Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
and
Technische Universität Darmstadt - Germany
2003

Towards Caesar:
Aspect interfaces, their Bindings and Implementations

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Walter Augusto Werner

Promotor: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotors: Prof. Mira Mezini and Dr. Klaus Ostermann
(Technische Universität Darmstadt)

Abstract

Generic functionalities of applications are those concerns of the system that can be useful in many places within the same application domain, or even in other domains, playing distinct roles. Even though this kind of functionality is very common in software development, current object-oriented language technologies give only little means to deal with those functionalities.

Aspect-oriented programming has brought great advance for defining generic concerns and integrating them into existent applications by introducing the concept of *aspect*. However this abstraction loses object-oriented features when describing generic functionalities chiefly because in the level of the aspects and its integration into the base application goals of modularity are not completely achieved.

This thesis presents language support to the notion of *aspect interfaces* to improve the current technology in the context of Caesar programming language. Aspect interfaces are abstractions to help decoupling the aspect implementation from the aspect binding, as well as describing concerns as a set of abstractions that together define the functionality as a whole. In addition to that, new support is presented in order to integrate these concerns into base applications without changing their modular structure. With these new concepts introduced in the language, Caesar facilitates reuse and componentization of aspects.

By using aspect interfaces and their involved concepts, generic functionalities can be expressed as self-contained components organized in stable architectures. In this structure they communicate with the application that they get integrated into through the restricted and clearly defined aspect interface, thereby achieving goals of reusability and extendibility.

Acknowledgements

I would like to acknowledge, first and foremost, my parents Carlos and Glaci Werner and my sister Mariana Werner for the enduring love and emotional support they have been giving me during my whole life and chiefly in the period of this master. Thank you for believing and encouraging me, without your love and counsels it certainly would not be possible.

To my friends in Brazil whom have not left me alone. Thank you for keeping our friendship even across so many kilometres away. To my new friends from EMOOSE, we have become a family here. Thank you for all "eco" parties and dinners, for showing me the culture of your countries, and for every moments we have shared. From these lasts I would like first to specially thank João Del Valle and Fabiana Nogoseke for the great moments we have passed together and for their sincere friendship. Thank you João for the laughs, advices, cooking experiences, trips and for everything we shared this year. Thank you Fabiana for listening to me, for making this period so enjoyable and for everything you have done for me. I cannot forget also Francisca Muñoz for being my "sister" here, thank you for worrying about me when I got ill. Also my sincere thanks to Diego De Sogos and Cintia Fernandez for inviting me for dinners and making me have fun in the hardest moments.

Thanks to Frederic Pfeiffer for speaking portuguese to me, showing me such beautiful places in France and giving the chance of diving in the other side of the Atlantic ocean.

I also would like to acknowledge my advisors Prof. Mira Mezini and Dr. Klaus Ostermann for the great support they have provided during this period even when we were not close to each other. Special thanks to Dr. Klaus Ostermann for the time spent in tedious revisions. I am also very grateful to Dr. Jacques Noyé for his efforts to welcome me in the beginning of this thesis.

I also want to specially thank Dr. Carlos Maziero for so far teaching me and giving me the chance to follow EMOOSE. Thank you very much!

Table of Contents

List of Figures

Chapter 1

Introduction

The main goal of *modularity* in software engineering is to provide support to developers describing flexible software architectures made of *extendable* and *reusable* components. Modularity leads to the term **module** that can be defined as the basic unit of decomposition of the systems. Modules are not only little pieces of code as it could seem. In order to be reusable and extendible, modules must be self-contained and organized in stable architectures where they communicate with each other through restricted and clearly defined interfaces.

Software is extendible when its requirements are changed without implying a big modification to the system's module structure, or when new requirements are added without causing a huge reorganization of the system. Reusability deals with the use of general-purpose components in the software structure. It implies that these components should be defined in a generic way, so they could be just plugged in other systems, or even be instantiated many times in the same application.

An important modularity issue is the *separation of concerns principals*. Concerns are the concepts, goals or purposes of the software. Hence the separation of concerns principals say that language constructs should enable these concerns to be declared independent from each other, thereby allowing them to be described in such a way that they are reusable and extendable.

Trying to achieve a better modularization than those currently available in object-oriented technology, the authors of [9] define the term which has driven this work: *generic functionalities*. Generic functionalities of applications are those concerns of the software that can be useful in many places within the same application domain, or even in other domains, playing distinct roles. In this scenario, the functionalities might even be instantiated many times in a single application.

Describing generic functionalities of software calls for technology support to: define module constructions for components whose implementation is decoupled from any particular application, as well as easy remodularization of the software in order to plug these components in many applications in a no invasive[1] fashion.

These requirements of generic functionalities can be directly mapped to the requirements of modularity for describing reusable and extendible software. The first requirement of defining generic components is related to reusability in the sense that modules must be defined as independent entities that can somehow be reused in other programs. Regarding the second requirement of generic functionalities for easy remodularization, it can be associated with both reusability and extendibility. Reusability in order that it asks for an easy manner for reusable components be bound into the software, and extendibility because it calls for support to insert or change concerns without a huge remodularization of the software. Therefore, describing generic functionalities requires language support to achieve modularity.

Object-oriented languages are not well equipped to cope with the subtle problems that occur when integrating independently developed components. The implementation of those generic functionalities might be scattered across the modular structure of the applications, which directly affects the separation of concerns principals, and the functionality is not modular anymore.

This kind of situation is the main target of aspect-oriented programming. The mainstream of this programming technology is the concept of *join point interception* (JPI). Languages equipped with this concept support the definition of points in the execution of the base program to intercept allowing then their modification. These execution points and their modifications are declared in an abstraction called *aspect*. The aspect may be applied in the base application without physically changing its source code.

Aspect-oriented technology avoids hence the problem of defining functionalities, or aspects in the terms of this technology, scattered across the modular structure of the system. Thus, generic functionalities could be defined by means of aspects, however as argued in [10], more powerful means for structuring aspect code are needed on top of JPI-based models. The deficiencies involving this concept will be better explained in section 1.2.

This thesis presents the implementation of language support for *aspect interfaces*. This new concept, first described in [9] as *collaboration interface*, is an abstraction which improves the object technology support to represent generic functionalities. It is the technology answer to the requirement of these functionalities for defining generic components since it has the ability to represent an 'off-the-shelf' component ready to be included in an application.

[1] No invasive means that physical changes are not needed

The aspect interfaces differs from ordinary interfaces as they are known from Java in two points:

- They can be nested; hence the functionality can be defined as a set of abstractions which together build up a concern of the software. Please note that the ability to be nested here is not the same as it is known from Java inner classes. Being nested means here that these interfaces are really part of the outer component, thereby obliging classes that implement the enclosing interface to provide implementation of these nested interfaces as well. These differences will be highlighted in chapter 3.

- They define what is implemented by the component, the *provided contract*, which is what possible clients of the component can expect from that implementation. This part of the contract can be seen as what ordinary interfaces can express. However, aspect interfaces can also define what component implementations expect from their clients: the *expected contract*.

Hence, the implementation of aspect interfaces is broken up into two parts: *implementation* and *binding*. It has been divided in order to support a flexible integration of generic components. Even though the implementation of aspect interfaces is separated in two abstractions, both implementation and binding abstractions can access the whole aspect interface contract as their own.

Implementing an aspect interface means implementing the provided contract of the interface. These implementations are the realization of the generic functionality as a component which communicates with the clients through the aspect interface.

On the other hand, *binding* an aspect interface means implementing the expect contract of the interface. Bindings are the application-dependent part of the component. These units allow at the same time no invasive and flexible remodularization of the software. The model thus implements the second requirement from generic functionalities for supporting easy decomposition of the software.

In [9], this kind of remodularization is called *on-demand remodularization*. This term is used not only for the kind of decomposition described - without physical changes in the modular structure - but also to highlight another important feature of the model: the remodularization is applied only when it is demanded, so the modular structure of the software is changed only when the remodularization is explicitly applied.

Please note that the model of on-demand remodularization used here is object-based in contrast to the Hyperspace model [12] where it is class-based. There the decompositions applied are valid for all instances of the class, but here it affects only specified objects. It

gives a finer grain to control remodularization and allows single objects to be integrated to different components.

This thesis presents the implementation of aspect interfaces and their bindings and implementations in context of **Caesar** programming language in order to provide its complete realization. This implementation adapts these new abstractions to abstractions already available in Caesar. Therefore, these new abstractions become able to use the other concepts available in such programming language.

This thesis is structured as follows. In the next sections of this chapter, the example used in the other chapters is presented in section 1.1, after the problems are outlined in section 1.2, followed by the presentation of some related works in section 1.3. Chapter 2 explains the Caesar mechanisms. Chapter 3 describes what has been defined in the language in order to support aspect interfaces, bindings and implementations in Caesar. Chapter 4 shows how these concepts have been implemented in the context of Caesar. Finally, chapter 5 describes future work and gives a summary of this thesis.

