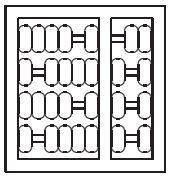

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

COSMOS*: a COmponent System MOdel for Software Architectures

Leonel Aguilar Gayard
Cecília Mary Fischer Rubira
Paulo Asterio de Castro Guerra

Technical Report - IC-08-004 - Relatório Técnico

February - 2008 - Fevereiro

COSMOS*: a COmponent System MOdel for Software Architectures

Leonel Aguilar Gayard (leonel.gayard@students.ic.unicamp.br) *
Cecília Mary Fischer Rubira (cmrubira@ic.unicamp.br)
Paulo Asterio de Castro Guerra (asterio@acm.org)

Abstract

Software architecture and component-based development are complementary approaches to the development of intensive software systems. Software architecture describes a system in terms of its logical architectural components, while component-based development focuses on the reuse of existing software components for building new software systems. However, software components are usually developed in programming languages that do not include component and software architecture abstractions, such as interfaces, architectural components and architectural connectors.

So there is a gap between the conception of an abstract software architecture and its concrete implementation in a programming language. The COSMOS* model uses programming language features and well-known design patterns to represent software architectures explicitly in the program code. The COSMOS* model defines the specification and implementation of COSMOS* components, COSMOS* connectors, COSMOS* architectural configurations, and COSMOS* composite components. Case studies applying the COSMOS* model are discussed using the Java programming language.

1 Introduction

Software architecture and component-based development (CBD) are complementary approaches to the construction of software systems. The architecture of a software system is the structure or structures of a software system, which consists of its software elements, the externally visible properties of these elements and the relations between them [3]. The architecture of a software system describes the system in terms of its **architectural components**, which are elements of its architecture responsible for the system's requirements, and its **architectural connectors**, which are simpler components, responsible for the communication between components. Architectural components define **provided and required interfaces**, which are specifications of the services they provide to other components and those they require from other components in order to function [5], respectively. An arrangement of architectural components connected by architectural connectors is called an **architectural configuration** [3].

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP. Supported by CAPES, Brazil under grant 01P-05603/2006

On the other hand, CBD is an approach to the construction of software systems which aims to reduce the time and cost of the construction of new systems by reusing existing software components. Szyperski [24] presents the following definition for a software component:

> "A software component is a unit of composition with specified contractual interfaces and explicit context dependencies. A software component can be deployed and is subject to composition by third parties."

While an architectural component is an abstract concept, a software component is a concrete implementation of one or more architectural components and can be executed in a physical or logical device. The software component integrates data and function with high encapsulation, high cohesion and low coupling. Also, whereas the interfaces of an architectural component are specifications of services provided by the component, the interfaces of a software component are implementations of the interfaces defined by the architectural component.

Reusable software components are usually developed without knowledge of the various different contexts in which they will be employed. The software architecture is therefore responsible for integrating these components in order to obtain the desired quality attributes for the developing system. Thus, a **concrete architectural configuration** is an organization of physical components in which each physical component implements an architectural component. The correspondence between software components and software architecture is evidenced in the main CBD processes, such as UML Components [6] and Catalysis [11], which are also software architecture centric.

In the design of a software system, a semantic gap can exist between the architecture of a system and the programming language used to implement it. On one hand, architectures are usually described in terms of formal or semi-formal artifacts, such as textual descriptions and diagrams. On the other hand, many modern programming languages do not include key abstractions of software architecture, such as architectural components and connectors. Therefore, the structure of a program may not reflect explicitly its architecture.

The COSMOS* model (COmponent System MOdel for Software architectures) reduces this gap by explicitly representing abstractions from software architecture, such as architectural components and connectors, into the program code. In the COSMOS* model, each architectural element is organized as a set of packages structured according to rules proposed by the model. Each architectural element is internally implemented by of classes implementing well-known design patterns such as *Façade*, *Adapter*, *Factory Method* [16] and *Dependency Injection* [15]. The COSMOS* model is divided in a specification model, by which a component exposes its specification; an implementation model, which guides the internal implementation of a component; a connector model, which specifies the connection between components using connectors; and a composition model, by which new components or entire systems are built using existing components.

The COSMOS* model extends the COSMOS model [7, 8] and extends it with the concepts of composite components and component ports. Throughout this work, the Java programming language [20] and UML 2.0 [17] will be used to represent COSMOS* components.

Section 2 presents an overview the COSMOS* model. Section 3 shows the COSMOS* model. Section 3.7 presents the package `br.unicamp.ic.sed.cosmos`, which contains classes and interfaces used by COSMOS* components. Section 4 and 5 present other aspects of the development of COSMOS* components, such as external dependencies and web development with COSMOS* components. Section 6 presents a case study of a system built using COSMOS* components. Section 7 describes techniques for exception handling in COSMOS* components. Section 8 compares the COSMOS* and EJB component models and describes how to use the EJB technology to build COSMOS* components.

2 Overview of the COSMOS* Model

The COSMOS* model presents definitions for architectural components, connectors and configurations. These elements are represented as abstractions implemented in modern object-oriented programming languages, such as packages, modules and *namespaces*. As stated in Section 1, in this work, we adopt the Java programming language to illustrate the use of the COSMOS* model. However, other object-oriented programming languages could be adopted, such as C++, C# and Python [13].

Figure 1 shows a COSMOS* component `A` with two provided interfaces `IA` and `IManager` and a required interface `IB`.

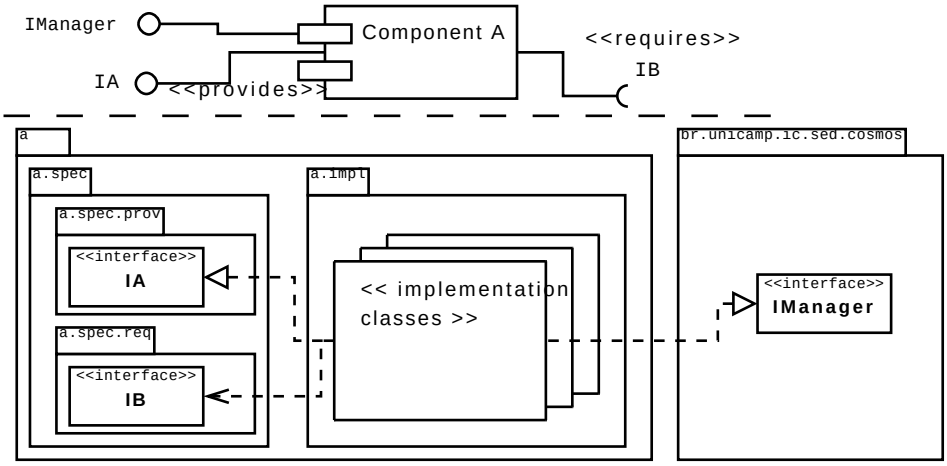

Figure 1: The package structure of COSMOS* components

- In the COSMOS* model, components and connectors are represented by packages with their name in lower case characters. As an example, in Figure 1, component `A` is represented by package `a`.

- Sub-package `a.spec` is the specification package of component A. It is divided in tow other packages `a.spec.prov` and `a.spec.req`, which contain the component's provided and required interfaces, respectively. The provided interface `IA` is defined in package `a.spec.prov`, and the required interface `IB` is defined in package `a.spec.req`. The specification of a COSMOS* component is detailed in Section 3.1.

- COSMOS* components and connectors present a management interface called `IManager`, which is a special interface allowing meta-information recovery. It allows users of a COSMOS* component to query about its interfaces and connect them to interfaces of other components. This interface is not defined in the package spec of a component, but in the special package `br.unicamp.ic.sed.cosmos`, so that all COSMOS* components share this interface definition. Interface `IManager` is detailed in Section 3.1.2.

- The implementation classes are in package `a.impl`. These classes realize the provided interfaces `IManager` and `IA`, and use the required interface `IB`. The implementation of a COSMOS*component is detailed in Section 3.2.

The connection between two or more components is made by a COSMOS* connector, which is a simple component responsible for the communication between required interfaces of a component and provided interfaces of other components. It is implemented following the *Adapter* design pattern [16]. A COSMOS* connector does not present a specification package; instead, it implements the connection between the required and provided interfaces of two or more components. Figure 2 depicts a composite COSMOS* component `X` composed by two components `A` and `B` and a COSMOS* connector `AB`. In Figure 2, the type of the provided interface `IB` of component B is `b.spec.prov.IB`, which is different from the type of the required interface `IB` of component A, which is `a.spec.req.IB`. So, connector `AB` adapts the provided interface `IB` of component B to the required interface `IB` of component A. Connector `AB` is implemented by the package `ab` and implements the interface `IManager` from package `br.unicamp.ic.sed.cosmos` . COSMOS* connectors are detailed in Section 3.3.

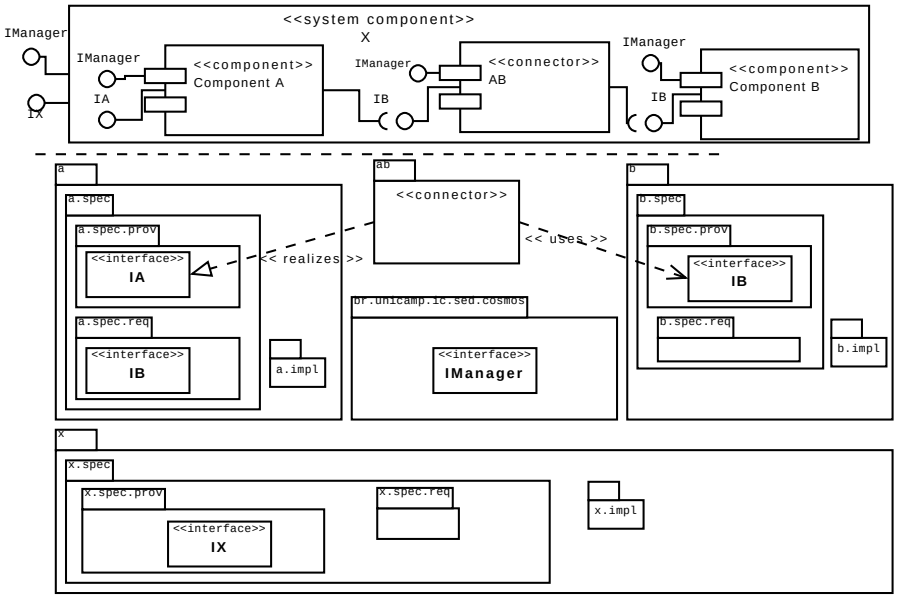

Figure 2: The package structure of COSMOS* connectors

In Figure 2, components `A` and `B`, connector `AB` and the connections between the provided and required interfaces constitute the **architectural configuration** of the **composite component** `X`. The COSMOS* model also defines a composition model which allows to create new components from existing components. The composition model is detailed in Section 3.4.

In Figure 2, component `X` is a **system component**. A system component is a COSMOS* component which is a stand-alone system that can be deployed and executed. The

COSMOS* model also defines a model for system components and component execution, detailed in Section 3.5.

3 The COSMOS* model

The COSMOS* model defines the specification (Section 3.1) and implementation (Section 3.2) of COSMOS* components, COSMOS* connectors (Section 3.3), COSMOS* composite components (Section 3.4) and systems based on the COSMOS* model (Section 3.5).

3.1 Specification of COSMOS* components

3.1.1 Specification package of a COSMOS* component

The specification of a COSMOS* component aims to emphasize the separation between the specification of a component, which should be externally visible, from its implementation. The implementation of a component should not be accessible, so that all communication with the component is comprised through its interfaces.

As defined in Section 2, the specification of a COSMOS* component is described by a package called `spec`, located in a larger package which represents the software component itself. Each provided interface of the architectural component is mapped to a UML interface in package `spec.prov` and each required interface, to a UML interface in package `spec.req`. In Figure 1, component `A` offers two provided interfaces `IA` and `IManager` and has a required interface `IB`. Figure 3 depicts the definition of these interfaces.

```
package a.spec.prov;
public interface IA { /* methods in provided interface IA */ }

package a.spec.req;
public interface IB { /* methods in required interface IB */ }
```

Figure 3: Packages definition for provided and required interfaces of component A

3.1.2 The interface `IManager`

A COSMOS* component is required to offer an interface called `IManager`, which is implemented by class `Manager` in the component's implementation package. The interface `IManager` is defined in package `br.unicamp.ic.sed.cosmos` and has three purposes:

1. An instance of this interface represents an instance of the component at runtime. Thus, different instances of a component can exist at runtime, each with a different state, and represented by different instances of the `IManager` interface.

2. An instance of the `IManager` interface allows one to discover at runtime what are the provided and required interfaces of a component.

3. Through interface `IManager`, it is possible to connect the required interfaces of a component to the provided interfaces of another component.

Figure 4 depicts the methods of interface `IManager`:

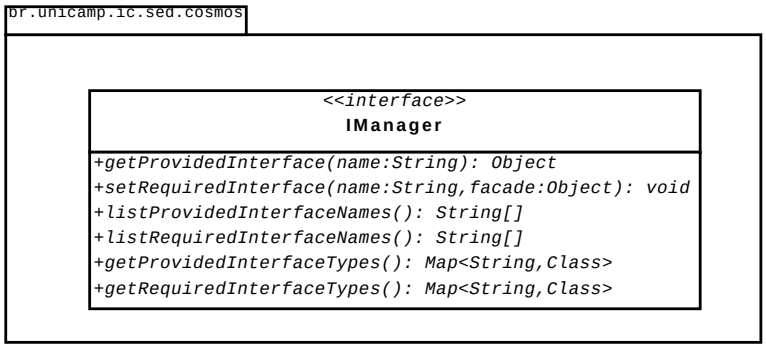

Figure 4: The `IManager` interface in UML.

The signatures of the methods in interface `IManager` are:

1. **getProvidedInterface(name : String) : Object**. Returns an implementation of a provided interface of this component; the argument is the name of the provided interface.

2. **setRequiredInterface(name : String, facade : Object) : void**. Connects a required interface of this component. The first argument is the name of the required interface to be connected; the second argument is the provided interface of another component, which implements the methods expected by the required interface.

3. **listProvidedInterfaceNames() : String**[]. Returns an array with the names of the provided interfaces of this component.

4. **listRequiredInterfaceNames() : String**[]. Returns an array with the names of the required interfaces of this component.

5. **getProvidedInterfaceTypes() : Map<String,Class>**. Returns a Map (an object of type `java.util.Map`) where each key is the name of a provided interface and its value is an object Class for the type of the interface.

6. **getRequiredInterfaceTypes() : Map<String,Class>**. This method is similar to method `getProvidedInterfaceTypes`, but returns the names and types of the required interfaces of the component.

Figure 5 depicts the Java definition of interface `IManager`, using the Java 1.5 syntax for Generics [20].

```
1  package br.unicamp.ic.sed.cosmos;
2  import java.util.Map;
3  public interface IManager {
4    public Object getProvidedInterface(String name);
5    public void setRequiredInterface(String name, Object facade);
6    public String[] listProvidedInterfaceNames();
7    public String[] listRequiredInterfaceNames();
8    public Map<String,Class<?>> getProvidedInterfaceTypes();
9    public Map<String,Class<?>> getRequiredInterfaceTypes();
10 }
```

Figure 5: The definition of the IManager interface in Java

3.2 Implementation of COSMOS* components

The implementation classes of a component are located in its `impl` package. Figure 6 depicts the classes located in the package `a.impl`, which represent the implementation of component `A`. This package should contain a class `ComponentFactory`, a class `Manager`, which implements the `IManager` interface (Section 3.1.2) and a set of `Façade` classes, one for each provided interface different from `IManager`.

