides the code of the Comanche web server introduced in Section 8. The code can be downloaded from
https://gitlab.ow2.org/fractal/juliac/tree/master/extension/adlet/examples/comanche.

```
1  @Component(provides=@Interface(name="r",signature=Runnable.class))
2  @StaticMetamodel("Comanche_")
3  public class Comanche {
4
5    @Component private FrontEnd fe;
6    @Component private BackEnd be;
7
8    @Binding
9    public static Binder binder() {
10     Binder binder = new Binder();
11     binder.export(Comanche_.r, Comanche_.fe.r);
12     binder.normal(Comanche_.fe.rh, Comanche_.be.rh);
13     return binder;
14 } }
15
16
17 @Component(provides=@Interface(name="r",signature=Runnable.class))
18 @StaticMetamodel("FrontEnd_")
19 public class FrontEnd extends CFrontEndType {
20
21   @Component private RequestReceiver rr;
22   @Component private MultiThreadScheduler s;
23
24   @Binding
25   public static Binder binder() {
26     Binder binder = new Binder();
27     binder.export(FrontEnd_.r, FrontEnd_.rr.r);
28     binder.impor(FrontEnd_.rr.rh, FrontEnd_.rh);
29     binder.normal(FrontEnd_.rr.s, FrontEnd_.s.s);
30     return binder;
31 } }
32
33
34 @Component(provides=@Interface(name="r",signature=Runnable.class))
35 public class CFrontEndType {
36   @Requires protected RequestHandler rh;
37 }
38
39
40 @Component(provides=@Interface(name="r",signature=Runnable.class))
41 public class RequestReceiver implements Runnable {
42
43   @Requires private Scheduler s;
44   @Requires private RequestHandler rh;
45
46   public void run() {
47     ServerSocket ss = null;
48     try {
49       ss = new ServerSocket(8042);
50       System.out.println("Comanche␣HTTP␣Server␣ready␣on␣port␣8042.");
51       System.out.println("Load␣http://localhost:8042/.../gnu.jpg");
52       while (true) {
53         final Socket socket = ss.accept();
54         s.schedule(new Runnable() {
55           public void run() {
56             try {
57               rh.handleRequest(new Request(socket));
58             } catch (IOException e) {}
59           } });
60       }
61     } catch (IOException e) { e.printStackTrace(); }
62       finally {
63         if( ss != null ) {
64           try { ss.close(); }
65           catch( IOException ioe ) {}
66 } } } }
67
```

```
68
69  @Component(provides=@Interface(name="s",signature=Scheduler.class))
70  public class MultiThreadScheduler implements Scheduler {
71    public void schedule(Runnable task) {
72      new Thread(task).start();
73  } }
74
75
76  @Component(provides=@Interface(name="rh",signature=RequestHandler.class))
77  @StaticMetamodel("BackEnd_")
78  public class BackEnd {
79
80    @Component private RequestAnalyzer ra;
81    @Component private BasicLogger l;
82
83    @Component(provides=@Interface(name="rh",signature=RequestHandler.class))
84    static class Handler {
85      @Component private RequestDispatcher rd;
86      @Component private FileRequestHandler frh;
87      @Component private ErrorRequestHandler erh;
88    }
89
90    @Binding
91    public static Binder binder() {
92      Binder binder = new Binder();
93      binder.export(BackEnd_.rh, BackEnd_.ra.rh);
94      binder.normal(BackEnd_.ra.l, BackEnd_.l.l);
95      binder.normal(BackEnd_.ra.a, BackEnd_.Handler.rh);
96      binder.export(BackEnd_.Handler.rh, BackEnd_.Handler.rd.rh);
97      binder.normal(BackEnd_.Handler.rd.h.setSuffix("0"), BackEnd_.Handler.frh.rh);
98      binder.normal(BackEnd_.Handler.rd.h.setSuffix("1"), BackEnd_.Handler.erh.rh);
99      return binder;
100 } }
101
102
103 @Component(provides=@Interface(name="rh",signature=RequestHandler.class))
104 public class RequestAnalyzer implements RequestHandler {
105
106   @Requires(name="a") private RequestHandler rh;
107   @Requires private Logger l;
108
109   public void handleRequest(Request r) throws IOException {
110     r.in = new InputStreamReader(r.s.getInputStream());
111     r.out = new PrintStream(r.s.getOutputStream());
112     String rq = new LineNumberReader(r.in).readLine();
113     l.log(rq);
114     if (rq.startsWith("GET␣")) {
115       r.url = rq.substring(5 ,rq.indexOf('␣' ,4));
116       rh.handleRequest(r);
117     }
118     r.out.close();
119     r.s.close();
120 } }
121
122
123 @Component(provides=@Interface(name="l",signature=Logger.class))
124 public class BasicLogger implements Logger {
125    public void log(String msg) {
126      System.out.println(msg);
127 } }
128
129
130 @Component(provides=@Interface(name="rh",signature=RequestHandler.class))
131 public class RequestDispatcher implements RequestHandler {
132
133   @Requires(name="h",cardinality=Cardinality.COLLECTION)
134   private Map<String,RequestHandler> handlers = new TreeMap<>();
135
136   public void handleRequest(Request r) {
137     for (RequestHandler h : handlers.values()) {
138        try {
139          h.handleRequest(r);
140          return ;
```

```
141         } catch (IOException e) {}
142 } } }
143
144
145 @Component(provides=@Interface(name="rh",signature=RequestHandler.class))
146 public class FileRequestHandler implements RequestHandler {
147   public void handleRequest(Request r) throws IOException {
148     File f = new File(r.url);
149     if ((f.exists()) && (!(f.isDirectory()))) {
150       InputStream is = new FileInputStream(f);
151       byte[] data = new byte[is.available()];
152       is.read(data);
153       is.close();
154       r.out.print("HTTP/1.0 200 OK\n\n");
155       r.out.write(data);
156     } else {
157       throw new IOException("File not found");
158 } } }
159
160
161 @Component(provides=@Interface(name="rh",signature=RequestHandler.class))
162 public class ErrorRequestHandler implements RequestHandler {
163   public void handleRequest(Request r) {
164     r.out.print("HTTP/1.0 404 Not Found\n\n");
165     r.out.print("<html>Document not found.</html>");
166 } }
167
168
169 public interface Scheduler {
170   void schedule(Runnable task);
171 }
172
173
174 public interface Logger {
175   void log(String msg);
176 }
177
178
179 public interface RequestHandler {
180   void handleRequest(Request r) throws IOException;
181 }
182
183
184 public class Request {
185   public Socket s;
186   public Reader in;
187   public PrintStream out;
188   public String url;
189   public Request(Socket s) { this.s = s; }
190 }
```



RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399