1.1 Example - the Mediator pattern

This section presents the example that is used in the next chapters. The example chosen is the Mediator design pattern [3]. Patterns are good examples of generic functionalities of applications, because they represent a concept in terms of collaboration between objects independent of an application domain. Though they have been hugely reused in the design level, their implementations are often rewritten.

The intent of the Mediator pattern is to "Define an object that encapsulates how a set of objects interact" [3]. Hence, the pattern consists of one object playing the role of the mediator of a collaboration among objects. The participants of this pattern are: **Mediator** and **Colleague**. Mediator will mediate the interaction between Colleagues. Figure 1.1 shows the class diagram of the pattern.

Note that the pattern itself does not define the protocol of their participants, but for the purposes of this work it has been set up as shown in figure 1.1.

The implementation of this pattern often has an object elected to play the role of Mediator which has references to all the Colleagues that it mediates and Colleagues have reference to their mediator. Thus, whenever a Colleague has its state changed, it calls its Mediator which issues suitable notifications to the other Colleagues.

In this work this pattern will be implemented as a generic component. With this generic component implemented, it will be applied to a simple example in order to highlight the main features of the aspect interfaces and other involved concepts.

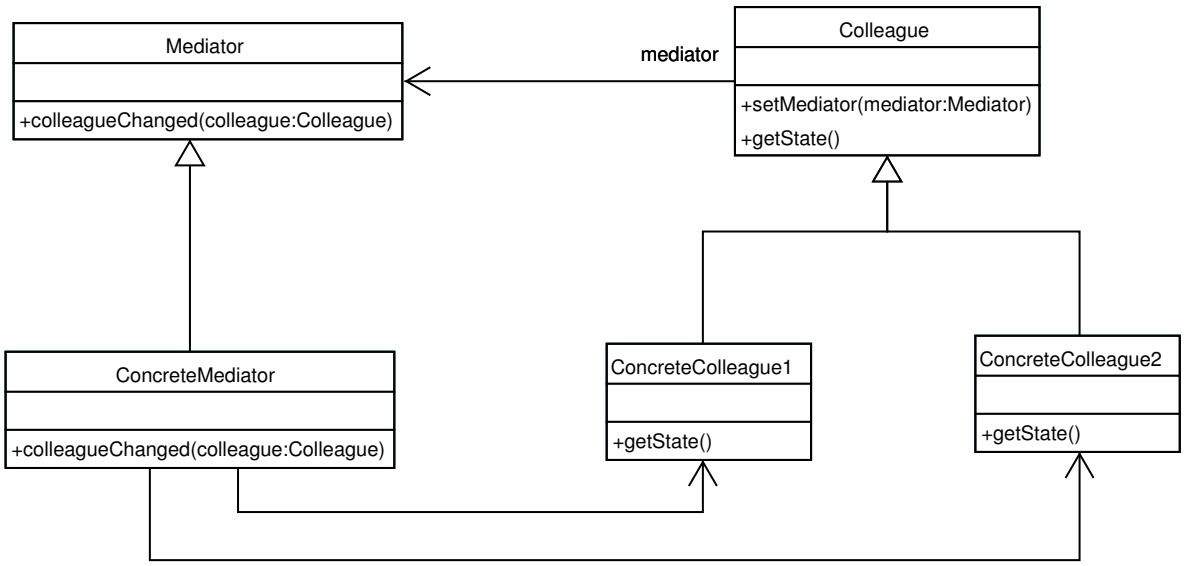

Figure 1.1: Class diagram of the Mediator Pattern

1.2 Problem statement

This section outlines the shortcomings of the current language technology to support reuse-based development in the context of generic components for a-posteriori integration in existing systems. In order to illustrate the problems described in this section an implementation as proposed in [5] is shown. This implementation uses the AspectJ [14] model to describe a reusable implementation of the Mediator pattern as presented in the previous section.

Before showing the approaches used for the AspectJ solution and its related problems, an important concept of AspectJ, which is used in the example, must be clarified: the *introduction mechanism*. This mechanism allows changing the structure of Java classes without having access to their sources. This is obtained through the declaration of introductions in the aspect definition. For example, in listing 1.2 the clause `declare parents:` inserts a subtype relationship between `Button` and `Colleague` turning `Button` into a subtype of `Colleague`. Even though the solution presented does not need to introduce methods or fields into the base classes (`Button` and `Label`), it would be possible. AspectJ allows declaring for example a method out of the class scope. A method is defined in such a way when its name contains also the name of the class it belongs to. For example, the definition of the method `void Button.doubleClick(){...}` would cause the addition of this method to the class `Button`. Similarly, `private LinkedList Label.colleagues` would add the field `colleagues` into the class `Label`.

Now that this concept has been clarified, the example can be explained. Listing 1.1 shows the implementation of the reusable protocol of the pattern. `MediatorProtocol` is an *abstract aspect* which defines two interfaces: `Colleague` and `Mediator`. These interfaces are the abstraction of the roles of this collaboration.

The *abstract pointcut* `change(Colleague c)`, when implemented, is responsible to activate the *advice* `after(Colleague c): change(c)`. Hence, this advice will run after all methods to which the pointcut `change(Colleague c)` gets bound. The execution of such advice causes the notification of the mediator, which is set to this colleague, calling the abstract method `notifyMediator()`. The mapping between the mediators and colleagues is stored in the `WeakHashMap` called `mappingColleagueToMediator`.

```
public abstract aspect MediatorProtocol {
  protected interface Colleague  { }
  protected interface Mediator { }
  private WeakHashMap mappingColleagueToMediator = new
     WeakHashMap();

  private Mediator getMediator(Colleague colleague) {
    Mediator mediator =
    (Mediator) mappingColleagueToMediator.get(colleague);
    return mediator;
  }
  public void  setMediator(Colleague c, Mediator m) {
    mappingColleagueToMediator.put(c, m);
  }

  protected abstract pointcut change(Colleague c);

  after(Colleague c): change(c) {
    notifyMediator(c, getMediator(c));
  }
  protected abstract void notifyMediator(Colleague c,
     Mediator m);
}
```

Listing 1.1: Reusable mediator protocol in AspectJ

In listing 1.2 a possible binding of the `MediatorProtocol` is shown. Using AspectJ's introduction mechanism, the subtype relationship is created between `Colleague` and `Button` as well as `Mediator` with `Label`. It means that for example `Button` plays the role of `Colleague` in this collaboration. The realization of the pointcut `change(Colleague c)` indicates that the mediators are notified always after the method `clicked()` is executed. This binding also implements the abstract method `notifyMediator(Colleague`

`c, Mediator m)` inserting the proper notification logic.

```
public aspect MediatorImplementation extends
   MediatorProtocol {

  declare parents: Button implements Colleague;
  declare parents: Label implements Mediator;

  protected pointcut change(Colleague c):
    (call(void Button.clicked()) && target(c));

  protected void notifyMediator(Colleague c, Mediator m) {
    Button button = (Button) c;
    Label  label  = (Label)  m;
    if (button == Main.button1) {
      label.setText("Button1 clicked");
    } else if (button == Main.button2) {
      label.setText("Button2 clicked");
    }
    button.setText("(Done)");
  }
}
```

Listing 1.2: Binding of the Mediator protocol in the base code

This code deals well with the requirement of describing a reusable component. First of all, the implementation of the mediator protocol is totally decoupled from the binding. So, this same implementation can be easily bound in other contexts. This is because the expected contract, the method `notifyMediator(Colleague c, Mediator m)` and the pointcut `change(Colleague c)` in this code, is separated from the component implementation. Furthermore, one simple role can be freely mapped to as much classes as it is desired, as well as many roles can be bound to the same class.

Note that the introduction mechanism could be used to inject the expected methods and fields into the base code instead of using the approach presented. Thus the aspect participants could define a more refined protocol expressing the expected contract. The problem with this approach is that they would loose flexibility. For example, in a certain application two aspects are defined. Each aspect has one role which expects the implementation of a method `m()`. When these aspects are bound to the base code, it is detected that a certain class must bind both roles. This code would definitely generate a compile error since the class would have the method `m()` defined twice. Similarly it would occur if a certain class has to be bound with the same role many times. Therefore, the code presented is more flexible concerning extendibility and reusability.

However, even though this solution achieves extendibility and reusability, it has yet some problems. The first problem becomes clear just by taking a quick look at the code. As can be seen all methods in this collaboration are defined in the aspect `MediatorProtocol` instead of in the abstraction they conceptually belong to. It breaks hence the separation of concerns within the aspect leading to procedural code and loosing therefore modularity.