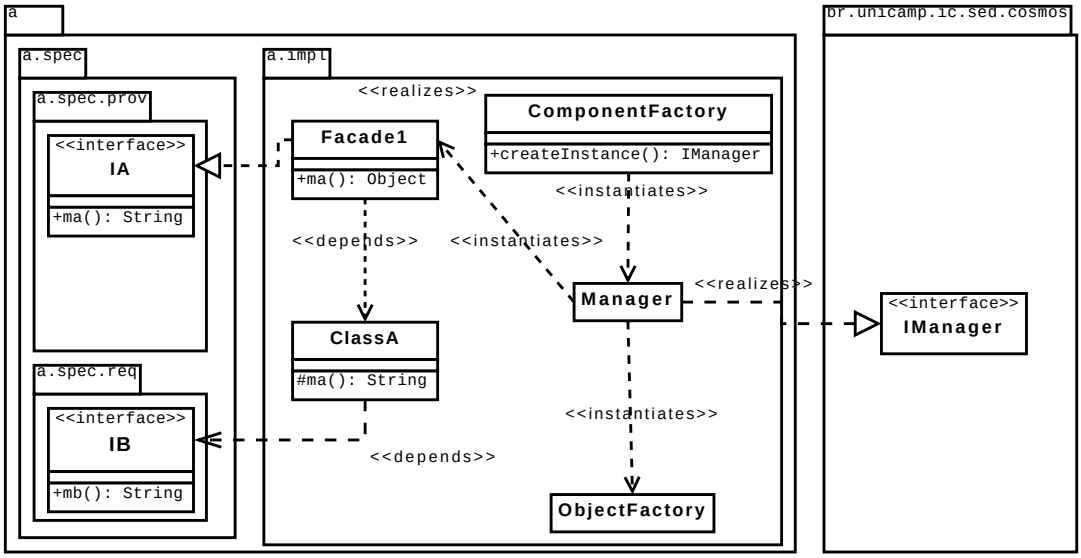

Figure 6: Class diagram of the implementation package of component `A`

3.2.1 The `ComponentFactory` class

The `ComponentFactory` class is implemented following the *Factory Method* design pattern [16]. It is responsible for instantiating the component. This class contains only one method `createInstance()`, static, and responsible for creating a new instance of the component. It is the only class in package `impl` with public visibility.

The implementation of class `ComponentFactory` of component `A` is depicted in Figure 7.

```
1 package a.impl;
2 import br.unicamp.ic.sed.cosmos.IManager;
3 public class ComponentFactory {
4   public static IManager createInstance() {
5     return new Manager();
6   }
7 }
```

Figure 7: Implementation of class ComponentFactory from component `A`

3.2.2 The `Manager` class

Each COSMOS* component provides a class `Manager` in its implementation package, which realizes the interface `IManager`. The implementation of class `Manager` is not part of the COSMOS* model and its implementation should be provided by the component's developer. Thus, each component can have its own implementation of class `Manager`. Two different implementations of class `Manager` can vary in how they manage internally the objects implementing the interfaces of the component.

```
1 package a.impl;
2 class Manager implements br.unicamp.ic.sed.cosmos.IManager {
3   public Object getProvidedInterface(String name) { ... }
4   public void setRequiredInterface(String name, Object facade) { ... }
5   public String[] listProvidedInterfaceNames() { ... }
6   public String[] listRequiredInterfaceNames() { ... }
7   public Map<String,Class> getProvidedInterfaceTypes() { ... }
8   public Map<String,Class> getRequiredInterfaceTypes() { ... }
9 }
```

Figure 8: Java implementation of the `Manager` class of component `A`

Figure 9 suggests one possible implementation, which consists in creating a method `setProvidedInterface` in class `Manager`. Such a method stores an instance of each `Façade` class in a Java map (the interface `java.util.Map`) indexed by the name of the interface (line 13). Then, method `getProvidedInterface` only needs to return the `Façade` object stored in the map under the name of the requested interface (line 21).

In a similar way, the methods `setProvidedInterfaceType` and `setRequiredInterfaceType` store the types of the provided and required interfaces (lines 29 and 32).

In the instantiation process of the component, the constructor of class `Manager` (lines 39 to 45) invokes the methods `setProvidedInterface`, `setProvidedInterfaceType` and `setRequiredInterfaceType` in order to store the objects which implement the provided interfaces of component `A` (Figure 6).

```
1  package a.impl;
2  import java.util.Map;
3  import a.spec.prov.IManager;
4  class Manager implements IManager {
5    private Map providedInterfaces;
6    private Map providedInterfaceTypes, requiredInterfaceTypes;
7
8    /*
9     * Method setProvidedInterface stores the object which
10    * implements a provided interface.
11    */
12   void setProvidedInterface(String name, Object facade) {
13     this.providedInterfaces.put(name, facade);
14   }
15
16   /*
17    * Method getProvidedInterface returns the object which
18    * implements a requested interface.
19    */
20   public Object getProvidedInterface(String name) {
21     return this.providedInterfaces.get(name);
22   }
23
24   /*
25    * Method setProvidedInterfaceType and setRequiredInterfaceType store the
26    * types of the provided and required interfaces.
27    */
28   void setProvidedInterfaceType(String name, Class type) {
29     this.providedInterfaceTypes.put(name, type);
30   }
31   void setRequiredInterfaceType(String name, Class type) {
32     this.requiredInterfaceTypes.put(name, type);
33   }
34
35   /*
36    * The constructor invokes methods setProvidedInterface, setProvidedInterfaceType
37    * and setRequiredInterfaceType to store the component's interfaces.
38    */
39    Manager() {
40     this.setProvidedInterface("IA", new FacadeA(this));
41     this.setProvidedInterface("IManager", this);
42     this.setProvidedInterfaceType("IA", a.spec.prov.IA.class);
43     this.setProvidedInterfaceType("IManager", br.unicamp.ic.sed.cosmos.IManager.class);
44     this.setRequiredInterfaceType("IB", a.spec.req.IV.class);
45    }
46 }
```

Figure 9: Class `Manager` with method `setProvidedInterface` of Component `A` in Java.

Another possible implementation of class `Manager` is to extend an existing class which implements the methods in interface `IManager`; as an example, the package `br.unicamp.ic.sed.cosmos`

contains an abstract class `AManager`, which implements the interface `IManager`. Thus a possible implementation of the class `Manager` in a component is to extend class `AManager`. However, this is only a possible implementation and is not mandatory in the COSMOS* model. Section 3.7 explains the implementation classes in package `br.unicamp.ic.sed.cosmos`.

3.2.3 The `Façade` classes

Façade classes in a COSMOS* component are the entry point for implementations of the provided interfaces of the component. Each provided interface is realized by a `Façade` class (except for interface `IManager`, which is realized by class `Manager`). A `Façade` class delegates the interface implementation to another class within the component, accordingly to the *Façade* design pattern [16].

In Figure 6, component `A` has a provided interface `IA`, realized by a class `FacadeA`. This class implements interface `IA` and delegates the calls to its methods to class `ClassA`. The implementation of class `FacadeA` is depicted in Figure 10:

```
1  package a.impl;
2  class FacadeA implements a.spec.prov.IA {
3    private ClassA a;
4    private Manager manager;
5    FacadeA(Manager mgr) {
6      this.manager = mgr;
7      this.a = new ClassA(this.manager);
8    }
9    public Object ma() { return this.a.ma(); }
10 }
```

Figure 10: Implementation of class `FacadeA` in Java

The constructor of class `FacadeA` has a parameter that allows the class to receive an instance of the component's `Manager` object (line 5). This reference to the manager is stored in the `FacadeA` object (line 6) and should be passed to the implementation classes to which the Façade object delegates (line 7). Thus, the implementation classes can access the component's `Manager` object and use the component's required interfaces.

3.2.4 Required Interfaces

Required interfaces in a component represent the dependencies of this component for services provided by other components. In Figure 6, component `A` has a required interface `IB`, with a method `mb()`.

Figure 11 depicts the implementation of class `ClassA`. When a component's class needs a service offered through interface `IB`, it invokes the method `getRequiredInterface` in the class `Manager` and acquires an instance of `IB`. In Figure 11, class `ClassA` gets a reference to the component's required interface `IB` and invokes the method `mb()` on the object `ib` (line 8).

```
1  package a.impl;
2  class ClassA {
3    private Manager manager;
4    ClassA(Manager mgr) {
5      this.manager = mgr;
6    }
7    String ma() {
8      a.spec.req.IB ib = (a.spec.req.IB) this.manager.getRequiredInterface("IB");
9      return "->" + ib.mb();
10   }
11 }
```

Figure 11: Access to a required interface.

3.2.5 Optional: class ObjectFactory

It is possible to include a class `ObjectFactory` in the implementation package. The class `ObjectFactory` is not mandatory in the COSMOS* model and its use is optional. This class acts as an object factory, as described in the *Abstract Factory* design pattern [16], and aims to reduce coupling between implementation classes within the component. With the class `ObjectFactory`, implementation classes in the component do not reference each other directly; instead, there exists in the implementation package an interface for each functionality implemented by the component; implementation classes reference only those interfaces. For each such interface, there is a method in class `ObjectFactory` which returns the interface.

Unlike the interfaces in the specification package, the interfaces in the implementation package have package-level visibility (which means they are not visible out of the component).

Figure 12 depicts the implementation package of component `A` modified to use the class `ObjectFactory`:

- Class `Manager` instantiates class `ObjectFactory`.
- The implementation classes `FacadeA`, `Class1` and `Class2` do not reference each other directly; instead, they reference interfaces `Interface1` and `Interface2`. Class `ObjectFactory` has methods `getInterface1()` and `getInterface2`, which return objects implementing these interfaces.
- Class `Manager` contains a method `getObjectFactory()`, which returns an instance of class `ObjectFactory`. The other implementation classes invoke this method to reach the component's object factory.

The use of an object factory reduces the effort spent on maintenance and evolution of the component when its implementation is complex: because every communication happens through interfaces, some implementation classes can be modified without impacting on other classes.

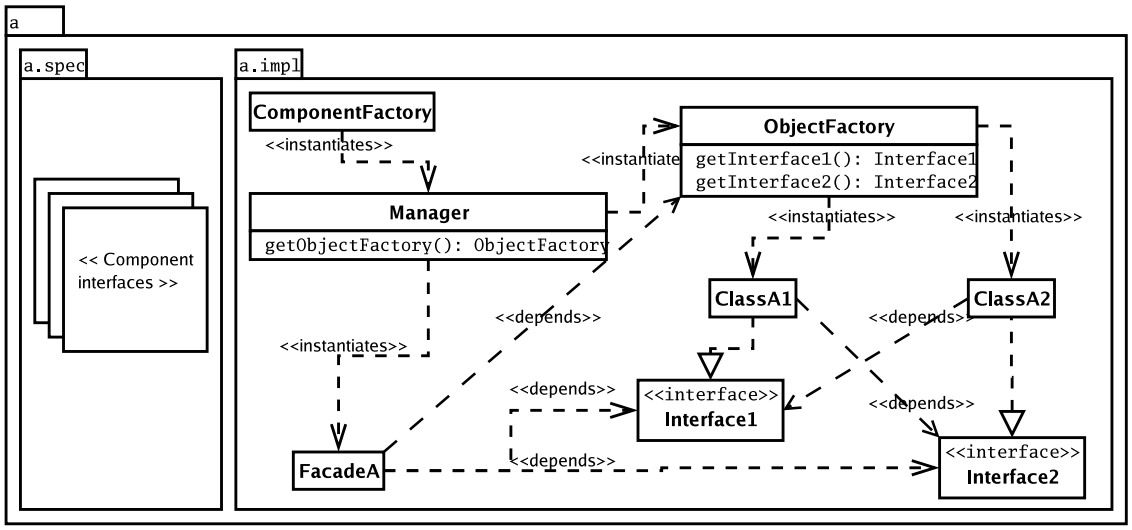

Figure 12: The implementation package of component A with class `ObjectFactory`

Moreover, the use of an object factory allows the versioning of the various implementations of the component, by creating different object factories.

3.3 COSMOS* Connectors

A COSMOS* connector implements the connection between required and provided interfaces of two or more components, as stated in Section 2. Connectors are simple COSMOS* components: they are implemented as packages in a similar way to COSMOS* components, but are not allowed to define new interfaces. Figure 13 depicts an architectural configuration with a COSMOS* connector `AB`, in package `ab` and its classes.

Because a COSMOS* connector does not define new interfaces, it does not present specification and implementation packages; instead, the connector is represented as a single package, which contains all the classes of the connector.

The *fully qualified names* [20] of the required interface of `A` and the provided interface of `B` are `a.spec.req.IB` and `b.spec.prov.IB`. Because these fully qualified names are different, these interfaces cannot be connected directly. Connector `AB` in Figure 13 implements the required interface `IB` of Component `A` by using the provided interface `IB` of Component `B`. The connector is an application of the *Adapter* design pattern [16], for it allows an indirect connection between the incompatible interfaces of components `A` and `B`.

Also, connector `AB` allows the communication between the components without component `A` referencing interfaces defined in component `B`, and, inversely, without `B` referencing interfaces defined in `A`. Thus, a component has no dependencies on another. This mechanism is core to the COSMOS* model for allowing a low coupling between components.

On the other hand, a dependency arises between the connector and the components it connects. In Figure 13, connector `AB` references interfaces defined in components `A` and `B`, which qualifies a high coupling between the connector and the components. The COS-

MOS* model establishes that components should be developed focusing on their reuse, and connectors should be developed with focus on the communication between the components and on the adaptation of their interfaces, and with no concern for their reuse. A connector is reused only if two or more components are reused in a new architecture and a connector exists that is suitable for the connection between the reused components. Otherwise, a new connector must be developed.

In Figure 13, connector `AB` has the following classes:

- `Adapter` classes adapt the provided and required interfaces of components. This sort of adaptation can be seen as an application of the *Adapter* design pattern [16].
- The interface `IManager`, identical to the interface IManager of a COSMOS* component (Section 3.1.2).
- Class `Manager`, which implements interface `IManager`.
- Class `ComponentFactory`, responsible for instantiating the connectors.

3.3.1 Classes `Adapter` in COSMOS* connectors

The implementation of class `AdapterB` is depicted in Figure 14. In the constructor of class `AdapterB`, the parameter allows the object to receive a reference to the Manager object of the connector and store it in an attribute (line 5). In order to implement method `mb` of required interface `IB` of component `A`, the connector uses the object `Manager` to retrieve its required interface `IB` (line 8). The required interface `IB` of the connector is of type `b.spec.prov.IB`, which is the same type of the provided interface of component `B`.

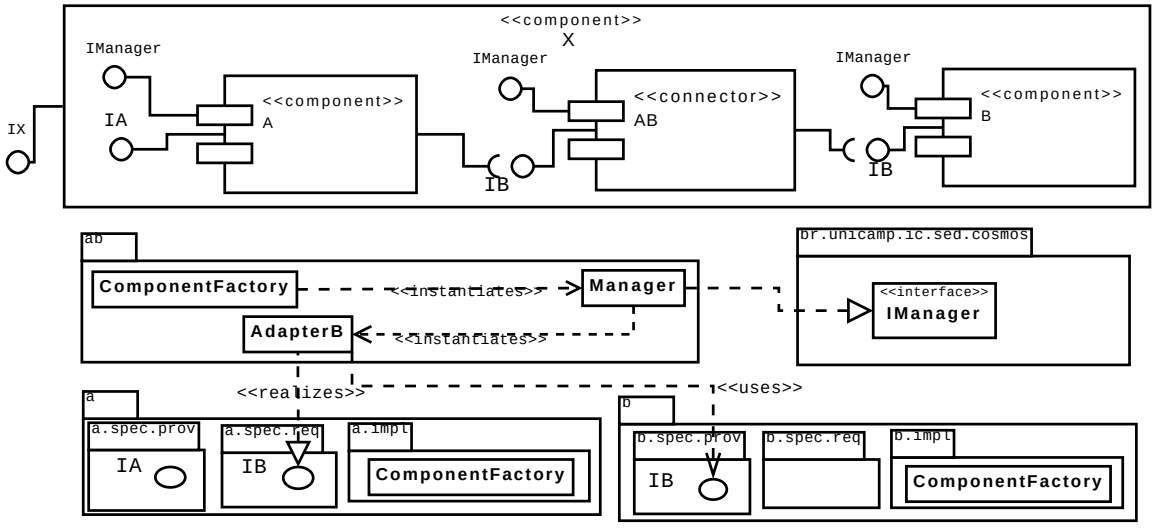

Figure 13: A COSMOS* connector and its classes

```
1  package ab;
2  class AdapterB implements a.spec.req.IB {
3    private Manager manager;
4    AdapterB(Manager mgr) {
5      this.manager = mgr;
6    }
7    public String mb() {
8      b.spec.prov.IB ib = (b.spec.prov.IB) this.manager.getRequiredInterface("IB");
9      return ib.mb();
10   }
11 }
```

Figure 14: Class `AdapterB` in connector AB.

3.3.2 Class `Manager` in a connector.

The implementation of class `Manager` in a connector is depicted in Figure 15. The class `Manager` of the connector is responsible for (1) storing the names and classes of the provided and required interfaces of the connector (lines 4 to 7); (2) instantiating class `Adapter`, which serves as an adapter between the interfaces of the connected components (line 8); and (3) storing and instance of class `Adapter` as a provided interface of the connector (line 9).

```
1  package ab;
2  class Manager implements br.unicamp.ic.sed.cosmos.IManager {
3    Manager() {
4      this.setProvidedInterfaceTypes(new Class[] { a.spec.req.IB.class });
5      this.setProvidedInterfaceNames(new String[] { "IB" });
6      this.setRequiredInterfaceTypes(new Class[] { b.spec.prov.IB.class } );
7      this.setRequiredInterfaceNames(new String[] { "IB" });
8      AdapterI3 adapter = new AdapterB(mgr);
9      this.setProvidedInterface("IB", adapter);
10   }
11 }
```

Figure 15: Implementation of class `Manager` of connector AB.

3.3.3 Class `ComponentFactory` in a connector

A connector's class `ComponentFactory` has the same funcionalities as a component's. It presents the method `createInstance()` , responsible for creating an instance of the connector (Figure 16).

3.4 Composite COSMOS* component

The COSMOS* model extends the COSMOS model [7] with a model for composition.

A composite COSMOS* component instantiates other COSMOS* components as its internal parts and propagates to them the implementation of its functionalities. Thus,

```
1 package ab;
2 import br.unicamp.ic.sed.cosmos.IManager;
3 public class ComponentFactory {
4   public IManager createInstance() {
5     return new Manager();
6   }
7 }
```

Figure 16: Implementation of class `ComponentFactory` in connector AB.

sub-components of a composite component are not visible to other components. This recursive definition is essential to a component model [24], for it allows the creation of a new component reusing features from other components.

The set of internal components and connectors of a composite and the connections between their interfaces is called the **architectural configuration** of the composite component, according to the terminology presented in [10].

Figure 17 depicts component `X` from Figure 2 revealing its implementation classes `FacadeX` and `AdapterXReq`, which allow component `X` to use its internal components `A` and `B` and its internal connector `AB`. Also in Figure 17, the class `Manager` of composite component `X` extends class `AManagerComposite` in package `br.unicamp.ic.sed.cosmos`; the class `Manager` in a composite component is responsible for handling the internal components of the composite and is explained in Section 3.4.1. The class `AManagerComposite` is explained in Section 3.7.

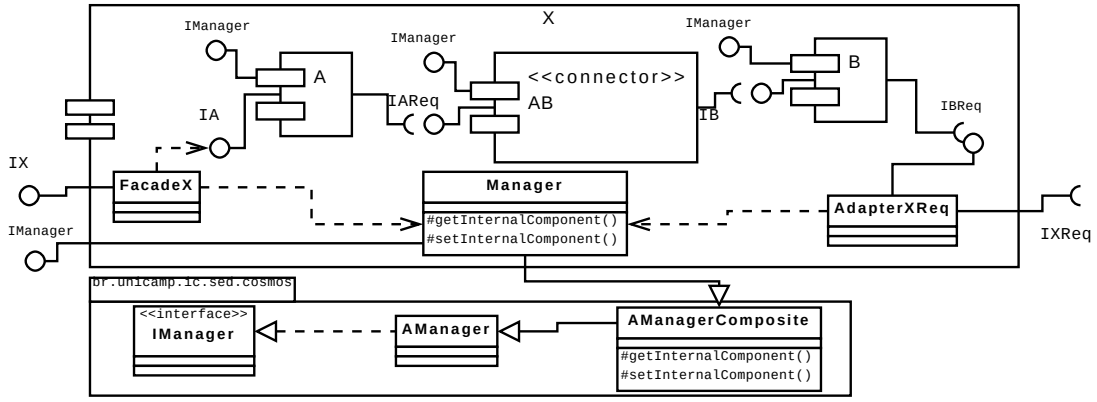

Figure 17: An example of a composite component.

Implementation classes in a component component undergo some changes in regard to the classes in an elementary component. In Figure 17, class `FacadeX` in the composite uses provided interface `IA` of its internal component `A`. Composite component `X` has a class `AdapterXReq`, similar the *Adapter* classes of the connectors (Section 3.3.1), which adapts required interface `IXReq` of the composite to required interface `IBReq` of the internal component `B`.

3.4.1 Class `Manager` in a composite COSMOS* component

Class `Manager` in a composite component implements interface `IManager`, as described in Section 3.1.2. Class `Manager` in a composite component is augmented with methods `setInternalComponent` and `getInternalComponent`, both with package visibility. The signatures of these methods are:

- **setInternalComponent(String name, IManager component) : void**. Stores a component as a sub-component of the composite. The first parameter of this method is the name under which the component will be stored; the second parameter, of type `IManager`, is an instance of the component to be stored.
- **getInternalComponent(String name): IManager**. Retrieves a component stored as internal to the composite. The argument is the name under which the component was stored.

The constructor of class `Manager` in a composite component instantiates class `FacadeX` of the composite and exposes it as a provided interface (Figure 18, line 6).

```
1  package x.impl;
2  import br.unicamp.ic.sed.cosmos.AManagerComposite;
3  import br.unicamp.ic.sed.cosmos.IManager;
4  class Manager extends AManagerComposite implements IManager {
5    Manager() {
6      this.setProvidedInterface("IX", new FacadeX(this));
7    }
8
9    void setInternalComponent(String name, IManager component) { ... }
10   IManager getInternalComponent(String name) { ... }
11 }
```

Figure 18: The class `Manager` of the composite component X in Figure 17

3.4.2 Classes `Façade` in a composite COSMOS* component

Each provided interface in a composite component is implemented as a class Façade, in a similar manner to the Façade classes in an elementary component (Section 3.2.3).

Figure 19 depicts class `FacadeX` of the composite component X in Figure 17:

- Class `FacadeX` implements provided interface `IX` in composite `X`.
- Because it is an implementation class, class `FacadeX` is located in package `x.impl` and has package visibility.
- The parameter in the constructor of class `FacadeX` allows objects of this class to store references to the object Manager of the component (line 7).

- In order to implement method `mx()` of interface `IX`, class `FacadeX` retrieves the internal component `A` through method `getInternalComponent` of class `Manager` (line 9), retrieves from it interface `IA` (line 10) and invokes its method `ma()` (line 11).

```
1  package x.impl;
2  import x.spec.prov.IX;
3  import a.spec.prov.IA;
4
5  class FacadeX implements IX {
6    private Manager manager;
7    FacadeX(Manager mgr) { this.manager = mgr; }
8    public void mx() {
9      IManager a = this.manager.getInternalComponent("a");
10     IA ia = (IA) a.getProvidedInterface("IA");
11     ia.ma();
12   }
13 }
```

Figure 19: Class FacadeX in composite component X from Figure 17

3.4.3 Classes `Adapter` in a composite component

Classes `Adapter` in a composite component adapt the required interface of the composite to the required interface of the internal components. In Figure 17, required interface `IBReq` of internal component `B` is of type `b.spec.req.IBReq`, while the required interface of the composite is of type `x.spec.req.IXReq`. Class `AdapterXReq` is depicted in Figure 20.

- Class `AdapterXReq` implements required interface `IBReq` and uses required interface `IXReq`.
- Class `AdapterXReq` obtains a reference to the component's class `Manager` through the constructor (lines 5 and 6).
- In order to implement method `mb()` of required interface `IBReq`, class `AdapterXReq` retrieves required interface `IXReq` of the composite (line 8) and invokes on it the method `mx()` (line 9).

3.4.4 Class `ComponentFactory` in a composite COSMOS* component

In a similar manner to class `ComponentFactory` in an elementary component (Section 3.2.1), class `ComponentFactory` in a COSMOS* composite component has a method `createInstance()`, responsible for creating an instance of the component.

Figure 21 depicts class `ComponentFactory` in composite component `X` from Figure 17.

```
1  package x.impl;
2  import b.spec.req.IBReq;
3  import x.spec.req.IXReq;
4  class AdapterXReq implements IBReq {
5    private Manager manager;
6    AdapterXReq(Manager mgr) { this.manager = mgr; }
7    public String mb() {
8      IXReq ixreq = this.manager.getRequiredInterface("IXReq");
9      return ixreq.mx();
10   }
11 }
```

Figure 20: Class AdapterXReq in composite component X from Figure 17

- The method `createInstance()` instantiates the internal components `A` and `B` and connector `AB` through their classes `ComponentFactory` (lines 5 to 7). Components are connected afterwards through calls to methods `setRequiredInterface` and `setProvidedInterface` (lines 9 to 10).
- The component's class `Manager` is instantiated (line 12). Each component is stored within the object `Manager` as an internal component through calls to method `setInternalComponent` (lines 13 to 15).
- Finally, class `AdapterXReq` is instantiated with a reference to the `Manager` object and is connected to the required interface of internal component `B` (line 17).

3.5 COSMOS* System components

A **system component** is a COSMOS* component which is a complete software system that can be deployed and executed. A COSMOS* component which is not a system component cannot be executed by itself; instead, it is instantiated as an internal component of a composite component and its interfaces are connected to other internal components through connectors. A **system component**, on the other hand, provides an entry point which allows its execution environment to execute it. Also, because a system component may be deployed and executed, according to the terminology presented in [10], it may also be called a **concrete architectural configuration**.

The entry point of a stand-alone software system written in Java is a method *main*, with the signature `public static void main(String[])`. A method *main* is also used as an entry point in other languages, such as C++ and C#. In the COSMOS* model, the class `ComponentFactory` of a system component has a method *main*, which is responsible for instantiating the system component, and running one of its interfaces. Figure 22 depicts the class `ComponentFactory` of the system component `X` from Figure 2:

In Figure 22, method `main` in class `ComponentFactory` is the entry point to the system. This method instantiates component `X` (line 10), retrieves its provided interface `IX` (line 11) and invokes its method `run()` (line 12), which executes the functionality of the system.

```
1  package x.impl;
2  import br.unicamp.ic.sed.cosmos.IManager;pp
3  public class ComponentFactory {
4    public static IManager createInstance() {
5      IManager a = a.impl.ComponentFactory.createInstance();
6      IManager b = b.impl.ComponentFactory.createInstance();
7      IManager ab = ab.ComponentFactory.createInstance();
8
9      a.setRequiredInterface("IAReq", ab.getProvidedInterface("IAReq"));
10     ab.setRequiredInterface("IB", b.getProvidedInterface("IB"));
11
12     Manager mgr = new Manager();
13     mgr.setInternalComponent("a", a);
14     mgr.setInternalComponent("b", b);
15     mgr.setInternalComponent("ab", ab);
16
17     b.setRequiredInterface("IBReq", new AdapterXReq(mgr));
18
19     return mgr;
20   }
21 }
```

Figure 21: Class `ComponentFactory` in composite component X in Figure 17

```
1  package x.impl;
2  import br.unicamp.ic.sed.cosmos.IManager;
3  public class ComponentFactory {
4    public static IManager createInstance() {
5          ...
6          /* the method createInstance() is described in Figure 21 */
7    }
8
9      public static void main(String args[]) {
10         IManager system = createInstance();
11         IX ix = (IX) system.getProvidedInterface("IX");
12         ix.run(); /* execute the system */
13     }
14 }
```

Figure 22: Class `ComponentFactory` in composite component X in Figure 17

The entry point of a system depends on its execution environment. For example, in a JavaEE servlet container [19], the method *main* of a class is ignored by the container, and other mechanisms are used to execute a system. The entry point of a COSMOS* system component must be adapted to the execution environment in which it will be deployed and executed. Section 5.1 details how to create a system component which will be run in a JavaEE servlet container.

3.6 Ports in the COSMOS* model

According to UML 2.0 [17], ports are groups of provided and required interfaces related to some feature of a component. Required interfaces not associated to a port are always used by the component for fulfilling any of its features; inversely, required interface associated to a port are used by the component only for fulfilling the features of the provided interfaces associated to the same port. In other words, required interfaces not associated to a port should always be connected; required interfaces associated to a port should only be connected if any provided interface associated to the same port is used. Likewise, provided interfaces associated to a port can only be used if all required interfaces associated to that port are connected. Thus if only some of the provided interfaces of a component are used, this mechanism reduces the number of required interfaces that should be connected.

In Figure 23, component `A` presents provided interfaces `IProv`, `IProv1` and `IProv2` and required interfaces `IReq`, `IReq1` and `IReq2`:

- Interfaces `IProv1` and `IReq1` are associated to port `p1`.
- Interfaces `IProv2` and `IReq2` are associated to port `p2`.
- Interface `IProv1` is used by component `B`, and required interface `IReq1` should therefore be connected.
- Provided interface `IProv2` is not used, and so it is not necessary to connect required interface `IReq2`.
- Required interface `IReq` is not associated to any port, and should always be connected.

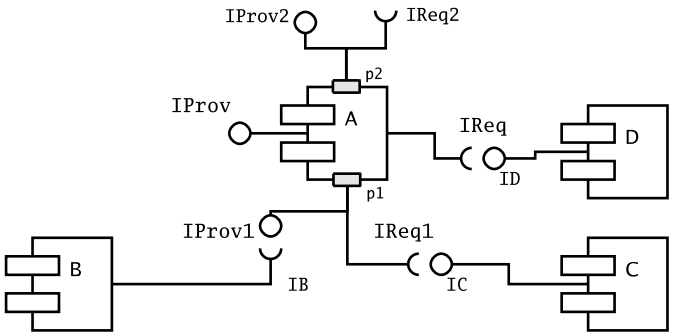

Figure 23: A COSMOS* component with ports

In order to retrieve a provided interface associated to a port, method `getProvidedInterface` of interface `IManager` is used, with the name of the port concatenated to the name of the interface as a parameter to the method. In Figure 24, in order to retrieve provided interface `IProv1`, associated to port `p1`, method `getProvidedInterface` is invoked with a parameter "p1:IProv1" (line 4).

```
1        IManager componentA  = a.impl.ComponentFactory.createInstance();
2        IManager connectorAB = ab.ComponentFactory.createInstance();
3
4        Object obj = componentA.getProvidedInterface("p1:IProv1");
5        connectorAB.setRequiredInterface("IB", obj);
```

Figure 24: Interface `IProv1` associated to port p1 is retrieved.

Likewise, the connection of a required interface to a port is made through method `setRequiredInterface` (Section 3.4.4). In Figure 25, method `setRequiredInterface` with parameter "p1:IReq1" connects interface `IReq1` associated to port `p1` in component `C`.

```
1        Object obj = componentC.getProvidedInterface("IC");
2        componentA.setRequiredInterface("p1:IReq1", obj);
```

Figure 25: Required interface IC associated to port p1 is connected.

3.7 The package `br.unicamp.ic.sed.cosmos`

The COSMOS* model specifies that COSMOS* components and connectors present the special interface `IManager` defined in the package `br.unicamp.ic.sed.cosmos`.