Lacking modularity implies losing object-oriented features to describe the aspect. For example, passing the role abstraction to another class that does not collaborate with the whole aspect would not be possible since the methods do not pertain to the role. Similarly, more complicated interactions between roles of the same aspect would increase much complexibility of the aspect because the outer aspect would have to manage itself all these interactions. It could get even worse if object-oriented features such as inheritance between roles are needed because it would have to be manually controlled in the outer aspect. Again the introduction mechanism could be used to insert the methods and fields into concrete classes in the base code allowing the definition of methods in the roles instead of in the outer aspect, but as discussed above it is not desirable as well.

All problems described above come because the implementation of the component needs to be decoupled from a particular application which it might get integrated. But as the language offers only little support to express the expected contract of the component, many object-oriented features are lost.

Another problem concerning this solution is that the aspect binding is coupled with a particular implementation of the aspect. It is clear that the implementation of the components should not be coupled with a particular binding since components represent generic concepts. Nonetheless, clients tightly coupled with component implementations are also undesirable. Bindings translate the application specific concepts, terms and abstractions to the component's world, but they do not directly depend on a specific component implementation. Therefore, the coupling of the binding with a particular component implementation implies that they cannot be reused to bind the application with different component implementations.

The solution of creating a subtype relation between roles and base classes is also questionable. Inserting this relation has an invasive nature since it changes physically the type structure of the base application, even though physical changes in the source code are not needed. Beyond conceptually wrong since the base object is not of the role type, but just plays that role in certain collaboration, it can cause practical problems. Objects of those changed classes will remain instances of the role type, even when the aspect collaboration has finished, or it has not started yet. It is not an issue if the component is applied only once in the application, but in systems that reuse this pattern many times, special care must be taken by the programmers.

For example, using listing 1.2, an instance of a class other than `Button` but which has been mapped to play the role of `Colleague` in other implementation of `Mediator-Protocol` might be passed as `Colleague` to the method `notifyMediator(Colleague, Mediator)` defined in the `MediatorImplementation` class. It would raise a `ClassCastException` since in the implementation of this method it casts the `Colleague` received to the `Button` class and the reference passed is not an instance of this class.

AspectJ from invasiveness is not very expressive for mapping the aspect abstraction to corresponding base classes. The aspect abstractions in AspectJ can be mapped to the base classes only through the `declare parents:` clause, as shown in example 1.2. So, the base structure must have a specific abstraction to be bound to the aspect role. However, this is not always the case; more sophisticated mappings may be needed. For example, a role of some aspect could be needed to be bound with a concept in the software that is not an abstraction itself, but a collaboration between abstractions instead, for example.

1.3 Releated work

This section presents other efforts that have been done in order to improve the current language technologies in the context of the problems that have been described in the last section. Two works[2] are shown here: Object Teams [15] and ACC [7][13]. These two models have their prototype implementations available as extensions of the Java compiler such as the implementation presented in this thesis.

1.3.1 Object Teams

Object Teams enable an abstraction to define generic components called Team Class. The Teams combine properties of the classes with properties of packages. In Teams, nested classes may be defined, these classes are called Role Classes. The Teams look like ordinary Java outer/inner classes, just look like. The differences become clear when regarding closer the inheritance relationship. The inheritance between teams implies an implicit inheritance relationship of the contained Roles. Therefore, when a subtype of a Team is created, all the roles defined in the super Team become available in the subtype and can be overridden. In order to do so, the sub-Team has just to declare a Role Class with the same name of the super Role it wants to override. It creates the implicit relationship between them.

[2]The features of these works presented here are just those relevant to this thesis, other possible features that are not described here are out of the scope of this work

For example[3], listing 1.3 describes inheritance relationships between Teams. In the Team `MySubTeam`, the Role `Role1` is overridden. It implies that this new `Role1` is subtype of that `Role1` defined in `MyTeam`. Hence methods and attributes can be overridden following the known rules.

```
public team class MyTeam {
  class Role1 {
    String name;
    public Role1 (String n) { name = n; }
    public void print() {
      System.out.println("Name=" + name);
    }
  }
  public Role1 getRole()  { return new Role1("Joe"); }
}
public team class MySubTeam extends MyTeam {
  class Role1 {
    int age;
    public void setAge(int a) { age = a; }
    public void print() {
      tsuper.print();
      System.out.println("age=" + age);
    }
  }
  public void doit() {
    Role1 r = getRole();
    r.setAge(27);
    r.print();
  }
}
...
MySubTeam myTeam = new MySubTeam();
myTeam.doit();
...
```

Listing 1.3: Implicit inheritance of Role Classes

Another important features of Object Teams for the purposes of this work are the definition of the binding abstractions and the expect contract. Binding abstractions here are Roles that map a Role to base classes. These mappings are performed using the key word `playedBy` as an extends clause in Java. The semantics of this language construction is that the Roles marked with this key word will *decorate* the base class

[3]All examples used to explain Object Teams are from [15]

defined in the clause.

The expected contract can be defined by means of *callin* or *callout* methods. Callout methods are defined as abstract methods in the super Roles, and they are bound to the base methods through simple method forwarding and parameter mappings. On the other hand, callin methods are roughly equivalent to advice weaving in existing aspect-oriented languages. As well as at the callout methods, callin methods can be mapped explicitly to base methods. However here three modifiers `before`, `after` and `replace` control the relation between the role method and the base method which it is bound. Callin methods may be defined in the Roles with a new modifier `callin`. Thus methods marked with this modifier shall be bound to a base method, and the body of such callin method should invoke "itself" by using the special name *base*. It causes the invocation of the original method (the method in the base class that it has been bound).

In the code shown in listing 1.4, `Role1` in `MySubTeam` binds itself to the base code `Staff`, mapping for example the callout method `payEuro(float)` to the base method `payDM(float)`, changing the value of the parameter `dm`.

```
public team class MyTeam {
  class abstract Role1 {
    abstract void payEuro(float euro);
    ...
  }
...
}

public class Staff { // a base class
  public void payDM (float dm) { ... };
  ...
}

public team class MySubTeam extends MyTeam {
  class Role1 playedBy Staff {
    void payEuro(float euro) -> payDM(float dm) with {
      euro * 1.95338 -> dm
    }
    ...
  }
  ...
}
```

Listing 1.4: A binding with callout methods

Listing 1.5 shows the callin method `log(String)` that is bound to the method `lo-`

`gin(String, String)` from the base class. Note that the methods do not have the same signature, it is hidden from the method implementation in the component but the second parameter is passed when it calls the base method.

```
...
class LogLogin playedBy Database {
   callin void log (String what) {
      System.out.println("enter:" + what);
      base.log(what.toLowerCase());
      System.out.println("leave:" + what);
   }
   void log(String what) <- void login(String uid, String
      passwd) with {
       what <- uid;
   }
}
...
(new Database()).login("Admin", "Passwd");
...
```

Listing 1.5: The callin method implementation

The Object Teams approach presents good support to describe the concern using several related abstraction with the use of the Role Classes and their implicit inheritance relationship that is one of the problems described in 1.2. Also it is not design invasive when binding the component to the base code. It does not change the base class but instead it just wraps the base class forwarding the methods as described by the developer.

Though Object Teams can express the separation of the expected contract implementation from the provided contract, this separation does not decouple the component implementation from the binding. In other words, there is no support for using the same binding with different implementations of the same component.

Another problem of this model is that it uses the same approach from AspectJ in order to bind the component to the base code. The mappings of the components to the base code are performed using the clause `playedBy`. This clause needs an abstraction from the source code to be bound to the component, but as pointed out before it is not very expressive.

1.3.2 ACC

The approach used in ACC is combine module systems with aspect-oriented technologies. The main abstraction in this model is the *Aspectual Collaboration*. This abstrac-

tion can be used to describe components. This language construction is like a package as it is known from Java, but it has the ability of be *attached* to other abstractions. This concept will be clarified soon.

An Aspectual Collaboration consists of defining components through the definition of their *participants*. The participants are similar to ordinary Java classes, but they can express the expected contract of the participant. In order to express the expected contract, participants can define two kinds of methods: *expected* and *aspectual*. The expected methods look like ordinary abstract methods in Java, but they do not statically inhibit instantiation of the participant. On other hand, aspectual methods can be seen as the aspect-oriented approach of the model. These methods intercept the execution of the methods which they are bound. In the prototype only *around advices* can be expressed.

Listing[4] 1.6 shows the Aspectual Collaboration implementing the Observer design pattern [3]. `ObserverPattern` is the Aspectual Collaboration and defines two participants `Watched` and `Observer`. The participant `Watched` provides an implementation of `getObservers()` as a normal Java class, but this participant also defines the expected method `notify(Observer)` and the aspectual method `watchedOp(JP)`. Note that the return type and parameter of this last method are undefined types; they will be defined only when it is mapped to a concrete method.

```
collab ObserverPattern;
  participant Watched {
    aspectual RV watchedOp(JP jp) {{
      Observer [] obs = getObservers();
      for (int i = 0; i < obs.length; i++) notify(obs[i]);
      return jp.invoke();
    }}
     expected void notify(Observer o);
     Observer [] getObservers() {{ return observers; }}
  }
  participant Observer {
    Watched getWatched() {{ ... }}
  }
```

Listing 1.6: Declaration of an Aspectual Collaboration

In order to bind the Aspectual Collaborations with the base code, they must be attached. When attaching Aspectual Collaborations, their participants are bound to the base abstractions that will play their roles during the collaboration. Expected methods may be provided and the aspectual methods may be mapped at this stage.