Figure 26 depicts the package `br.unicamp.ic.sed.cosmos`, with the definition of interface `IManager` and classes `AManager` and `AManagerComposite`. The packages `a.impl`, `b.impl`, `ab` and `x.impl` are the implementation packages of the components in Figure 17. Class `Manager` in the connector `AB` and in elementary components `A` and `B` extends the class `AManager`; class `Manager` in the composite component `X` extends class `AManagerComposite`.

Class `AManager` provides a reusable implementation of the methods in interface `IManager` and methods `setProvidedInterface` and `setProvidedInterfaceType`; class `AManagerComposite` extends class `AManager` with methods `setInternalComponent` and `getInternalComponent` for composite components.

As stated in Section 3.2.2, only interface `IManager` is mandatory for COSMOS* components. Classes `AManager` and `AManagerComposite` provide an easy way for a component to implement interface `IManager`, however, this is an implementation detail, and is not mandatory of the COSMOS* model. The developer of a COSMOS* component is free to choose not to use these classes and provide his own implementation of interface `IManager`.

4 External dependencies in the COSMOS* model

A software component can have some dependencies which cannot be represented in the form of required interfaces [24]. In this case, the COSMOS* model does not support the declaration of such dependencies at run-time, and they should be described in the documentation shipped with the component.

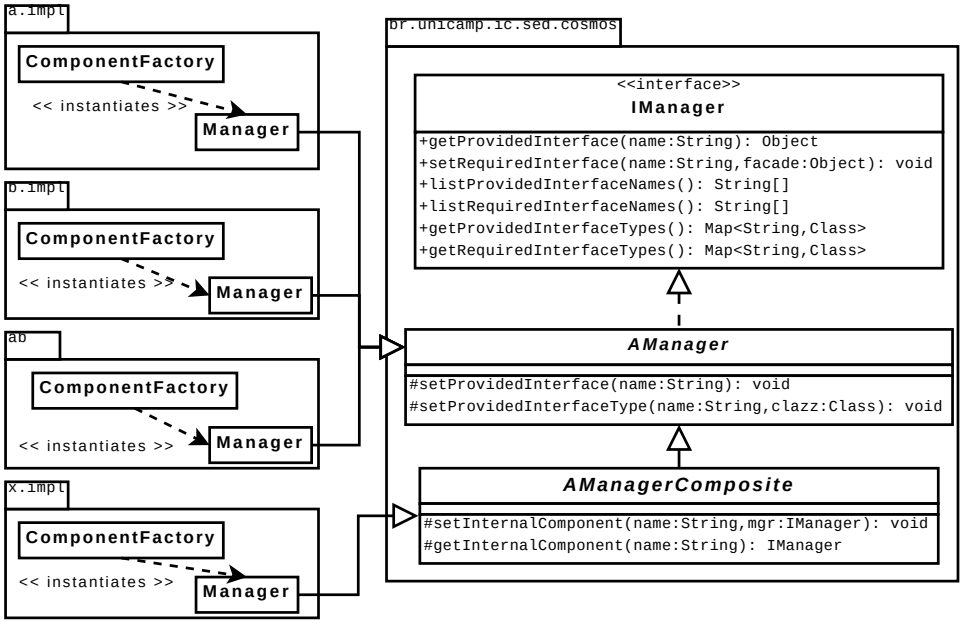

Figure 26: The interface `IManager` in package `br.unicamp.ic.sed.cosmos`

4.1 Environment dependencies

A component can depend on certain conditions of its execution environment, such as:

- **Structure of files and directory trees.** Some components may operate on files on the operating system and presume certain structures of directories. The correct structure of directories and files expected by the component should be declared in component's documentation.
- **Database table schema.** Some components may access a relational database and presume the existence of a certain schema of tables in the database in order to construct SQL commands which will be sent to the database. The table schema should be described in the component's documentation.

4.2 External libraries

The specification model of a COSMOS* component avoids direct references between components, and instead forces the communication between components to happen through their required and provided interfaces, thus minimizing the coupling between components (Section 3.1). However, implementation classes in a COSMOS* component may directly reference classes in third-party libraries, for the purpose of reusing a functionality provided by such libraries. Such a direct reference is qualified as a strong coupling between the component and the referenced classes, and is a dependency which cannot be expressed as a required interface. Also, in order to be employed by the COSMOS* component, a library must be available during the execution of the system.

A COSMOS* component may also reference directly another COSMOS* component. This happens in composite COSMOS* components (Section 3.4), when the composite component invokes directly the `ComponentFactory` class of its sub-components. Thus, there exists a strong coupling between the composite and its sub-components.

In the Java platform, making a component available means adding its file path to the *Class-Path* parameter, recognized by the Java Virtual Machine (JVM). The Class-Path of a Java application is a list of file and directory paths in which the JVM will search for a class, when it is referenced at run-time [20].

4.3 Data types in the COSMOS* model

The operations in the component interfaces may present in their signatures some data types defined by them and which are not present in the programming platform. As an example, a component for student registration may provide an operation `registerStudent(Student)`. In this operation, the parameter is of type `Student`. Data types may appear in the signature of an operation as a parameter or as a return value.

There are two approaches for the definition of data types, described by Silva Júnior in [7]:

4.3.1 Global package with data type definitions.

In this approach, there exists a global package which contains the definitions of all the data types, shared by all components in the system. This global package contains interfaces of the data types, and each component has an implementation of the interfaces of the data types it uses.

Figure 27 shows two components `UniversitySystem` and `StudentRegistration` of a system for student registration from a university.

Both components manipulate interface `IStudent`, defined in package `datatypes` and which represents a student record in the university database. Both components have a dependency on package `datatypes`. This solution has the advantage that instances of interface `IStudent` may transit between the components without the need for adaptations.

The disadvantage of this solution is that the global data types package becomes an external dependency of the component (Section 4).

Moreover, data types in package `datatypes` are interfaces, and each component using these data types contains an implementation of these interfaces, in their `impl` packages. Thus, in Figure 27, package `universitySystem.impl` of component `UniversitySystem` contains a class `StudentImpl`, which implements interface `IStudent`. Still in Figure 27, the implementation package of component `StudentRegistration` also contains a class `StudentImpl`, which also implements that interface.

It should be emphasized that the use of a global package already happens with the programming platform: classes such as `String` and `Date` in packages `java.lang` and `java.util` are referenced by the components and may transit between them without the need for adaptation; the Java platform, which define these classes, is a dependence of the components.

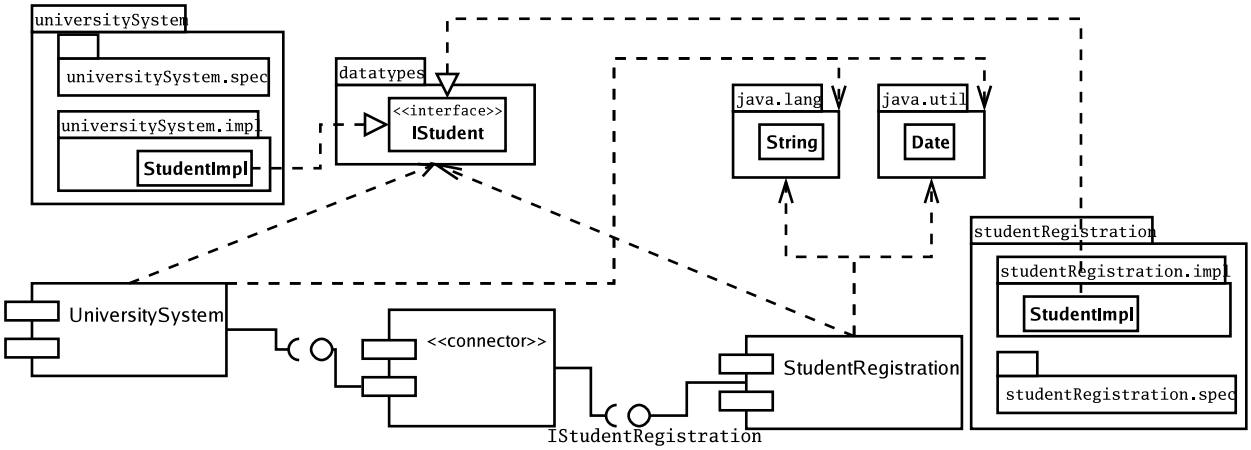

Figure 27: Data types defined in a global package.

The component depends on the global package with datatypes in a way similar to a dependency on third-party libraries (Section 4.2). This approach has the advantage of being easily implemented, since there is no need for adaptation of data types before they can transit between the interfaces. The disadvantage of this approach is that the reuse of a component in a different context implies in the reuse of the global data types package; moreover, an application reusing various different components will contain many data types packages, reducing system cohesion and hindering its maintenance.

4.3.2 Data types defined by a component.

In this approach, the specification of a component is extended to include the data types used by the component. Each component contains a package `spec.datatypes`, which contains interfaces for the data types used by the component and present in its interfaces.

In Figure 28, components `UniversitySystem` and `StudentRegistration` define interfaces `IStudent` in their packages `spec.datatypes` and use classes `StudentImpl` in their implementation package and which implement those interfaces. As such, a component has no dependencies on an external data types package. The connector contains classes `StudentImplUS` (which stands for "StudentImpl from package UniversitySystem") and `StudentImplSR` (which stands for "StudentImpl from package StudentRegistration"). These classes implement the data types from each component and the connector uses them to convert data coming from a component to the data type expected by the other.

The advantage of this approach is that a component is self-contained and does not present dependencies on a global data types package. Its disadvantage is that the development of a connector is harder, since it needs to adapt data types coming from different components.

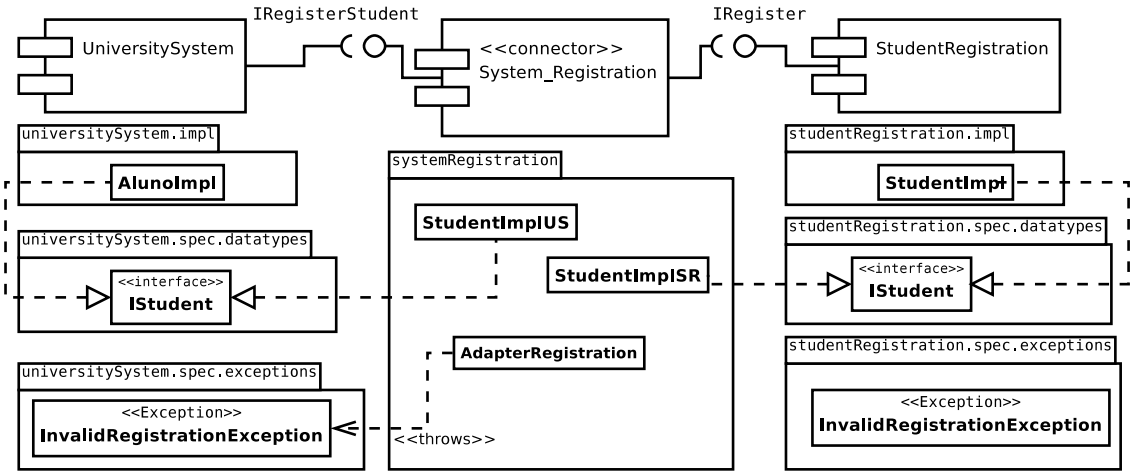

Figure 28: Data types defined by a component.

4.3.3 Exceptions

The signatures of operations in provided or required interfaces of a component can declare exceptions thrown by the operation. The types of the exception classes transit between the interfaces of the components. Like other data types, they can be declared in a global data type package or defined by the component; in the latter case, they should be declared in package `spec.exceptions`.

In Figure 28, operations in the required interfaces of component `UniversitySystem` present exception `InvalidRegistrationException`, which means the component is prepared for possible faults in registration operations. Class `InvalidRegistrationException` is defined in package `universitySystem.spec.exceptions`.

In the Java platform, exception classes do not implement interfaces. Thus, a connector between two components may reference directly the exception class and does not need to contain a class which implements the exception, as is done in the other data types.

Section 7 explains with more details the usage of exceptional behavior in the COSMOS* model.

5 Web applications based on the COSMOS* model

5.1 Web applications in the JavaEE platform

The COSMOS* model may be used in web applications, using the *Servlets* technology. Servlets are classes in the JavaEE platform which bridge the interaction between a web browser in the client and the server and form the core of the web development model in the JavaEE platform [4].

In the JavaEE platform, *Servlets* are responsible for serving request from clients: a servlet class receives a request from a client and sends back an answer. A web application may contain a servlet class for each different kind of request expected by the application.

Moreover, an application may depend on objects whose life cycle exceeds the lifetime of a request and which are shared by all classes in the application. A servlet container provides a specific space for such objects, called *application context*.

Web applications are deployed in a Servlet container. The deployment consists in describing in a *deployment descriptor* the servlets which will be invoked by the container. The deployment descriptor is a file `web.xml` file which contains meta-information on the application understood by the servlet container.

Upon reading the deployment descriptor of the application, the container copies the application classes and registers which servlets will be responsible for serving requests from clients. During deployment, it is also possible to register a class `Listener`, which will be invoked by the container when it starts the application, and thus, initializes the other classes in the application. A `Listener` class should implement interface `ServletContextListener` from package `javax.servlet` in the JavaEE platform.

5.2 Usage of servlets in the COSMOS* model

Figure 29 depicts the COSMOS* component `WebSystem`. Component `WebSystem` is a system component (Section 3.5) which can be deployed in a servlet container and has the following properties:

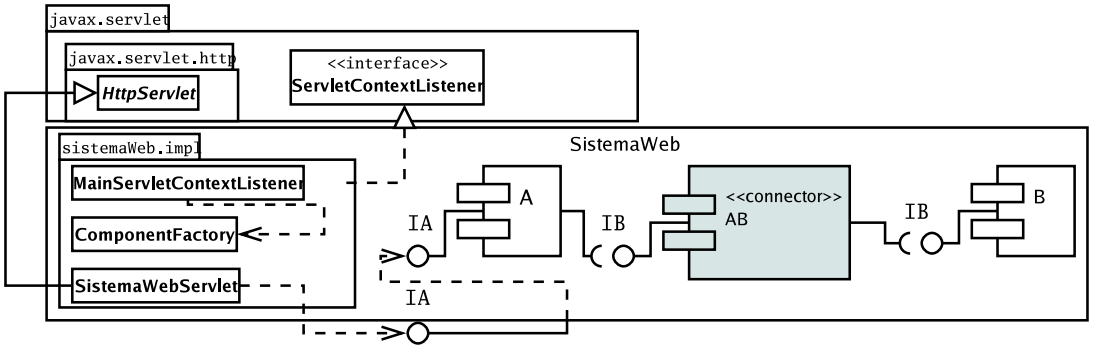

Figure 29: A web system with COSMOS* components.

- The entry point of this system component is the class `MainServletContextListener` in the package `webSystem.impl`, whose implementation is deployed in Figure 30.

  This class is registered within the container as a *listener* (Figure 32) and is responsible for instantiating component `WebSystem` by invoking method `createInstance()` in class `ComponentFactory` (line 7) when the application is started.

  After instantiating component `WebSystem`, the component's `Manager` object is stored in the application context in the container (line 8).

- The application has a servlet `WebSystemServlet`, also in package `impl` and whose implementation is depicted in Figure 31. In order to serve a request, this servlet:

1. retrieves the component's `Manager` object from the application context (line 7);
2. retrieves the component's provided interface `IA` from the `Manager` object and
3. invokes a method in the retrieved interface (line 9).