[4]This example is shown in [13]

When it is bound, as shown in listing 1.7, it maps its participants to the base classes, in the example `TextEditor` will play the role of `Watched` in this collaboration providing an implementation of the expected method `notify(Observer)` and mapping the method `edit` to be intercepted by the aspectual method `watchedOp`.

```
collab ObservedBase;
extends Base;
  attach ObserverPattern {
    TextEditor += Watched {
      around edit do watchedOp;
      provide notify with
        void ntfy (Observer obs) {{
          TextLogger tl = (TextLogger)obs;
          tl.logText(this.getText());
        }}
    }
    TextLogger += Observer { }
  }
  attach ObserverPattern {
    File += Watched {
      around write do watchedOp;
      provide notify with
        void ntfy (Observer obs) {{
          FileLogger fl = (FileLogger)obs;
          f.logFile(this.getBytes());
        }}
    }
    FileLogger += Observer { }
  }
```

Listing 1.7: Two bindings of the ObserverPattern

This approach deals well with the problem of expressing components as a collaboration of related abstractions with the definition of the Aspectual Collaboration and its participants. It also has support to separate the expected contract from the implementation of the component, but as it was detected in Object Teams, it does not decouple the implementation of the component from its bindings. The bindings are directly related with a specific implementation of the component. On the other hand, the bindings are not design invasive as AspectJ. They do not add the component types as super types of the base abstractions. Again here, when binding the component to the base program, an abstraction must exist in the base code to play the role of a participant of the collaboration. However, this problem is better treated in ACC than in Object Teams, because it can join more than one abstraction when binding a component.

Chapter 2

The Caesar Model

This chapter presents the Caesar model. Many features of the model are already implemented. These implementations were realized in the context of two other Master theses from Andreas Wittmann and Jürgen Hallpap. This chapter gives an overview of these works. Obviously this chapter does not describes all the mechanisms in details, for more details see [16] and [4]. Showing these mechanisms becomes necessary because they are the basis for implementing Aspect Interfaces and their mechanisms in this thesis.

Caesar is a new programming language which introduces support to facilitate the separation of concerns of the applications into reusable software blocks. The purpose of the language is to provide support for allowing the developers to face with the problems described in 1.2, and for giving more flexibility to the join point interception model of current aspect-oriented language such as AspectJ.

With Caesar, crosscuting concerns become modular, thereby allowing them to be grouped into components. On the other hand, it facilitates the integration of such components with existing software in a no invasive fashion. The implementation of Caesar is an extension of the Java programming language. All functionalities of that language are also available in Caesar, and all bytecode produced by the Caesar compiler is Java Virtual Machine compatible.

Although aspect interfaces are part of the Caesar model, they are not presented in this chapter because they are explained in details in chapters 3 and 4. In the next sections the Caesar mechanisms are presented. Some mechanisms such as *Virtual types* and *Delegation* are explained in more details because they are extensively used in this thesis.

2.1 Family polymorphism

Andreas Wittmann, in his thesis, developed a compiler called *FamilyJ*. This compiler introduces the concepts of *Family Polymorphism* in the Java programming language and is the base for other concepts of Caesar. This section gives an overview of this concept and shows the approach used in FamilyJ to implement it on top of Java language.

Family polymorphism deals with grouping of types for allowing reuse and safety in collaborations of group of objects. In [2] it is defined as "a programming language feature that allows us to express and manage multi-object relations, thus ensuring both the flexibility of using any of an unbounded number of families, and the safety guarantee that families will not be mixed. This kind of polymorphism gives more power to the compiler in order to ensure statically that the objects in a particular collaboration are all pertaining to the same *family* of objects.

FamilyJ implements family polymorphism using the notions of *virtual types* and *delegation*. The next subsections discuss the implementation of these two concepts and the family polymorphism implementation itself in Caesar.

2.1.1 Virtual types

Nested classes allow defining classes that are in someway contained in an object. These classes provide interesting features to implement family polymorphism. However, they do not have themselves enough expressiveness to do so.

Family polymorphism requires the types of enclosing objects to be *late bound*. In a nutshell, for implementing the grouping of types purposed for such technique, nested types should be *virtual* similar to what is known as *virtual methods*. In a subclass relationship, virtual methods may be overridden by subclasses when they define a method with the same signature as the virtual method in the super class. During compile time, if a reference of the type containing the virtual method invokes the method, it will be late bound, allowing the execution of the most recent implementation of the method in the class hierarchy.

Virtual types give means to define nested classes that may be overridden by sub classes of the enclosing classes. For example, if a class C defines a nested class N, and its subtype C' has also N declared, it will cause an implicit inheritance relationship between $C.N$ and $C'.N$. And the instantiation of N will be late bound similar as the execution of virtual methods.

FamilyJ extends the semantics of Java nested types introducing two keywords to express virtual types. These keywords are the two class modifiers: `virtual` and `override`.

Listing 2.1 shows how the virtual types are defined and overridden in FamilyJ.

```
public class MediatorProtocol {
  public virtual class Mediator {
    public void colleagueChanged(Colleague c) { ... }
  }

  public virtual class Colleague {
    public void setMediator(Mediator m) { ... }
    public Object getState() { ... }
  }
}

public class SubMediatorProtocol
  extends MediatorProtocol {

  public override class Mediator {
    public void colleagueChanged(Colleague c) { ... }
  }
}
...
MediatorProtocol protocol = new SubMediatorProtocol();
MediatorProtocol.Mediator mediator = protocol.Mediator();
...
```

Listing 2.1: Virtual types in FamilyJ

The classes `Mediator` and `Colleague`, in example 2.1, are defined as virtual in `MediatorProtocol` using the new class modifier `virtual`. Therefore, they can be overridden in sub classes of `MediatorProtocol`. It is what happens in `SubMediatorProtocol`. In this class, `Mediator` is overridden because it is defined with the same name as defined in the super class, and it uses the new class modifier `override`. Note that, now `SubMediatorProtocol.Mediator` inherits from `MediatorProtocol.Mediator`, hence methods and fields may be overridden following the rules known from ordinary Java classes.

It may seem just syntactic sugar for developers do not have to explicitly create the inheritance relationship. Late bound of nested types is not yet achieved. However, for each virtual class that is defined, the compiler introduces a factory method into the enclosing class. Hence, it uses the late binding available for methods to introduce late bound instantiation of virtual classes. The code in listing 2.2 shows `MediatorProtocol` and `SubMediatorProtocol` after these transformations.

Now in the transformed classes shown in listing 2.2, `Mediator` is late bound; the object created will be of the type `SubMediatorProtocol.Mediator`. Obviously, developers

do not have to use the factory method instead of `new` operator to create their objects. The compiler automatically replaces the instantiation of virtual types by factory method calls. Note that the factory methods return `Object`, it is because the return type cannot be changed when overriding a method in Java. The solution was to introduce casts to the right type when calling the factory methods as shown in listing 2.2. Again the casts are automatically introduced by the compiler.

```
public class MediatorProtocol {
  public class Mediator {
    public void colleagueChanged(Colleague c) { ... }
  }

  public class Colleague {
    public void setMediator(Mediator m) { ... }
    public Object getState() { ... }
  }

  public Object createMediator() {
    return new Mediator();
  }
  public Object createColleague() {
    return new Colleague();
  }
}

public class SubMediatorProtocol
  extends MediatorProtocol {

  public class Mediator
    extends MediatorProtocol.Mediator {
    public void colleagueChanged(Colleague c) { ... }
  }

  public Object createMediator() {
    return new Mediator();
  }
}
...
MediatorProtocol protocol = new SubMediatorProtocol();
MediatorProtocol.Mediator mediator =
  (MediatorProtocol.Mediator) protocol.createMediator();
```

Listing 2.2: Transformed virtual types

2.1.2 The delegation mechanism

The concept of virtual type brings a great flexibility for describing group of collaborating classes. For example, the code in listing 2.3 inserts the virtual class `SpecialMediator` in `MediatorProtocol`. As this class is a specialization of `Mediator`, and keeping in mind the late bound instantiation of the virtual types discussed before, the super class of `MediatorProtocol.SpecialMediator` in the instantiation shown in the example is conceptually `SubMediatorProtocol.Mediator` instead of `MediatorProtocol.Mediator`. But it cannot be achieved with the virtual types as they were described above. Therefore, to allow late binding in such cases, *delegation* is used instead of inheritance to allow virtual classes to "inherit" from classes that are not known at compile time.

Delegation gives the flexibility needed to implement virtual types. It is because delegation deals with inheritance through collaboration of objects instead of statically defined class relationships. The delegation used in FamilyJ is static, meaning that parents and children cannot be changed at run-time, because the static approach is sufficient to deal with the requirements of virtual types and dynamic delegation would lead to a more complex implementation.

FamilyJ's compiler performs some transformations in the base code in order to implement delegation. In the following, these transformations are discussed:

In FamilyJ all virtual types are transformed to have a reference to their *parents*. Hence, all super class methods that are called for instances of subclasses of a class are delegated to their parent object. For example, be `C` a virtual class and `C'` another virtual class that is subclass of `C`. After some transformations, the class `C'` has a reference called `parent` of the type `C`. Thus, if `C` defines a method `m()` which is not overridden in `C'`, all `m()` calls for instances of `C'` are delegated to the object `parent`.

Beyond adding the parent reference, the subclass relationship is replaced by a subtype relation. It is required because subclasses may not know the real implementation of their parent at compile time, what they know is the type of their parents. Therefore, it implies that the classes are divided from their types. In terms of Java, it means that class is now divided into a class which contains the implementation and an interface which defines the type of the object. Hence, instead of subclasses referring to their super class, they implement their super class interface.

Figure 2.1 shows these two transformations. On the left side, the classes before the transformations are shown. This class hierarchy is transformed as shown on the right side of the figure. The class `Mediator` is broken up into the interface `Mediator` and the class `Mediator_Impl`, and a subtype relation is created between them. Now the implementation is not part of the type `Mediator` anymore. Similar to `Mediator`, `Spe-`

cialMediator is also divided into an interface and an implementation class, replacing therefore the subclass relation between Mediator and SpecialMediator by the inheritance among their interfaces.

```
public class MediatorProtocol {
  public virtual class Mediator {
    public void colleagueChanged(Colleague c) { ... }
  }

  public virtual class Colleague {
    public void setMediator(Mediator m) { ... }
    public Object getState() { ... }
  }

  public virtual class SpecialMediator
    extends Mediator {
    ...
    public void addColleague(Colleague c} { ... }
  }
}

public class SubMediatorProtocol
  extends MediatorProtocol {

  public override class Mediator {
    public void colleagueChanged(Colleague c) { ... }
  }
}
...
MediatorProtocol protocol = new SubMediatorProtocol();
MediatorProtocol.SpecialMediator mediator =
  protocol.SpecialMediator();
...
```

Listing 2.3: A more complex virtual type definition

In the transformed code of figure 2.1, one can also see the reference in subclasses to their parent. The class SpecialMediator_Impl has a reference of Mediator which is used now to delegate method calls. Therefore the method calls for methods that are implemented in Mediator and are not overridden in SpecialMediator in the original code will be forwarded to the right implementation through the reference parent by instances of SpecialMediator_Impl.

These transformations enable the structure for delegation, but delegation is not achieved

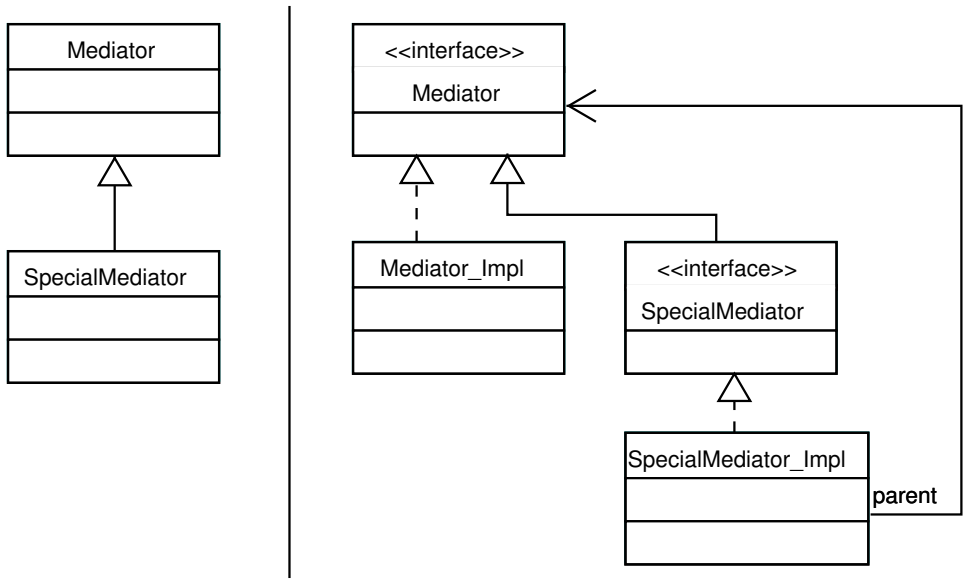

Figure 2.1: Class transformations for delegation

yet only with this structure. With this structure it is possible to forward methods to the parent reference, but alone it does not allow methods to be dispatched to the right objects, using the terms introduced before methods are no longer late bound anymore.

For example, in listing 2.4 if an instance of `SpecialMediator` receives a call for `colleagueChanged(Colleague c)`, it will forward the call to its parent. The implementation of the method `colleagueChanged(Colleague c)` in the parent calls the method `getColleagues(Colleague c)`. This call will be dispatched to the wrong implementation since the implementation reached will be that in the class `Mediator` instead of the implementation in the class `SpecialMediator` that is the type of the original receiver of the method call. Therefore FamilyJ does more transformations in the base code to achieve delegation. Now the transformations are performed for methods.

The approach used in FamilyJ to avoid this problem is adding a new method for each method defined in the virtual class. These new methods contain all method parameters plus a parameter that is the reference to the first receiver of the method call. This new parameter, called `self` in FamilyJ, plays the same role as `this` in Java. Therefore, the methods are now dispatched by the subtypes to this new implementation of the method, passing itself to be bound with `self`.

Using only this new method to avoid the dispatching problem would cause copies of the same method implementation in the classes increasing the complexity of the transformations. For this reason, FamilyJ breaks up each method into three methods: the self context method with the parameter `self`, the implementation method also with the

parameter `self` and the method with the signature as it was defined by the developer. Hence, the method implementation is kept in just one method, the implementation method, and the other two methods just forward their execution to such method.

```
public class MediatorProtocol {
  public virtual class Mediator {
    ...
    public void colleagueChanged(Colleague c) {
      ...
      Colleague[] collegues = getColleagues(c);
      ...
    }
    public Colleague[] getColleagues(Colleague c) { ... }
  }
  public virtual class SpecialMediator
     extends Mediator {
     ...
     public Colleague[] getColleagues(Colleague c) { ... }
  }
}
```

Listing 2.4: Wrong method dispatching

Listing 2.5 shows how the code of the method `colleagueChanged(Colleague c)` shown in 2.4 would look like after these transformations[1]. The implementation method `colleagueChanged_implementation(Colleague c)` now encapsulates the implementation, which is the same as it was originally implemented but every reference for this which act as a receiver of a public method is replaced by `self`. The self context method `colleagueChanged_selfContext(Object self, Colleague c)` just forwards the method call to the implementation method passing the parameter `self` as it was received, adding just the cast to the right `self` type. And the method with the original signature forward the execution to the implementation method passing the current method receiver as `self`.

This implementation has yet a problem: self context methods and original methods would have to be copied for all abstractions of the class hierarchy in a complex way, because it would have to look for which methods would have to be delegated to the parent, and which one would be executed by itself. Beyond that, the class hierarchy would have to be recompiled whenever a protocol of some class in the hierarchy changes.

This problem is resolved in FamilyJ creating another abstraction from the virtual classes: the *proxy classes*. For each virtual class therefore is also created a proxy class. These

[1]The class transformations discussed before are not shown because they would pollute the listing with code that is not necessary to understand the method transformations

classes implement the protocol of the virtual classes providing the implementation of these two kinds of methods. The methods defined with the original signature forward now the execution to the self context method, passing the receiver (`this`) as the self parameter, and the self context methods defined in these classes delegate the execution to the parent reference, passing `self` as received.

```
public class MediatorProtocol {
  public virtual class Mediator {
    ...
    //Method with the original signature
    public void colleagueChanged(Colleague c) {
      colleagueChanged_implementation(this, c);
    }
    //Self context method with the self parameter
    public void colleagueChanged_selfContext(Object self,
       Colleague c) {
      colleagueChanged_implementation((Mediator)self, c);
    }
    //Original implementation of the method, but replacing
        this by self.
    private final void colleagueChanged_implementation(
       Mediator self, Colleague c) {
      ...
      //Calls getColleagues(c) in self instead of this
      Colleague[] collegues = self.getColleagues(c);
      ...
    }
    ...
  }
  ...
}
```

Listing 2.5: After the method transformations

Hence FamilyJ performs this new class transformation that implies in another relationship. The virtual classes now have an inheritance relationship with the proxy class of their super virtual class. Figure 2.2 shows in the left side the original code, and the right side illustrates the original code after all the transformations discussed, the method `m()` was used instead of `colleagueChanged(Colleague c)`, as it has been used, for reasons of space.