- Although classes `MainServletContextListener` are `WebSystemServlet` are implementation classes and reside in the component's `impl` package, they have public access. This is necessary in order for the servlet container to invoke them.

```
1  package webSystem.impl;
2  import javax.servlet.ServletContextListener;
3  import javax.servlet.ServletContextEvent;
4  import br.unicamp.ic.sed.cosmos.IManager;
5  public class MainServletContextListener implements ServletContextListener {
6      public void contextInitialized(ServletContextEvent event) {
7          IManager manager = webSystem.impl.ComponentFactory.createInstance();
8          event.getServletContext().setAttribute("websystem-manager", manager);
9      }
10 }
```

Figure 30: Class `MainServletContextListener` in component `WebSystem`

```
1  package webSystem.impl;
2  import javax.servlet.http.HttpServlet;
3  import br.unicamp.ic.sed.cosmos.IManager;
4  import sistemaWeb.spec.prov.IA;
5  public class WebSystemServlet extends HttpServlet {
6      public void doGet(HttpServletRequest request, HttpServletResponse response) {
7          IManager manager = this.getServletContext().getAttribute("websystem-manager");
8          IA ia = (IA) manager.getProvidedInterface("IA");
9          ia.ma();
10     }
11 }
```

Figure 31: Class `WebSystemServlet` in component WebSystem

Figure 32 depicts a snippet of the deployment descriptor web.xml. Class `MainServletContextListener` is stored as a *listener* (lines 2 and 3); class `WebSystemServlet` is registered as a servlet, responsible for serving requests ending with `/WebSystem` (lines 4 to 9).

```
1  ...
2  <listener>
3    <listener-class>webSystem.impl.MainServletContextListener</listener-class></listener>
4  <servlet>
5    <servlet-name>WebSystemServlet</servlet-name>
6    <servlet-class>webSystem.impl.CadastrarAlunoServlet</servlet-class></servlet>
7  <servlet-mapping>
8    <servlet-name>WebSystemServlet</servlet-name>
9    <url-pattern>/WebSystem</url-pattern></servlet-mapping>
```

Figure 32: A snippet of the deployment descriptor file web.xml

6 An example system based on the COSMOS* model

This Section presents a system for student registration used in a university as an example of a software system based on COSMOS* components.

6.1 A student registration system

The student registration system of a university is based on two components `StudentRegistrationUI`, which provides a graphical user interface, and `StudentRegistrationBusiness`, which executes business operations which allow to include, remove, search and update student records. Component `StudentRegistrationBusiness` depends on a data base so it can persist student data; component `DB` encapsulates operations for data base access and queries.

These components are depicted in Figure 33. Component `StudentRegistrationUI` provides interface `IStart`, which starts the system and presents the graphical user interface. Component `StudentRegistrationBusiness` provides interfaces `IRegisterStudent`, `IRemoveStudent`, `IUpdateStudent` and `ISearch`, responsible for including, removing, updating or searching for a student record, respectively. Component `DB` provides interface `IDB`, with methods for data base access.

Component `StudentRegistrationBusiness` has a required interface `IDB`, which represents its dependency of the component on operations for data base access. Component `StudentRegistrationUI` has a single required interface `IStudentOperations`, which concentrates the component's dependencies for operations for including, removing, searching and updating student records.

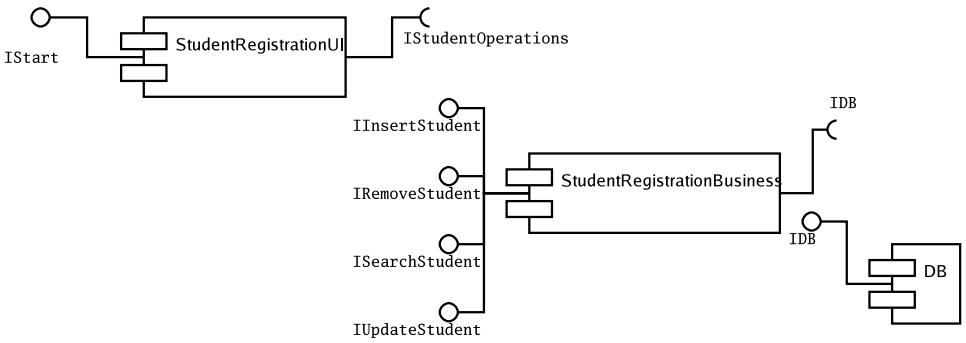

Figure 33: The components of a system for student registration.

6.2 Architectural configuration of the system component

Figure 34 depicts system `StudentRegistrationSystem`, composed by COSMOS* components `StudentRegistrationUI`, `StudentRegistrationBusiness` and `DB`. These components are connected by the COSMOS* connectors `UI_Business` and `Registration_DB`.

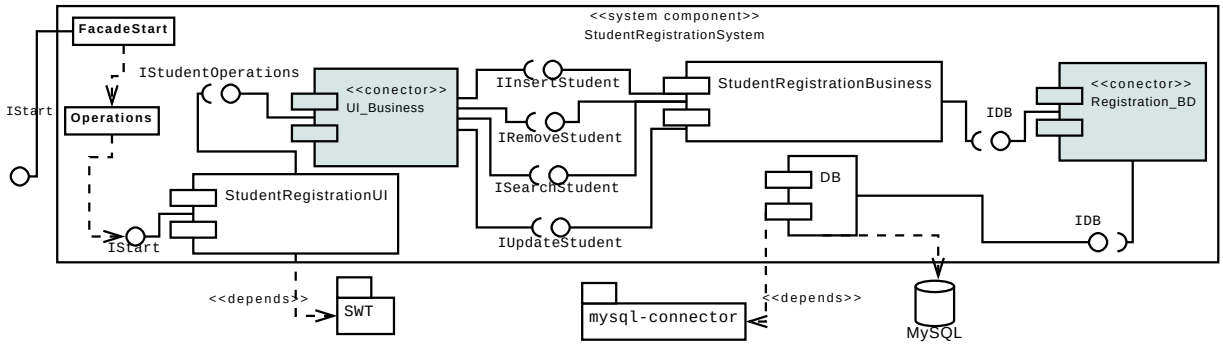

Figure 34: Architecture configuration of a system for student registration.

`StudentRegistration` is a system component (Section 3.5), which means it is both a COSMOS* component and a stand-alone system.

6.3 External Dependencies

As described in Section 4, the COSMOS* model does not provide a way for a component to declare dependencies on libraries at run-time; these should be stated in the component's documentation.

Figure 34 depicts the dependencies of the components in the student registration system: component `StudentRegistrationUI` uses the widget library SWT [12] for drawing the graphical user interface and component `DB` uses the mysql-connector [2] library which contains classes for data base access to a MySQL data base; component `DB` also requires that a MySQL data base is installed in the operating system and accessible by the component.

Libraries SWT and mysql-connector should be available on the application *Class-Path* (Section 4.2). The data base should contain the table schemes expected by the components.

6.4 System instantiation

Component `StudentRegistrationSystem` (Figure 34) is a system component. It provides the interface `IStart`, which instantiates and starts the system. According to the composition model (Section 3.4), component `StudentSystem` contains an implementation class `FacadeStart`, which implements interface `IStart` of component `StudentSystem` and uses the implementation class `Operations`. Class `Operations` uses provided interface `IStart` of component `StudentRegistrationUI`.

The class `ComponentFactory` of component `StudentRegistrationSystem` also contains a method *main*, as described in Section 3.5. The class `ComponentFactory` of the system component is depicted in Figure 35:

1. Method main instantiates the system component (line 13);

2. The interface `IStart` of the component is retrieved (line 14).

3. The method `start()` in interface `IStart`, which is responsible for running the system, is invoked (line 15).

```
1  package studentRegistrationSystem.impl;
2
3  import br.unicamp.ic.sed.cosmos.IManager;
4
5  import studentRegistrationSystem.spec.prov.IStart;
6
7  public class ComponentFactory {
8      public static IManager createInstance() {
9          /* instantiates the component. */
10     }
11
12     public static void main(String args[]) {
13         IManager mgrStudentSystem = createInstance();
14         IStart iStart = mgrStudentSystem.getProvidedInterface("IStart");
15         iStart.start();
16     }
17 }
```

Figure 35: Instantiating the system

6.5 Implementation of component interfaces

According to Section 3.2, each interface of a component is implemented by a Façade class. Figure 36 depicts Façade classes in the component's `impl` package .

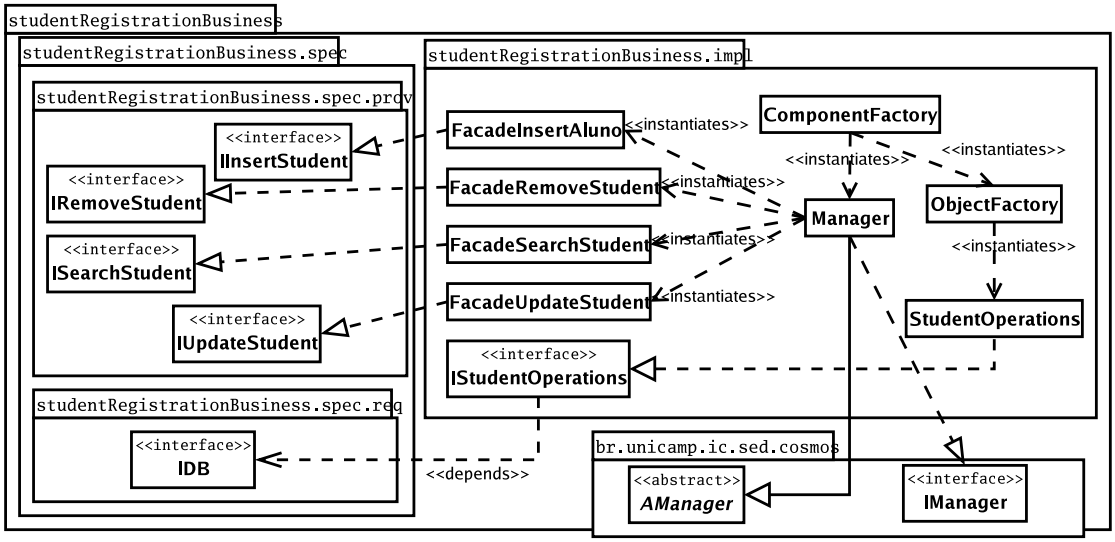

Figure 36: Implementation of component StudentRegistration

Figure 37 depicts the implementation of class Manager in component `StudentRegistrationBusiness`.

- The constructor in class `Manager` is responsible for instantiating the Façade classes (lines 7 to 10) and store them as provided interfaces (lines 12 to 15).
- The component's class `ObjectFactory` is instantiated and stored in the `Manager` object (line 15).
- Each Façade class is instantiated with a reference to the component's `Manager` object (referenced by **this**, in lines 7 to 10). This construction follows the *Dependency Injection* design pattern [15].

```
1  package studentRegistrationBusiness.impl;
2  import br.unicamp.ic.sed.cosmos.IManager;
3  import br.unicamp.ic.sed.cosmos.AManager;
4  class Manager {
5      private ObjectFactory objectfactory;
6      Manager() {
7          IIncludeStudent include = new FacadeIncludeStudent(this);
8          IRemoveStudent  remove  = new FacadeRemoveStudent(this);
9          ISearchStudent  search  = new FacadeSearchStudent(this);
10         IUpdateStudent  update  = new FacadeUpdateStudent(this);
11
12         this.setProvidedInterface("IIncludeStudent", include);
13         this.setProvidedInterface("IRemoveStudent", remove);
14         this.setProvidedInterface("ISearchStudent", search);
15         this.setProvidedInterface("IUpdateStudent", update);
16
17         this.objectFactory = new Objectfactory(this);
18     }
19 }
```

Figure 37: Class Manager in component StudentRegistrationBusiness

Figure 38 presents the code for class `FacadeRemoveStudent` in component `StudentRegistrationBusiness`. It implements provided interface `IRemoveStudent` (line 3). In order to erase a student record, it first retrieves the component's object factory (line 9), uses it to retrieve the internal interface for operations in the component (line 10), and invokes on this the operation for student record removal (line 11).

6.6 Implementation of the operations in component `StudentRegistrationBusiness`

The implementation of class `StudentOperations` in component `StudentRegistrationBusiness` is depicted in Figure 39. Class `StudentOperations` is responsible for the implementation of the features of the component: including, removing, updating and searching student records. Class `StudentOperations` communicates with the

```
1  package studentRegistrationBusiness.impl;
2  import studentRegistrationBusiness.spec.prov.IRemoveStudent;
3  class FacadeRemoveStudent implements IRemoveStudent {
4      private Manager manager;
5      FacadeRemoveStudent(Manager mgr) {
6          this.manager = mgr;
7      }
8      public void removeStudent(int id) {
9          ObjectFactory factory = this.manager.getObjectFactory();
10         IStudentOperations op = factory.getStudentOperations();
11         op.removeStudent(id);
12     }
13 }
```

Figure 38: Class FacadeRemoveStudent in component StudentRegistrationBusiness

data base through the component's required interface `IDB` (Figure 33). Access to the required interface is obtained with the `Manager` object of the component (line 8). In line 9, the component creates a dependency with the table schema in the data base (Section 4) because it assumes the existence of a table named "Students".

```
1  package studentRegistrationBusiness.impl;
2  class StudentOperations implements IStudentOperations {
3      private Manager manager;
4      StudentOperations(Manager mgr) {
5          this.manager = mgr;
6      }
7      void removeStudent(int id) {
8          IDB db = (IDB) this.manager.getRequiredInterface("IDB");
9          String sql = "delete from Students where id = " + id;
10         db.sendQuery(sql);
11     }
12     /*
13      * The implementation of the other operations is similar to operation removeStudent
14      */
15     void includeStudents(...) { ... }
16     void searchStudents(...) { ... }
17     void updateStudent(...) { ... }
18 }
```

Figure 39: Code for class StudentOperations

6.7 A web version of the student registration system

With a few changes, it is possible to change the student registration system from using a SWT-user interface to a web-based interface. The system then runs on a servlets container, as described in Section 5. Figure 40 depicts the changed system.

The component is renamed as `WebStudentSystem`; the method *main* in class

`ComponentFactory`, which served as the entry point of the component is replaced by a class `StudentSystemContextListener`, according to Section 3.5 so the system can be deployed in a servlet container. The provided interface `IStudentOperations` is used to serve requests from clients.

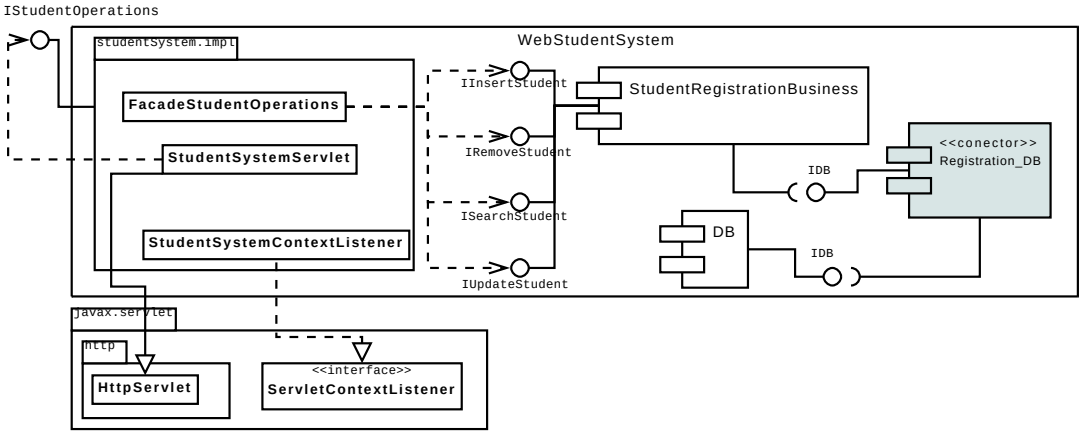

Figure 40: Architectural configuration of the web-based version of the system.

7 Exceptional behavior in the COSMOS* model

This section presents the architectural approach for building COSMOS* components with exception handling proposed by Guerra et al in [9]. This approach consists of inter-component and intra-component strategies for structuring exceptions thrown by components, and of an exception hierarchy which unifies both strategies.

The behavior of a component which implements the expected execution of a system is known as **normal behavior**. The corrective behavior of a component in front of exceptional situtations in known as **exceptional behavior**.