For the classes taking advantage of the delegation mechanism some restrictions[2] have

[2]Actually, the restrictions are not necessary but it was chosen in [16] to simplify the process.

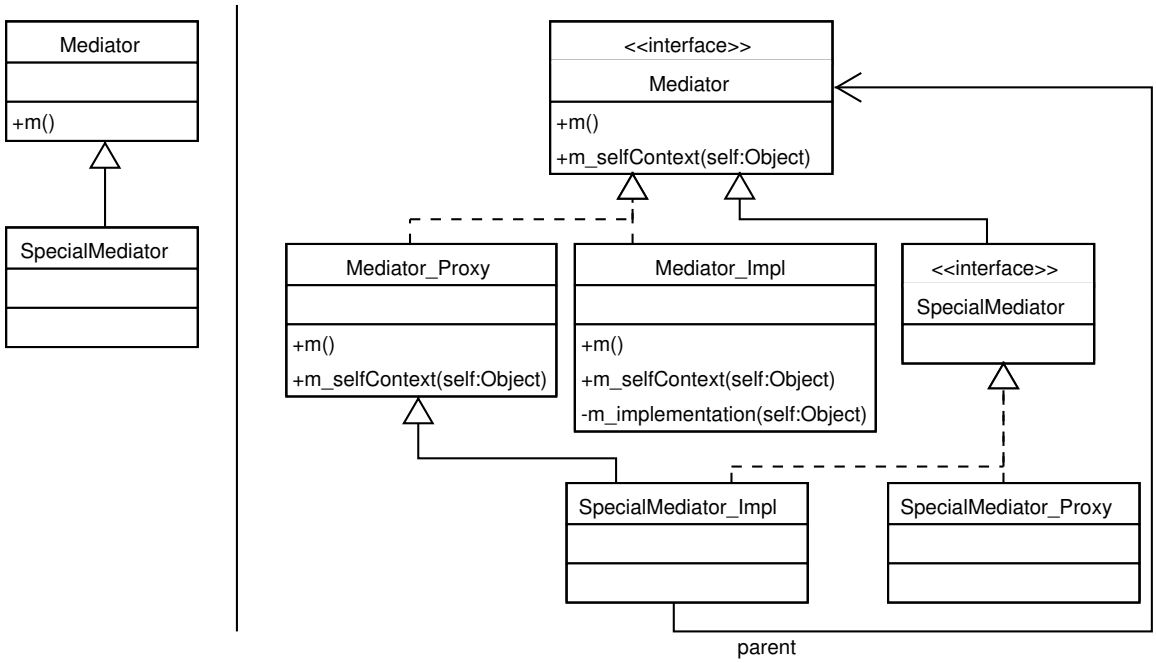

Figure 2.2: Code structure after the transformations

to be respected. The classes that use the delegation mechanism must have a *clean interface*. Clean classes, or classes which have a clean interface, are another type of class in FamilyJ. The clean classes are defined using a new class modifier: `clean`, and all transformations discussed in this section to enable delegation for virtual classes are performed also for this kind of class. Therefore, not only virtual and override classes are able to use the advantages of the delegation mechanism. Any class which respects the restrictions of clean classes and is defined as clean (marked with the `clean` modifier) uses delegation instead of ordinary inheritance. Since virtual and override classes use delegation, it implies that they must be clean as well, therefore they must respect the restrictions.

A class is declared to have a clean interface if it respects the following restrictions:

- it does not declare fields with visibility other than private;
- it does not declare methods with visibility other than public or private;
- it references private members only on this and never on different instances of same class;
- it inherits another clean class or no other class;

- it is super class of another clean class or no other class.

Beyond that, for every constructor defined in clean classes, implicitly or explicitly, a new constructor which receives the parent reference of the object is added. This new constructor has the implementation of the original constructor, and the original constructor now does not have implementation anymore, but just a call to this new constructor, passing as parent a reference for a fresh object of the class marked to be super class of the current class by the developer. This is to allow the `parent` reference initialization; since this reference is immutable during the execution, it has to be initiated by the constructors. For example, listing 2.7[3] shows the clean class `MediatorProtocol` defined in 2.6 after the constructor transformation.

```
public clean class MediatorProtocol {
  ...
  public MediatorProtocol(String name) {
    this.name = name;
  }
  ...
}
public clean class SubMediatorProtocol
  extends MediatorProtocol {
  ...
  public MediatorProtocol(String name) {
    super(name);
  }
  ...
}
```

Listing 2.6: MediatorPrototocol as a clean class

Although the implementation of delegation as presented so far deals well with many cases, it fails yet in more complex cases. The delegation implementation described in this section is flexible enough to allow that subclasses of some clean class act as parent of each other. For example, be *C* a clean class, and *C'* and *C"* clean subclasses of *C*. *C"* can act as parent of *C*. Therefore, when an instance of *C' c'* delegates a method to its parent, it would pass itself as *self* parameter, but even that a reference of *C"* is set as parent of *c'*, *c'* is not of the type *C"*, hence methods defined in *C"* would not be able to call methods that are not defined in *C* but only in *C"*.

Thus, in order to avoid this problem, `self` is calculated at runtime. FamilyJ has a mechanism which creates a *method dispatcher* at run time. This dispatcher is a reference

[3]Again here, this is a pseudo code, the code generated by the class and method transformations were omitted

of a class created on-the-fly, which is of the type of the parent reference. This object has ability to dispatch methods that are defined only in the parent to the parent reference, and others to the original receiver of the method call. Therefore, in a super call, explicit or implicit, the self parameter passed is calculated at runtime.

```
public clean class MediatorProtocol {
  ...
  public MediatorProtocol(String name) {
    this(new Object(), name);
  }
  public MediatorProtocol(Object parent, String name) {
    ...
    this.name = name;
  }
  ...
}
public clean class SubMediatorProtocol
  extends MediatorProtocol_Proxy {
  ...
  public SubMediatorProtocol(MediatorProtocol parent,
     String name) {
    super(parent);
  }
  public SubMediatorProtocol(String name) {
    this(new MediatorProtocol(name));
  }
  ...
}
```

Listing 2.7: Constructor transformation of clean classes

2.1.3 Family polymorphism implementation

All mechanisms described in last sections provide the base for the implementation of family polymorphism in Caesar. Classes which enclose virtual classes then can serve as group of classes. Family polymorphism allows checking at compile time if the families, or group of objects, are not mixed.

FamilyJ allows references of enclosing classes to be part of the reference type of their virtual class instances. For example, listing 2.8 contains the definition of two possible family types: `MediatorProtocol` and `SubMediatorProtocol`. They are possible family types because they enclose virtual classes. Therefore, their instances can be part of the

reference type of instances of their virtual class.

```
public class MediatorProtocol {

  public virtual class Mediator {
    public void colleagueChanged(Colleague c) { ... }
    ...
  }
  public virtual class Colleague {
    ...
  }
}
public class SubMediatorProtocol
  extends MediatorProtocol {
  ....
  public override class Colleague {
    ...
  }
}
...
//Family definition
final MediatorProtocol family = new SubMediatorProtocol();
//Another family
final MediatorProtocol anotherFamily = new
   MediatorProtocol();
//A type definition using family
family.Mediator mediator = family.new Mediator();
//Now using family2
anotherFamily.Colleague colleague =
  anotherFamily.new Colleague();
//It is not allowed by the compiler
anotherFamily.Colleague anotherColleague =
  family.new Colleague();

//It is ok!
mediator.colleagueChanged(family.new Colleague());
//It generates a compile time error,
//since colleague belongs to another family
mediator.colleagueChanged(colleague);
...
```

Listing 2.8: Families in FamilyJ

For example, in the bottom of listing 2.8, two families are created: `family` and `anotherFamily`. Thus these families now are able to belong to the reference type of `Mediator` and `Colleague` instances. It is what happens following the code, for example, `family` is used to be part of the type of mediator.

Defining these families hence allows the compiler to check if the code is not mixing families. For example, the compiler does not allow `anotherColleague` to be instantiated. It would mix the families, because the type of this reference defines that it belongs to `anotherFamily`, and the developer is trying to assign a reference of the family `family` instead. Note that families can be polymorphic, the type of `family` is `MediatorProtocol` in the code, but an object of the type `SubMediatorProtocol` is assigned to it.

Defining reference types using families is available when declaring a reference type of a virtual class in any part of the code. For example, it could be defined as a parameter type, a local variable or a field in this fashion. It would just have to respect the same rules, therefore the compiler can check if the code respects the families in all these places as well.

2.2 Dynamic join point interception

This section gives a brief overview of the concept of dynamic join point interception applied in Caesar. This concept was introduced in the language by Jürgen Hallpap in his Master thesis [4]. His implementation provides language means to define when an *aspect* is applied. The term aspect used in this section has the same meaning as in AspectJ, since the implementation of the join point interception (JPI) in Caesar is an extension of AspectJ's model on top of FamilyJ.

Hence Caesar has an enhanced JPI model. Beyond all features provided by the JPI model of AspectJ (such as joinpoint and advices definition), Caesar gives language support to allow aspects to be deployed dynamically. Besides, Caesar does not need a special crosscut abstraction, like `aspect` in AspectJ, to define joinpoints and advices. In Caesar they can be declared in ordinary classes.

For an aspect to become active in Caesar, it must be deployed through the language construction `deploy(aspect)`. After this statement, a block is defined, and only the code surrounded by this block will be affected by the aspect. Listing 2.9 shows how the aspects can be deployed in Caesar. In the bottom of this listing the aspect that will be used is chosen and deployed. So, in this code the first time the method `colleagueChanged(Colleague)` is called, it is affected by the aspect chosen while the second one is not.

```
public class ChangeAspect {
...
   pointcut change(Mediator m, Colleague c):
    call (* colleagueChanged(..) ) && target(m);
}

public class LoggingChangeAspect
  extends ChangeAspect {
  ...
  before(Mediator m, Colleague c): change(m, c) {
    out.println(c + `` changed and '' + m + `` called.'');
  }
}

public class CountingChangeAspect
  extends ChangeAspect {
  ...
  before(Mediator m, Colleague c): change(m, c) {
    count ++;
  }
}
...
//Create the aspect to be applied.
ChangeAspect aspect = null;
if (wantsLogging)
  aspect = new LoggingChangeAspect();
else if (wantsCounting)
  aspect = new CountingChangeAspect();

//Deploy the chosen aspect
deploy(aspect) {
  ...
  //If an aspect was chosen, it is aplied here
  mediator.colleagueChanged(colleague);
  ...
}
...
//here there is no aspect deployed anymore
mediator.colleagueChanged(colleague);
...
```

Listing 2.9: Aspect deployment

Listing 2.9 presents also another important feature of the model: the *Aspectual Polymorphism*. The Caesar model gives flexibility enough for aspect types to be chosen dynamically, for example, if the variable `wantsLogging` is `true`, the aspect instantiated is `LoggingChangeAspect`, otherwise if `wantsCounting` is `true`, `CountingChangeAspect` is created.

Note that the aspect to be deployed can even be `null`. The semantics of a `null` aspect is the same as when there is no aspect deployed, in other words, the code inside the deploy block is not affected by any aspect.

Chapter 3

Language support for Aspect Interfaces

As introduced before, this thesis provides the implementation of aspect interfaces and the concepts involved in context of Caesar. This chapter explains in more details these concepts and shows what was included into the language in order to allow developers to express the concepts.

The following sections are divided by the core concepts of aspect interfaces that were implemented during this thesis. Section 3.1 presents the aspect interfaces, followed by sections 3.2 and 3.3 which show how the aspect interfaces are implemented using Caesar. After that, sections 3.4 and 3.5 describe how to use aspect interfaces and their implementations and bindings.

3.1 Aspect Interfaces

Defining generic functionalities is not a simple task. Using current language technologies it gets worse since they do not have expressiveness enough to do so. As pointed out in section 1.2, even though using current aspect-oriented models such as AspectJ brings a great advance in modularizing concerns, the structure of the aspects does not follow the concepts of extendibility and reusability.

The notions of aspect interfaces bring new kinds of abstraction to the language to describe generic functionalities. Aspect interfaces are responsible to define the protocol of such functionalities allowing them to be defined as self-contained components. These interfaces let developers define concerns of applications detached from each other, thereby avoiding the problems of modularity in such abstractions.

Beyond defining the provided behaviour of components as ordinary interfaces do, aspect interfaces have the ability to describe what components expect from their clients, decoupling hence aspect implementations and bindings that can now be defined in distinct modules. Though they have this special ability, they behave as ordinary interfaces. It allows both implementations and bindings to use the whole protocol as their own.

In Caesar, an aspect interface can be defined using the common Java's interface declaration clause and adding an interface modifier: `aspect`. Therefore interfaces marked with this new modifier play the role of describing how the components will communicate with their clients and vice-versa. As common interfaces, aspect interfaces can be used in any place of the code to define reference types.

Since the whole functionality cannot always be described using just one abstraction, but a set of collaborating participants instead, aspect interfaces can contain nested interfaces to describe a set of related abstractions. Any interface declared inside an aspect interface is implicitly an aspect interface, even if it is not marked with the modifier `aspect`.

Nested aspect interfaces become part of the enclosing aspect interface. The semantics of these interfaces can be seen as the semantics of methods defined in interfaces. Their implementation must be supplied by their implementers. Thus as well as implement methods of the interface, bindings and implementations of an enclosing aspect interfaces must provide implementation of the nested interfaces.

In addition to that, nested aspect interfaces can be overridden by subtypes of their enclosing interfaces. Due to FamilyJ's virtual classes, it causes an implicit inheritance relationship between them, and the implementers must provide implementation of the most specific nested aspect interface in the hierarchy. In order to override a nested aspect interface, sub interfaces of the enclosing aspect interface must define a nested interface with the same name, using the FamilyJ's class modifier `override`. Therefore, it follows the same rules discussed for virtual classes in FamilyJ.

Listing 3.1 shows an example of an aspect interface. The interface `MediatorProtocol` is an aspect interface since it is defined with the modifier `aspect`. It defines the nested aspect interface `Mediator`. And this interface is overridden by the sub aspect interface `SubMediatorProtocol` using the interface modifier `override`.

Defining an interface as aspect interface implies that all methods defined on it must be marked with one of the two new method modifiers: `expected` or `provided`. These modifiers hence allow choosing which methods will be implemented by the component and which ones are going to be provided by the clients of such component.

Listing 3.2 shows the aspect interface definition for the Mediator pattern which is used during the rest this chapter. `MediatorProtocol` is an aspect interface which defines

two nested aspect interfaces: `Mediator` and `Colleague`, which are the participants of the pattern. Therefore implementations and bindings of `MediatorProtocol` must provide implementation of `Mediator` and `Colleague`. The method `colleagueChanged( Colleague colleague)` defined by `Mediator` is dependent on the specific context that the aspect interface gets bound to, therefore it was defined *expected*. On the other hand, `setMediator(Mediator mediator)` is generic enough to be implemented by the component, thus it was defined *provided*.

```
public aspect interface MediatorProtocol {

  //it is an aspect interface too
  public interface Mediator {
    ...
  }
 ...
}
public aspect interface SubMediatorProtocol
  extends MediatorProtocol {

  //This interface overrides MediatorProtocol.Mediator
  public override interface Mediator {
    ...
  }
}
```

Listing 3.1: Aspect interface inheritance

```
public aspect interface MediatorProtocol {

  public interface Mediator {
    //this method must be implemented by the client.
    public expected void colleagueChanged(
      Colleague colleague);
  }

  public interface Colleague {
    //this method must be implemented by the component.
    public provided void setMediator(Mediator mediator);
    public provided Mediator getMediator();
    public expected String getState();
  }
}
```

Listing 3.2: Aspect interface of the Mediator pattern

Aspect interfaces describe the type of generic functionalities. As ordinary interfaces, they do not provide any implementation but just describe the protocol. It decouples the functionality abstraction from any specific implementation detail. Besides it separates particular component implementations from their clients allowing them to use any implementation of such component.

3.2 Aspect Implementations

Aspect implementations are the implementation of a generic functionality described by an aspect interface as pluggable component. They realize the aspect interface by providing implementations for the provided contract defined on it. These classes are called *implementation classes* in this work.

The main purpose of implementation classes is to separate the component implementation from the component definition itself, or aspect interface using the terms of this work. It is because for a single component there may be many implementations. With that, binding classes can be decoupled from a specific implementation of the component. Thus, the model does not suffer from the problem described in section 1.2 of AspectJ's solution where binding classes are directly attached with a particular component implementation.

In Caesar, implementation classes are defined using the language construction `provides`. This clause is defined in the same way as Java's `extends` clause but the type defined at the clause must be an aspect interface. Using this clause when defining a class implies a subtype relationship between the class and the aspect interface; in addition to that, it must implement all methods marked with the `provided` modifier in the aspect interface.

Furthermore, implementation classes must implement all nested aspect interfaces defined in the aspect interface that it provides. For doing so, this class has to provide nested classes with the same name of the super nested aspect interfaces, similarly as it is done with the methods defined in the interface. So, nested implementation classes do not need to define the `provides` clause since it is implicitly achieved by matching names. Besides that, they cannot declare other inner classes than those ones defined in the aspect interface.