The normal behavior of the component `StudentRegistrationBusiness` described in Section 6.1 consists of operations for including, removing, updating and searching student records. The exception behavior of this component consists in the correction activities it realizes in face of some exceptional situation, such as a failure when accessing the data base.

7.1 Anticipated exceptional conditions

The specification of a component is composed by a normal specification, and, optionaly, by an exceptional specification. The normal specification specifies the behavior of the component under the hypothesis that no failures happen in the system. This hypothesis can rarely be guaranteed in a real system, and the occurence of a failure causes an unpredictable behavior in the component [9].

The exceptional specification specifies the behavior of the component under the hypothesis of certain failures. The exceptional specification determines which failures can be expected and a set of exceptions anticipated in the project of the component. In a fault-tolerant system [22], these exceptions aim to mask the error which caused the exceptional condition, or, when this is not possible, terminate the execution of the program in an orderly manner, signaling the insuccess of the operation [9].

Because it needs access to a data base, component `StudentRegistrationBusiness` in Section 6.1 should anticipate possible failures in data base access. Thus, the exceptional specification of the component contains an exception `DBException`, which represents a failure in data base access. Figure 41 depicts component `StudentRegistrationBusiness`. The exception specification package, `spec.exceptions` (Section 4.3.3), contains class `DBException`. This package also contains the exception `OperationFailedException`, which will be thrown by the component if he cannot execute an operation.

Figure 42 depicts the modified implementation of class `StudentOperations`, from Figure 39. The implementation classes of component `StudentRegistrationBusiness` wrap calls to the method `sendQuery` in interface `IDB` in a *try/catch* block (lines 14 to 17). The component makes three attempts to use the interface for data base access. At the end of the third attempt, the component throws an exception of type `OperationFailedException` (line 19).

7.2 Unanticipated exceptional conditions

Software systems are always subject to failures and, consequently, to errors not anticipated in their exceptional specification. The behavior of a component in an unanticipated excep-

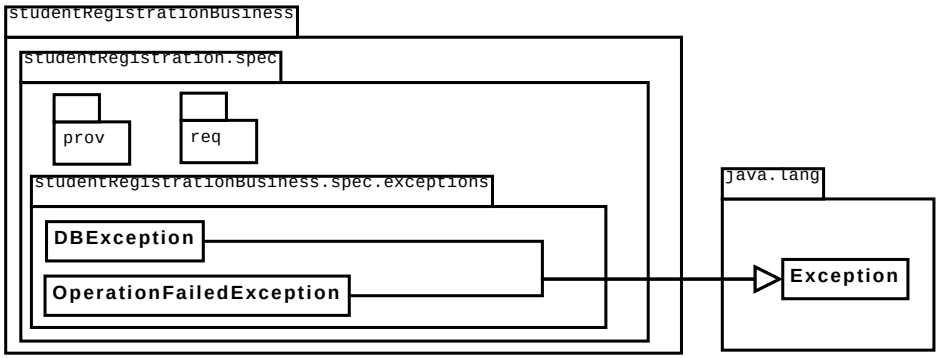

Figure 41: Component `StudentRegistrationBusiness` with a specification for exceptional behavior.

tional condition is entirely dependent of its implementation. A component implementation can, as an example, proceed in the execution of its operations without the error being detected and handled. This behavior results in propagation of the error to other parts of the system, causing incorrect results to be returned and new errors to be introduced in the system [9].

To avoid these problems, the implementation of a component can detect some kinds of errors unanticipated in the specification, known as **unanticipated exceptional conditions**. Thus, the exceptional behavior handles these exceptional conditions which, when they cannot be masked, hinder the execution of the component's operations [9].

In the Java platform, subclasses of class `RuntimeException` are used to signal unanticipated exceptional conditions. In Figure 43, class `StudentOperations` in component `StudentRegistrationBusiness` was modified to catch exceptions of type `RuntimeException`, unanticipated, and register their occurence with a logging mechanism from the platform (lines 4 and 5, 18 and 20); an exception of type `UnrecoverableFailureException` (Section 7.3) is thrown, to indicate that an unanticipated failure has just occured and the execution of the operation cannot proceed.

```
1  package studentRegistrationBusiness.impl;
2  import studentRegistrationBusiness.spec.exceptions.DBException;
3  import studentRegistrationBusiness.spec.exceptions.OperationFailedException;
4  class StudentOperations implements IStudentOperations {
5      private Manager manager;
6       StudentOperations(Manager mgr) {
7          this.manager = mgr;
8      }
9      public void removeStudent(int id)
10             throws OperationFailedException {
11         IDB db = (IDB) this.manager.getRequiredInterface("IDB");
12         String sql = "delete from Students where id = " + id;
13         for (int i = 0; i < 3; i++) {
14             try {
15                 db.sendQuery(sql);
16                 break;
17             } catch (DBException ex) {
18                 // Ignore this exception twice.
19                 if (i >= 2) throw new OperationFailedException(ex);
20             }
21         }
22     }
23 }
```

Figure 42: Introducing exceptional behavior in class StudentOperations

```
1  package studentRegistrationBusiness.impl;
2  import studentRegistrationBusiness.spec.exceptions.BdException;
3  import studentRegistrationBusiness.spec.exceptions.OperacaoImpossibilitadaException;
4  import java.util.logging.Logger;
5  import java.util.logging.Level;
6  class StudentOperations implements IStudentOperations {
7      private Manager manager;
8      StudentOperations(Manager mgr) {
9          this.manager = mgr;
10     }
11     public void removeStudent(int id) throws OperationFailedException {
12         try {
13             IDB db = (IDB) this.manager.getRequiredInterface("IDB");
14             String sql = "delete from Students where id = " + id;
15             db.sendQuery(sql);
16
17         } catch (RuntimeException ex) {
18             Logger logger = Logger.getAnonymousLogger();
19             logger.log(Level.SEVERE,
20                 "The system has failed when trying to remove a student record.");
21             throw new OperationFailedException(ex);
22         }
23     }
24 }
```

Figure 43: Catching unanticipated exceptional conditions.

7.3 A hierarchy of exception types

Unanticipated and anticipated exceptional conditions are materialized in an exception type hierarchy in the package `br.unicamp.ic.sed.cosmos.exceptions` as depicted in Figure 44 (a):

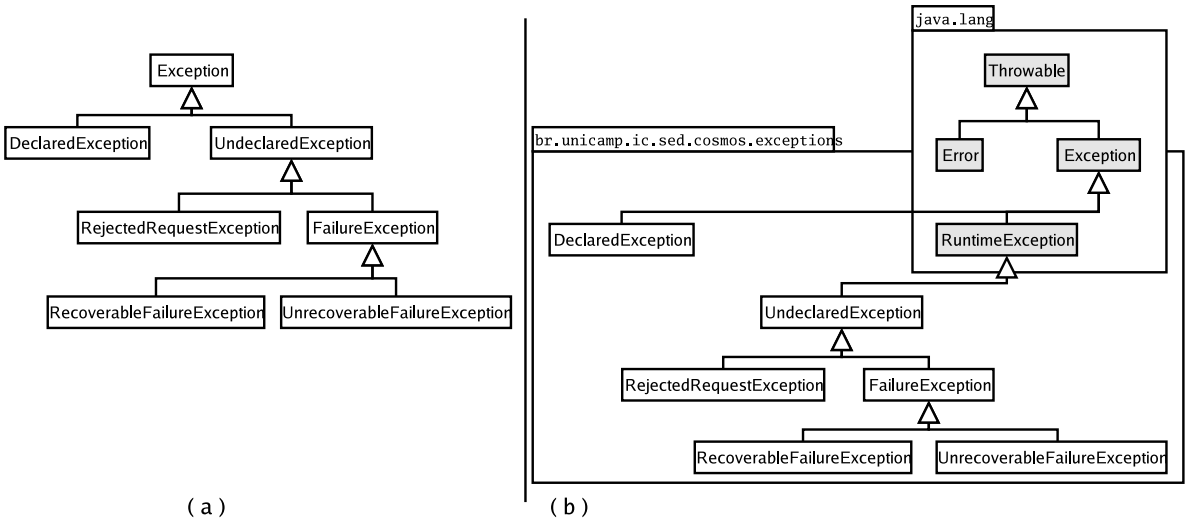

Figure 44: An exception type hierarchy (a) and its mapping to the Java platform (b).

- **Exception**: this is the root class of the exception type hierarchy
- **DeclaredException**: this class is the root of the hierarchy of types of exceptions declared in component specifications. Every declared exception in a component specification should be of a sub-type of `DeclaredException`.
- **UndeclaredException**: this class is the root of the hierarchy of types of undeclared exceptions. Exceptions not declared in a component specification should be of sub-types of `UndeclaredException`.
- **RejectedRequestException**: exceptions of this type are thrown when the contract for a pre-condition in the specification of a component is not met. The request sent to the component is rejected and no effect is produced in the internal state of the component.
- **FailureException**: this hierarchy is used to signal a failed attempt to execute a valid request, when a defect occurs in the component or in some other part of the system on which the component depends. The hierarchy is further divided in two other types: `RecoverableFailureException` and `UnrecoverableFailureException`.
- **RecoverableFailureException**: exceptions of this type are thrown when a defect in the implementation of the component prevents a request from being executed, but

no effect is produced in the internal state of the component and pre-conditions are kept.

- **UnrecoverableFailureException**: exceptions of this type are thrown when a component's implementation fails in executing an operation and there is no guarantee that no colateral effects produced by the operation have affected the internal state of the component or its environment.

7.4 Mapping the exception type hierarchy to the Java platform

In the Java platform, class `Exception` and its sub-classes are used to signal errors in a program [20]. Thus, operations that throw exceptions which are sub-types of class `Exception` should declare the thrown exception in their signatures. Clients of these operations can anticipate exceptions in a *try/catch()* block and provide in the *catch()* block the adequate handling, should an exception be thrown; or they can propagate the exception to their clients. In this case, these operations should declare the propagated exceptions on their signatures.

Class `RuntimeException` and its sub-classes represent errors in a program which result from implementation errors [21]. A common example is a null parameter being passed to an operation, which results in a `NullPointerException`, or an illegal parameter being passed to an operation, which results in an `IllegalArgumentException`. It is not necessary to declare in an operation's signature that it throws an Exception of type `RuntimeException`, and it is not necessary to wrap in a *try/catch* block a call to an operation which throws this type of exception.

Figure 44 (b) depicts the package `br.unicamp.ic.sed.cosmos.exceptions`, which contains the hierarchy of exception classes in the Java platform. Class `DeclaredException` is introduced, which inherits from base class `Exception` from the Java platform, so that components which throw this exception have to declare it in the signatures of their operations. Sub-type `UndeclaredException` is mapped to a new class which inherits from `RuntimeException`, so that operations can throw it without needing to declare them in their signatures and their clients do not need to wrap them in *try/catch* blocks.

7.5 Intra-component strategy

The intra-component strategy defines the internal responsibilities of a software component in face of exceptional conditions. The objective of this strategy is to ease the integration of these components in various different contexts, which may include different failure hypothesis. In order to do so, the system integrator should be able to predict the normal and exceptional behaviors of a component by checking only its specification.

The intra-component strategy is used in the construction of new components, when a component is developed from scratch ou when an existing component is adapted for a new context. In both cases, there exists a development effort which allows one to make decisions on the internal implementation of the component. The intra-component strategy is used to increase the possibilities of reuse of a component through the following actions:

1. Regarding the required interfaces of the component:

   (a) Declare explicitely, in the description of these interfaces, every exception type anticipated in the specification, and which extend `DeclaredException`. The implementation of any component must provide handlers for all these types of exceptions. Such handlers implement a behavior in accordance with the anticipated in the exceptional specification of the component.

   (b) The implementation of the component may include, optionaly, handlers for exceptions of type `UndeclaredException` (Figure 44) and which can be caught through these interfaces. These exceptions signal unanticipated exceptional conditions for which the implementation of the interface may offer some guarantee on the effects produced by the execution of the operation.

   As an example, the component implementation may use internally transactions with *commit* and *rollback* operations to guarantee that the internal state of the component is not affected by a failure in another component.

   (c) The implementation of the component may include, optionaly, a generic handler for other exception types not declared in these interfaces and which are not sub-types of `UndeclaredException`. Such exceptions signal unanticipated exceptional conditions for which the implementation of the interface does not offer any guarantee. Exceptions of this type should be considered as not recovered, with the same semantics of `UnrecoverableFailureException`.

   (d) Any exception resulting from a request emitted by the component and for which there isn't, in the implementation of the component, any appropriate handler, should cause the operation being executed by the component which depends on this required interface to fail.

2. Regarding the provided interface of the component:

   (a) Declare explicitely, in the definition of these interfaces, every type of exception that the component may throw in face of exceptional conditions anticipated in its specification. The implementation of any component should be able to detect and signal correctly these anticipated exceptional conditions.

   (b) Other unanticipated exceptional conditions, which cause an operation to fail, should be signaled by the implementation of the component through an exception of types `RejectedRequestException`, `RecoverableFailureException` or `UnrecoverableFailureException`, considering the specified semantics in the hierarchy of exception types from Figure 44 (a).

7.5.1 Applying the intra-component strategy to new components

The intra-component strategy extends the COSMOS model with two types of exception handlers (Figure 45): (i) handlers attached to the implementation classes, called **Application-**

Level Exception handlers (ALE handlers); and handlers attached to the component façades, called **Boundary-Level Exception handlers** (BLE handlers).

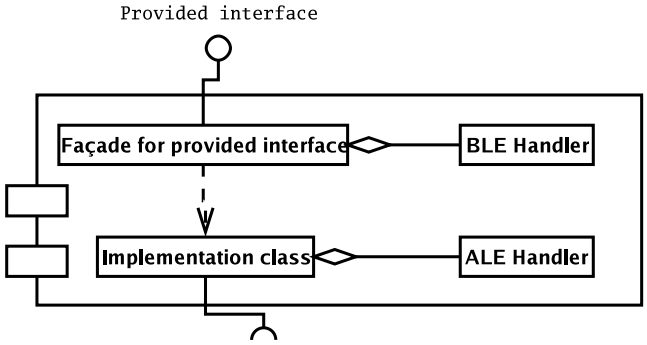

Figure 45: COSMOS component with exception handlers.

The responsibilities for the exceptional behavior of the component is divided among those elements as follows:

1. The implementation classes are responsible for detecting all exceptional conditions anticipated in the specification of the component.

2. The component's Façade classes may, as an implementation choice, detect exceptional conditions not anticipated which imply in a violation of pre-conditions and post-conditions specified for the component's operations.

3. The implementation classes may, also as an implementation choice, detect any kind of exceptional condition not anticipated which might interfere in the normal behavior of the component.

4. The exceptional conditions which cannot be masked are signaled to clients of the operations by the implementation class itself ou by a façade class which detects the condition.

5. Exceptional conditions which cannot be masked are signaled to an ALE or BLE handler inside the component.

6. ALE and BLE handlers are responsible for handling internal exceptions thrown, respectively, by implementation classes and façade classes in the component.

7. ALE handlers are also responsible for handling internal exceptions, which are exceptions thrown by the external façades of the component.

In Figure 46, two handlers were added to the student registration component from Figure 36 (Section 6.5). Class `HandlerInsertStudent` is responsible for handling exceptions detected by class `FacadeInsertStudent`; class `HandlerStudentOperations` is responsible for handling exceptions detected by class `StudentOperations`.

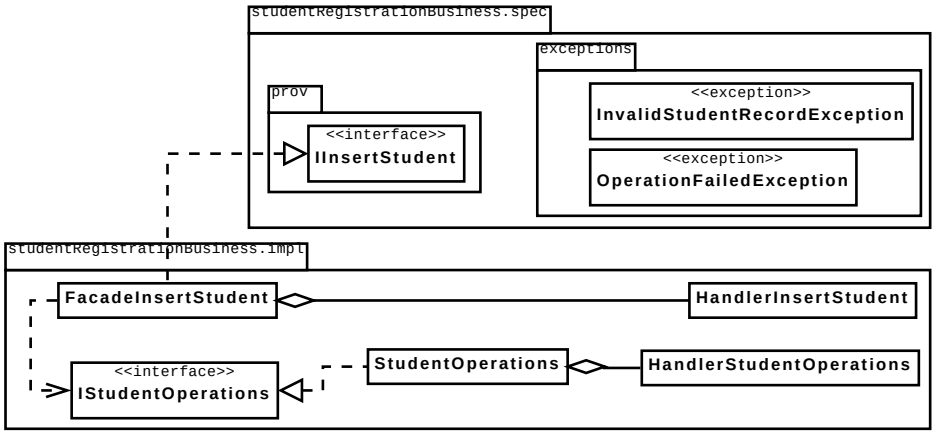

Figure 46: The student registration component with exception handlers.

The code in classes `FacadeInsertStudent` and `HandlerInsertStudent` is depicted in Figure 47.

- Class `FacadeInsertStudent` checks wether the parameter for the name of a student is missing in an insert operation (linha 12). Should this exceptional condition occur, this class invokes method `invalidStudent()` in the handler (line 13). Otherwise, the façade class invokes interface IStudentOperations through the object factory (Section 3.2.5) and uses it to insert the student (line 15).
- Class `HandlerInsertStudent` handles the exception condition. A message is written to the logging mechanism (line 8) and an exception is thrown which indicates a violation of the pre-condition (line 10).

The code in class `HandlerStudentOperations` is similar to the one in class `HandlerInsertStudent`, presented in Figure 47. Class `StudentOperations` invokes class `HandlerStudentOperations` when it detects an exceptional condition in the operation execution. Class `HandlerStudentOperations` handles the exceptional condition.

7.5.2 Applying the intra-component strategy to reused components

With reused components, the strategy is applied in a composite component (Section 3.4), which raps the reused component. In Figure 48, component `StudentRegistrationBusiness` is the one presented in Figure 36. It is reused inside a composite component which contains classes `FacadeInsertStudent` and `HandlerInsertStudent`.

1. The implementation of the reused component takes the responsibilities assigned to the implementation classes and to the ALE handlers, that is, the reused component should detect and handle exceptional conditions.
2. Class `HandlerInsertStudent` is a boundary-level exception (BLE) handler. Along with class `FacadeInsertStudent`, it is responsible for adapting the types of exceptions

```
1  package studentRegistrationBusiness.impl;
2  import studentRegistrationBusiness.spec.prov.IIncluirAluno;
3  import studentRegistrationBusiness.spec.datatypes.Aluno;
4  import studentRegistrationBusiness.spec.datatypes.exceptions.CadastroAlunoInvalidoException;
5
6  class FacadeInsertStudent implements IInsertStudent {
7      private HandlerInsertStudent handler;
8      FacadeInsertStudent(Manager manager) {
9          this.handler = new HandlerInsertStudent();
10     }
11     public void insertStudent(Student student) throws InvalidStudentRecordException {
12         if ("".equals(student.getName())) {
13             this.handler.invalidStudent();
14         }
15         this.manager.getObjectFactory().getOperacoesAlunos().insertStudent(student);
16     }
17 }
```

```
1  package studentRegistrationBusiness.impl;
2  import java.util.logging.Logger;
3  import java.util.logging.Level;
4  import studentRegistrationBusiness.spec.datatypes.exceptions.CadastroAlunoInvalidoException;
5
6  class HandlerInsertStudent {
7      void invalidStudent() throws InvalidStudentRecordException {
8          Logger.getAnonymousLogger().log(Level.SEVERE,
9              "Error: got invalid parameter when attempting to insert a student.")
10         throw new InvalidStudentRecordException();
11     }
12 }
```

Figure 47: Code in classes `FacadeInsertStudent` and `HandlerInsertStudent` from Figure 46.

thrown by the reused component into an exception type from the hierarchy described in Section 7.3. In this adaptation process, exceptions which are not of types declared in the specification of its interfaces are converted to a sub-type of `UndeclaredException`.

3. Classes `FacadeInsertStudent` and `HandlerInsertStudent` also take the responsibilities assigned, in the previous case, to the façade class of the components and to the BLE handlers, respectively.

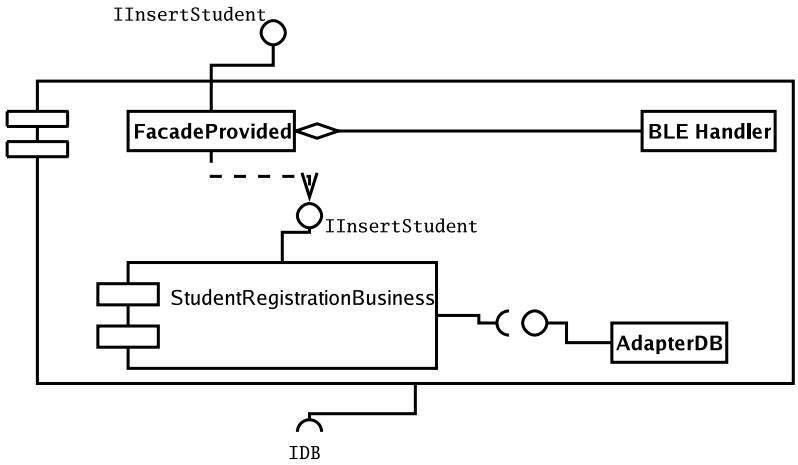

Figure 48: Adaptation of the exceptional interfaces of an existing component.

7.6 Inter-component strategy

The inter-component strategy handles the integration of components for the creation of a new system. This strategy is based on exception handlers attached to connectors, and which are specific for each configuration (Figure 49). These handlers, called Connector-Level Exception (CLE) handlers, are responsible for:

1. Handling configuration exceptions, which cannot be treated in the context of an isolated component. An example of this kind of error handling is the use of redundant components in an architectural configuration.

2. Adapting the exceptional interfaces of the components, converting the types of exceptions thrown by a server component into an exception of a type declared in the required interfaces of its clients.

3. Solve architectural conflicts between the hypothesis of failures of two or more components which communicate through the connector. This type of conflict happens when, in response to a request, the implementation of the components that serves the request throws an exception signaling an exceptional condition which is not anticipated in the specification of the component which made the request.

A connector should provide appropriate handlers for exceptions of any type that may be thrown through the provided interfaces of the components. This includes handlers for (1) the types of exceptions declared in the specification of these interfaecs, which are sub-types of `DeclaredException`; (2) the types of exceptions not declared in these specifications, but which may be anticipated in the implementation, which are the types `RejectedRequestException`, `RecoverableFailureException` and `UnrecoverableFailureException`; and (3) other types of exceptions which may be thrown by an implementation, in unanticipated conditions.

The CLE connectors implement the exceptional behavior specified for the system in its architectural level, which is independent of specific implementations of its components. A CLE handler may be able, in certain cases, to mask the exception and reestablish a normal condition of operation and sending a normal result to the client of the operation. In case it is not possible to mask the exception, the CLE handler throws a new exception for the operation client, adapting the type of the thrown exception to the specifications of the required interface of that client.

Figure 49 depicts a generic configuration, with two components `A` (client) and `B` (server), and a connector `AB`, responsible for translating the types of the exceptions in interfaces `A` and `B`. Figure 50 defines the general rules for determining the type of the exception to be thrown by the CLE handler in the connector.

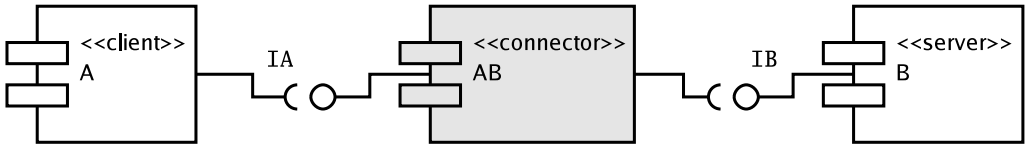

Figure 49: Architectural configuration with an exceptional connector

Type of the exception thrown by the server	**Type of the exception propagated by the connector**
The specifications of interfaces `IA` and `IB` declare that they throw `E1`.	Interface `IA` in connector `AB` throws `E1` (the exception may be propagated automatically).
The specification of interface `IB` declares that it throws `E1` and the specification of interface `IB` declare that it throws a type `E2` corresponding to the same semantics of defect. Example: `E2` is a supertype of `E1`	Interface `IA` in connector `AB` throws `E2`
The specification of interface `IB` declares that it throws `E1`, for which there is no corresponding exception type in the specification of interface `IA` with the same semantics of defect.	Interface `IA` in connector `AB` throws a sub-type of one of the exception types `RejectedRequestException`, `RecoverableFailureException` or `UnrecoverableFailureException`, according to the semantics of defect specified for `E1`
The specification of interface `IB` declares that it throws `E1` which is a sub-type of `UndeclaredException`	Interface `IA` in connector `AB` throws `E1` (the exception may be propagated automatically).

Figure 50: Rules for translating exception types by a CLE handler

7.7 An example of using exceptional connectors.

Figure 51 depicts a change in the student registration system described in Section 6: component `DB` was replaced with two new components, `MySQL_DB`, which connects to a MySQL data base [1]; and `PostgreSQL_DB`, which connects to a PostgreSQL data base [18]. The implementation of connector `Registration_DB` consists in using the interfaces of both DB components to store data. Thus, the data base is redundant, implemented by MySQL and PostgreSQL data bases.

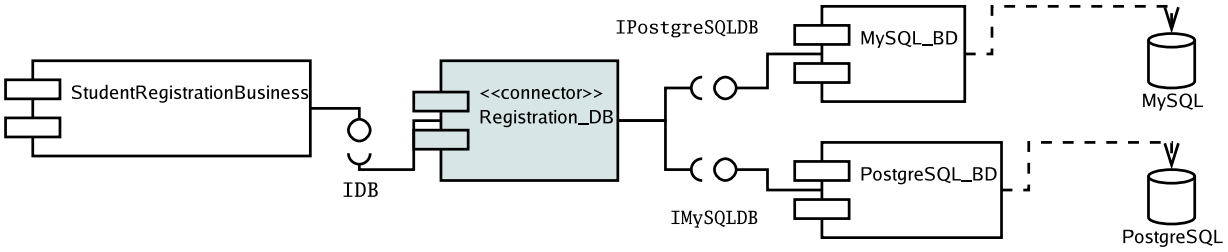

Figure 51: The student registration system with two data base components.

Redundant components may be employed in a system for the following reasons:

1. **Protection against errors in the component implementation:** if different implementations of a same specification of the component are used, should an implementation present defects caused by implementation errors, the connector may discard it and redirect the requests to another implementation.

2. **Choice between results from different implementations of a component:** the connector serves as a referee between different implementations of the component: the same request is sent to all components. The responses of all components should be equal. A different response characterizes as a defect in one of the components, which is discarded by the connector, and is no longer used.

8 The COSMOS* and EJB models

Entreprise JavaBeans (EJB) is the software component model in the Java Entreprise Edition (JavaEE) platform [19, 4]. The life cycle of EJB components are managed by an EJB container, which provides services for the components, such as transaction, distribution and access control. In the EJB model, component clients do not access them directly; a client accesses a component through a request to the container.

8.1 The EJB model

An EJB is composed by three Java classes:

- a *home interface*, which specifies infra-structure methods of the EJB. This interface has methods that make it possible to create instances of the EJB. This interface is required to extend interface `javax.ejb.EJBHome`.
- a *remote interface*, in which the developer specified the business methods of the application. This interface is required to extend interface `javax.ejb.EJBObject`.
- an implementation class, which provides implementation of the business methods of the EJB. The implementation class is required to implement one of the `javax.ejb.SessionBean`, `javax.ejb.EntityBean` or `javax.ejb.MessageDrivenBean` interfaces, which distinguish the class as an EJB component.

An EJB component is of one of three types [23]:

- A **Session Bean** represents a client in the EJB container. A client of an application interacts with the application by retrieving an instance of a Session Bean and invoking methods on it. A Session Bean is not persistent; that is, when the client has finished using it, it disappears and is no longer associated to the client.
- An **Entity Bean** represents an instance of an application object in the server. This object is persistent, in the sense that it is stored in a data base by the container.

  A client does not access directly the methods of an Entity Bean: instead, an Entity Bean is manipulated by a Session Bean. The client accesses the methods of the Session Bean, which then invokes the methods on the Entity Bean.
- A **Message-driven Bean** (MDB) allows a JavaEE container to process assynchronous messages. MDBs are striclty related to the Java Message Service (JMS) technology [23]. A MDB resides in the container and is executed when the container receives a JMS message.

Each EJB component is registered in the container under a Java Naming and Directory Interface name (JNDI name). The JNDI technology is a naming service which associates a name to an object in an EJB container. In order to use an EJB component, a client makes a request for the *home interface* of the EJB. This is an interface to an object implemented internally by the EJB container, which wraps the implementation of the EJB written by the developper of the EJB and which adds the transaction and distribution capabilities.

The names under which each client requests the home interface to the container are also registered in the container and usually differ from the JNDI names under which the EJB component is registered. The container is responsible for translating JNDI names and return an implementation of the home interface to each client.

8.2 A comparison between the COSMOS* and EJB models.

Table 1 summarizes the comparison between the COSMOS* and EJB models.

1. **Infra-structure of non-functional requirements**: In the COSMOS* model, non-functional requirements such as distribution, persistence and access control should be implemented by the application.

   In the EJB model, the container provides an infra-structure for non-functional requirements.

2. **Specification of provided interfaces**: In the COSMOS* model, provided interfaces are defined in the `spec.prov` package of a component.

   In the EJB model, the remote interface of an EJB specifies the methods that a client can invoke on an EJB component. Thus, the remote interface is the only one provided by an EJB.

3. **Specification of required interfaces**: In the COSMOS* model, required interfaces of a component are defined in the `spec.req` package of a component.

   In the EJB model, an EJB component does not declare explicitely its dependecy for other components in the form of required interfaces. Instead, its references to other EJB's are registered in the container, along with the JNDI name of the EJB on which the former EJB depends. As such, there isn't a way to find the dependencies of an EJB component similar to the `getRequiredInterfaceNames` method (Section 3.2.4) of a COSMOS* component.

4. **Coupling**: In the COSMOS* model, the communication between two components happens through provided and required interfaces, with a connector (Section 3.3). In this manner, there is low coupling between COSMOS* components. To replace a component in the system implies in creating connectors between the new component and the other components in the system.

   In the EJB model, an EJB component communicates with another EJB component through his remote interface. Because of this, to reuse the first EJB in a new con-

Table 1: Comparing the COSMOS* and EJB models

	The COSMOS* model	The EJB model
(1) Infra-strucure of non-functional requirements	No	Yes (container)
(2) Specifies provided interfaces	Yes	Yes (Remote interface)
(3) Specifies required interfaces	Yes	No
(4) Coupling between components	Weak	Strong
(5) Sub-components	Yes	Yes
(6) Explicit connection between components	Yes (Instantiation code)	Yes (JNDI names)

text, it is necessary to include the second EJB, or a new EJB which implements the same remote interface, which can be unfeasible. This characterizes a strong coupling between EJB components.

5. **Composition**: Its is possible to create a composite component in both the EJB and COSMOS* models. The hierarchical composition of COSMOS* components is described in Section 3.4. In the EJB model, an EJB can instantiate other components which are its internal parts.

6. **Explicit connection between components**: In the COSMOS* model, the connection between components and connectors is made by a composite component, with calls to methods `getProvidedInterface()` and `setRequiredInterface()` of the internal components (Section 3.4).

   The connection between an EJB component and its clients is evidenced by the JNDI names and references registered in the container. The connection between components depends on two factors: the registration of names in the container and the request by a component in the client source code. Differently from the COSMOS* model, in which the connection happens only at the instantiation time, in the EJB model, new connections may occur when the program is executing.

8.3 Access to an EJB component from a COSMOS* component

A COSMOS* component may instantiate and use an existing EJB component. In this case, the COSMOS* component accesses classes and methods defined in the EJB: this characterizes the EJB as an external dependency (Section 4.2).

In Figure 52, component `StudentRegistrationBusiness` from Section 6.1 was changed to interact with EJB component `StudentEJB`. The persistence of student data is realized by the latter, and interface `IDB` of component `StudentRegistrationBusiness` is discarded.

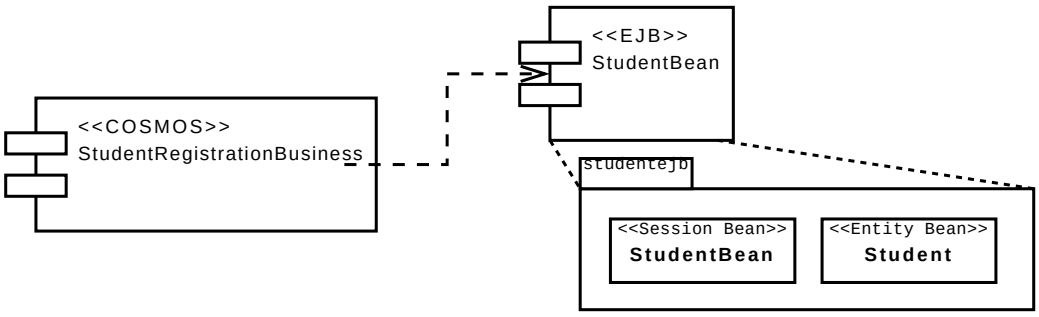

Figure 52: A COSMOS* component accesses an EJB component.

Component `StudentEJB` has a package `studentejb` with two classes, `Student` and `StudentBean` (Figure 52). Class `Student` is an *Entity Bean*; class `StudentBean` is a *Session Bean*, with methods that allow persisting and retrieving `Student` objects.

In Figure 53, method `listStudents()` of class `StudentOperations` in component `StudentRegistrationBusiness` obtains with the container a reference to the `StudentBean`

```
1  package studentRegistrationBusiness.impl;
2  import java.util.List;
3  import javax.naming.InitialContext;
4  class StudentOperations {
5      List listStudents() {
6          InitialContext ic = new InitialContext();
7          StudentBean studentBean = (StudentBean) ic.lookup("studentejb.StudentBean");
8          List allStudents = studentejb.findStudents();
9          return allStudents;
10     }
11     ...
12 }
```

Figure 53: Class StudentOperations in component StudentRegistrationBusiness

session bean (lines 6 and 7). Session bean `StudentBean` is registered in the container under name "studentejb.StudentBean"; this name is passed to class `InitialContext`, which allows to obtain a reference to an EJB in the container (line 7). Then, method `findStudents()` is invoked, to return a list of students (lines 8 to 9).

8.4 COSMOS* + EJB: Using EJB in the construction of COSMOS* components

A COSMOS* component may use the infra-strucure of an EJB container in its implementation. COSMOS* components differ from EJB components in the manner they locate an instance of a component. In the COSMOS* model, when a component is instantiated through method `createInstance()`, its classes are instantiated in the same Java virtual machine than the client class. In the EJB model, a component is instantiated by the container, in possibly a different JVM than its client, and the client class communicates with it through a remote interface; the client class does not know the location of the EJB component. In short, EJB components are distributed.

Section 8.4.1 e 8.4.2 incorporate in the implementation of COSMOS* components elements of the EJB technology which accord them location transparency properties. Such components are called COSMOS* + EJB components.

8.4.1 COSMOS* + EJB 2.1: COSMOS* components with EJB model, version 2.1

The class `Manager` of a COSMOS* component is replaced with a *Session Bean*. According to version 2.1 of the specification of the EJB technology [4], three classes should be created (Figure 54):

- A *remote interface* `Manager`, which extends interfaces `javax.ejb.EJBObject` and `IManager`.
- A *remote home interface* `ManagerHome`, responsible for creating a new instance of the EJB. This interface extends interface `javax.ejb.EJBHome`;

- A class `ManagerBean` that contains the implementation of the Session Bean and which implements interface `javax.ejb.SessionBean`.

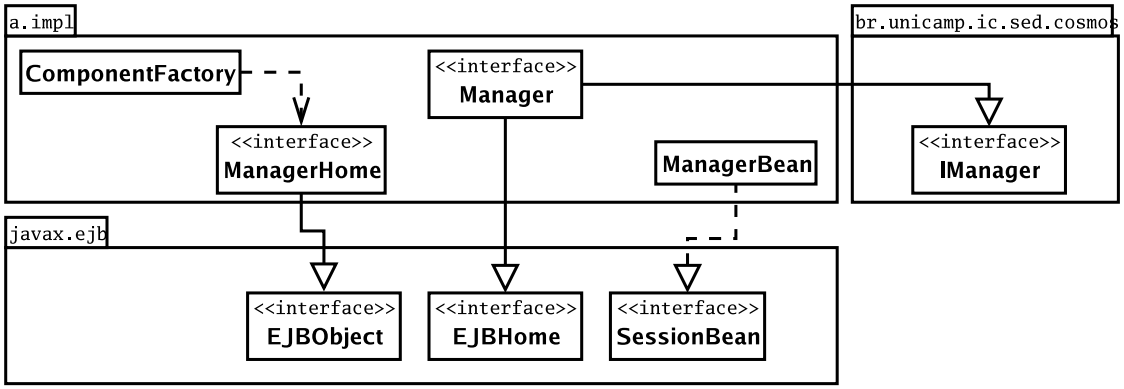

Figure 54: Class Manager as a Session Bean in EJB version 2.1

Figure 55 depicts the code in class `ComponentFactory` in a COSMOS*+EJB 2.1 component `A`. Session Bean `Manager` is deployed in the EJB container under JNDI name "a.impl.ManagerHome". In order to instantiate bean `Manager`, class `ComponentFactory` of the component retrieves with the container an instance of interface `ManagerHome` through its JNDI name (line 10); then, it invokes method `create` on this home interface. This creates an instance of the session bean, which is the `IManager` object returned (line 15).

```
1  package a.impl;
2  import javax.naming.Context;
3  import javax.naming.InitialContext;
4  import javax.rmi.PortableRemoteObject;
5  import br.unicamp.ic.sed.cosmos.IManager;
6  public class ComponentFactory {
7      public static IManager createInstance() {
8          Context initial = new InitialContext();
9          Context env = (Context) initial.lookup("java:comp/env");
10         Object objref = env.lookup("a.impl.ManagerHome");
11
12         ManagerHome home = (ManagerHome)
13                 PortableRemoteObject.narrow(objref, ManagerHome.class);
14
15         Manager mgr = home.create();
16         return mgr;
17     }
18 }
```

Figure 55: Instantiating the Manager object of a COSMOS* + EJB 2.1 component

The signature of method `createInstance()` in class `ComponentFactory` and interface `IManager` remain unchanged. Thus, there is no visible difference between the instantiation of a COSMOS*+EJB component and the instantiation of a pure COSMOS* component.

8.4.2 COSMOS* + EJB 3.0: COSMOS* component with EJB 3.0

Version 3.0 of the EJB technology brought great changes in relation to version 2.1 [23]. In particular, the implementation of a Session Bean now uses only one class, with annotations, instead of the home and remote interfaces of version 2.1.

A COSMOS*+EJB 3.0 component is different from a COSMOS* component in regard to its interface `IManager`: a COSMOS*+EJB 3.0 component defines an interface `IManager` in package `spec.prov`, and which extends the interface `IManager` in package `br.unicamp.ic.sed.cosmos` (Figure 56). The interface `IManager` in package `spec.prov` is the *identifier IManager interface* of a COSMOS*+EJB 3.0 component.

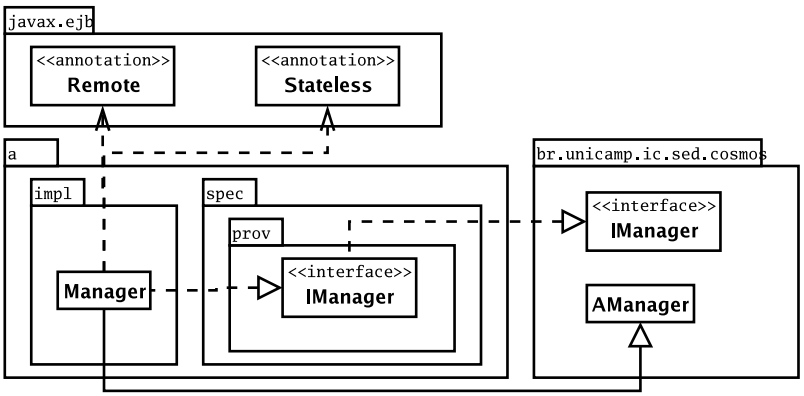

Figure 56: The interface `IManager` of a COSMOS* + EJB component

Differently from version 2.1, in version 3.0 it is not necessary to specify a JNDI name for an EJB. Instead, a Session Bean receives the annotation `Remote`, which determines which remote interface it implements. In the container, the JNDI name of the EJB is the same name of the remote interface indicated in the `Remote` annotation. Thus, class `ComponentFactory` of a COSMOS*+EJB component uses the fully qualified name of the identifier IManager interface of a COSMOS* component to instantiate it (Figure 57). Because the identifier IManager interface of a COSMOS*+EJB extends interface `IManager` from package `br.unicamp.ic.sed.cosmos`, the signature of method `createInstance` in class `ComponentFactory` is not changed.

To make a COSMOS*+EJB component a Session Bean, it is necessary to change class `Manager`, inserting into it annotations `javax.ejb.Stateless` and `javax.ejb.Remote`, as shown in Figure 58 (lines 4 and 5).

8.4.3 The System Component in the COSMOS*+EJB model

The changes in Sections 8.4.1 and 8.4.2 do not modify the signature of class `ComponentFactory` nor interface `IManager` of a component. Thus, the instantiation of the system still happens with a class `Main` ou through a Servlet: each component is instantiated by its method `createInstance` and they are connected through methods `getProvidedInterface` and `setRequiredInterface` from interface `IManager`.

```
1 package a.impl;
2
3 public class ComponentFactory {
4     public br.unicamp.ic.sed.cosmos.IManager createInstance() {
5         Context context = new InitialContext();
6         IManager mgr = (IManager) context.lookup("a.spec.prov.IManager");
7         return mgr;
8     }
9 }
```

Figure 57: Instantiating the Manager of a COSMOS*+EJB 3.0 component

```
1 package a.impl;
2 import javax.ejb.Stateless;
3 import javax.ejb.Remote;
4 @Stateless
5 @Remote(a.spec.prov.IManager.class)
6 public class Manager implements a.spec.prov.IManager {
7     /* The methods in class Manager are described in Section 3.2 */
8 }
```

Figure 58: Instantiating a composite component

In Figure 59, components `A` and `B` are COSMOS* + EJB component, built according to Section 8.4.2. However, the instantiation of these components is equal to the instantiation of pure COSMOS* components. This is shown in the code in Figure 60: the composite component `X` instantiates its internal components by calls to method `createInstance` in each class `ComponentFactory` and connects them through method `setRequiredInterface`; the instantiation code does not reference details of the EJB technology.

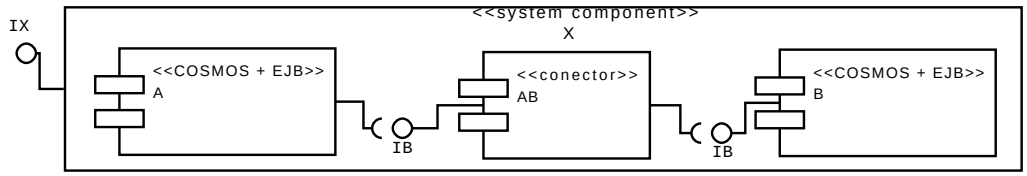

Figure 59: Uma arquitetura de componentes COSMOS* + EJB

```
1  package x.impl;
2
3  import br.unicamp.ic.sed.cosmos.IManager;
4
5  public class ComponentFactory {
6      public static IManager createInstance() {
7          IManager mgrA = a.impl.ComponentFactory.createInstance();
8          IManager mgrB = b.impl.ComponentFactory.createInstance();
9          IManager mgrAB = ab.ComponentFactory.createInstance();
10
11         mgrA.setRequiredInterface("IB", mgrAB.getProvidedInterface("IB"));
12         mgrAB.setRequiredInterface("IB", mgrB.getProvidedInterface("IB"));
13
14         ...
15     }
16     public static void main(String args[]) { ... }
17 }
```

Figure 60: A concrete configuration with COSMOS*+EJB components

References

[1] MySQL AB. Mysql ab::the world's most popular open source database. online, September 2006.

[2] MySQL AB. Mysql connector / j, September 2006.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 2nd edition, 2003.

[4] Stephanie Bodoff, Dale Green, Kim Haase, Eric Jendrock, Monica Pawlan, and Beth Stearns. *The J2EE tutorial*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[5] Francois Bronsard, Douglas Bryan, W. Kozaczynski, Edy S. Liongosari, Jim Q. Ning, Asgeir Olafsson, and John W. Wetterstrand. Toward software plug-and-play. In *Proceedings of the 1997 symposium on Software reusability*, pages 19–29. ACM Press, 1997.

[6] John Cheesman and John Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[7] Moacir Caetano da Silva Júnior. COSMOS - um modelo de estruturação de componentes para sistemas orientados a objetos. Master's thesis, Universidade Estadual de Campinas (Unicamp), 2003.

[8] Moacir Caetano da Silva Júnior, Paulo Asterio de Castro Guerra, and Cecília M. F. Rubira. A java component model for evolving software systems. In *ASE*, pages 327–330. IEEE Computer Society, 2003.

[9] Paulo Astério de Castro Guerra. *Uma Abordagem Arquitetural Para Tolerância a Falhas Em Sistemas de Software Baseados Em Componentes*. PhD thesis, Instituto de Computação, Universidade Estadual de Campinas, June 2004.

[10] Paulo Astério de Castro Guerra, Tiago César Moronte, Rodrigo Teruo Tomita, and Cecília Mary Fischer Rubira. Um modelo conceitual e uma terminologia para o desenvolvimento baseado em componentes e centrado na arquitetura de software. In Regina Maria Maciel Braga, editor, *Workshop de Desenvolvimento Baseado em Componentes*, pages 41–48, November 2005.

[11] Desmond F. D'Souza and Alan Cameron Wills. *Objects, components, and frameworks with UML: the catalysis approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[12] Eclipse.org. Swt: The standard widget toolkit. online, September 2006.

[13] Python Software Foundation. Python programming language – official website, August 2007.

[14] The Apache Software Foundation. Apache struts, September 2006. http://struts.apache.org/.

[15] Martin Fowler. Inversion of control containers and the dependency injection pattern. online, fevereiro 2007.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[17] Object Management Group. The unified modeling language, September 2006. http://www.omg.org/uml/.

[18] PostgreSQL Global Development Group. The world's most advanced open source database. online, January 2007.

[19] Sun Microsystems. Java platform, enterprise edition, março 2006. http://java.sun.com/javaee/index.jsp.

[20] Sun Microsystems. Java technology, março 2006. http://java.sun.com/j2se.

[21] Sun Microsystems. Unchecked exceptions - the controversy, Outubro 2006.

[22] Cecília Mary Fischer Rubira and Paulo Astério de Castro Guerra. *Desenvolvimento Baseado em Componentes e Confiabilidade*. (EDUEM), Editora da Universidade Estadual de Maringá, 2003.

[23] Rima Patel Sriganesh, Gerald Brose, and Micah Silverman. *Mastering Enterprise JavaBeans 3.0*. Wiley Publishing, Inc., 2006.

[24] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.