For example, listing 3.3 shows `MediatorProtocolImpl` which is an implementation of the aspect interface `MediatorProtocol` defined in the last section. In this code, the aspect interface `Colleague` is implemented just because `MediatorProtocolImpl` defines a nested class with the name `Colleague`. Therefore, in spite of the subtype relationship between `MediatorProtocolImpl` and `MediatorProtocol`, the `provides` clause implies that `MediatorProtocolImpl.Colleague` is subtype of `MediatorProto-`

`col. Colleague`. Note that `Mediator` was not implemented in this class; it is because this abstraction does not define any provided method.

```
public class MediatorProtocolImpl
  provides MediatorProtocol {
  //Here above, the class defines that it is
  //the implementation of MediatorProtocol

  //Collegue is implemented here
  public class Colleague {
    private Mediator mediator;
    public void setMediator(Mediator mediator) {
      this.mediator = mediator;
    }
    public Mediator getMediator() {
      return mediator;
    }
  }
}
```

Listing 3.3: Implementation class of the Mediator pattern

Implementation classes can be specialized. Subclasses of implementation classes become implementation classes as well and can override methods as in an ordinary subtype relationship, besides that they can also override the nested implementation classes. Nested implementation classes have the same semantics as FamilyJ's virtual classes. Thus, they can be overridden using the same rules presented in the last section.

However, implementation classes on top of the class hierarchy *cannot* extend other classes. In other words, if a class defines the clause `provides`, it is forbidden to define the clause `extends`. Besides it *cannot* provide implementation of more than one aspect interface as well as it *cannot bind* (this concept will be highlighted soon) an aspect interface; other ordinary interfaces *can* be freely implemented in the same way as common classes do. In a nutshell, the `provides` clause can be seen as an `extends` clause with different semantics.

Another restriction of implementation classes is that they cannot define other nested classes than those which implement the nested aspect interfaces of their aspect interfaces. In other words, they can declare only implementation nested classes. However, as their subclasses are not transformed in the same way as the classes that directly declare the `provides` clause, their subclasses are allowed to define any nested class it is desired. The transformations applied in these classes will be described in chapter 4.

3.3 Binding classes

Binding classes are responsible to adapt the component to a specific application that it gets integrated into. They translate the component protocol to the application concepts. In other words, they implement the expected contract of the aspect interface mapping it to the base application.

In Caesar, it is defined by means of the `binds` clause. Similarly as the `provides` clause, it is used in the same way as Java's `extends` clause and creates a subtype relationship between the aspect interface and the binding class which defines the clause, as well as obliges the class to implement the nested aspect interfaces. However, binding classes implement methods defined with the `expected` modifier in the aspect interface.

Unlike in implementation classes, nested binding classes *must* be defined using the `binds` clause instead of be declared with the same name of the aspect nested interface. Still, they do *not* need to be defined with the same name. It allows binding classes to provide more than one nested binding of a single nested aspect interface.

Listing 3.4 shows how a binding class looks like in Caesar. In this code, the class `MediatorProtocolBinding` is marked to bind the aspect interface `MediatorProtocol`. As it can be seen, it defines more than one nested binding to the aspect interface `Mediator`.

Similar to implementation classes, binding classes can have subclasses. All features and restrictions discussed for implementation classes are valid for binding classes as well. Therefore, it cannot extend other class or implement or even bind other aspect interfaces. Furthermore, their subclasses can also override their nested classes.

Another important feature of nested binding classes is that they can *wrap* another class. As presented in the beginning of this section, binding classes adapt a component to a particular application. In order to facilitate that, another language construction was inserted: the `wraps` clause.

Defining a binding class with the `wrap` clause maps the binding to a specific class of the base code. Thus, every binding class that wraps a base class can access methods of such class by means of `wrappee`. Hence, inside the scope of the wrapper class every public methods of that base class can be directly accessed as a normal method call, for example `wrappee.getName()` in listing 3.5.

Actually, the wrapper declaration:

```
public class MediatorBinding ... wraps Label {...}}
```

is just syntactic sugar for:

```
public class MediatorBinding ... {
  private Label wrappee;
  public MediatorBinding(Label wrappee) {
    this.wrappee = wrappee;
  }
  ...
}
```

Therefore, when creating instances of wrapper classes a reference of the wrappee class must be provided. Besides that, it is forbidden for wrapper classes to declare constructors with parameters.

```
public class MediatorProtocolBinding
  binds MediatorProtocol {
  //Here above, the class defines that it is
  //the binding class of MediatorProtocol

  public class MediatorBinding
    binds Mediator {

    public void colleagueChanged(
      Colleague colleague) {...}
  }

  public class SpecialMediatorBinding
    binds Mediator {

    public void colleagueChanged(
      Colleague colleague) {...}
  }

  public class ColleagueBinding
    binds Colleague {

    public String getState() {...}
  }
}
```

Listing 3.4: Binding class in Caesar

In listing 3.5, a possible binding implementation[1] of `MediatorProtocol` is declared.

[1] A complete realization of the component as it was defined in AspectJ would need to use the concepts of JPI of the language, but as it is out of the scope of this work, it is not shown here.

In this code, it is used the same application used in section 1.2 for the AspectJ's solution. In this collaboration, `Button` is wrapped by `ColleagueBinding` that binds `MediatorProtocol.Colleague`. It means that `Button` plays the role of `Colleague` in this collaboration as well as `Label` plays the role of `Mediator`.

```
public class MediatorProtocolBinding
  binds MediatorProtocol {

  public class MediatorBinding
    binds Mediator
    wraps Label {
    //It is a wrapper of Label

    public void colleagueChanged(
      Colleague colleague) {
      //Here it access a method of Label
      wrappee.setText(colleague.getState() + ``clicked'');
   }
  }

  public class ColleagueBinding
    binds Colleague
    wraps Button {
    //It wraps Button

    public String getState() {
      return wrappee.getName();
    }
  }
}
```

Listing 3.5: Binding class of MediatorProtocol

Note that, though in the example all nested bindings explicitly declare the `wraps` clause, it is *not* necessary. Binding nested classes can be defined just with the `binds` clause and map the component to a *concept* of the base application which does not have a specific abstraction. Thus, it allows sophisticated mappings, which is one of the problems identified in section 1.2 with the AspectJ's solution. Besides that, these mappings are no invasive. It does not physically change the structure of the abstractions of the base code.

With binding classes the model implements the concept of *on-demand remodularization* outlined in the beginning of this document. Binding classes are the abstractions that allow the decomposition that is achieved only when the component is created using an

implementation and a binding class. This combined with the join-point interception of the language can directly affect the behaviour of the base application.

3.4 Weavelet classes

For a component to become effective it must combine an implementation and a binding class. *Weavelet classes* are responsible for joining these two abstractions in order to create a class that is valid to be instantiated.

In Caesar, weavelet classes are declared using a special `extends` clause. It differs from the common `extends` clause because the type defined in this clause must be an aspect interface followed by an implementation and a binding class between parenthesis. Hence, they allow the creation of the component bound with the application by means of the binding class defined in the clause.

Declaring a weavelet class implies that beyond weaving the binding class with the implementation class defined in the `extends` clause, all nested implementation and binding classes of such classes are implicitly joined as well, without needing to declare it inside the weavelet body.

For example, in listing 3.6, the weavelet class `MediatorProtocolWeavelet` extends the aspect interface `MediatorProtocol` providing implementation for it with the methods declared by the implementation class `MediatorProtocolImpl` and the binding class `MediatorProtocolBinding`. Thus, the component now can be instantiated as shown in the bottom of the code.

```
//This is a weavelet class,
//since it uses the special extends clause
public class MediatorProtocolWeavelet
  extends MediatorProtocol(MediatorProtocolImpl,
     MediatorProtocolBinding) {
}
....
MediatorProtocolWeavelet component = new
   MediatorProtocolWeavelet();
....
```

Listing 3.6: Weavelet class of MediatorProtocol

Having a reference of a weavelet instance it is possible to create the nested aspect interfaces implemented by the component. The name of the objects to be created are those defined in the binding class, because since it can declare more than one nested

binding of the same nested aspect interface, it would be not possible to know which class should be used in that instantiation.

References of weavelet classes become *families* like those ones discussed in chapter 2 for FamilyJ's implementation. Therefore reference type definitions and object creation of nested classes of the component must respect the rules of family polymorphism. With that, the compiler ensures that nested components are not being mixed.

Listing 3.7 shows how components are instantiated. The reference `mediator`, for example, is well typed, because the reference of the type definition is the same as that used to instantiate `MediatorBinding`. However, the reference `colleague` does not respect the rules of family polymorphism; hence it is not allowed by the compiler. Note that a reference of `Label`, for example, is passed to the constructor of `MediatorBinding`. It is because this class wraps `Label` and therefore, as discussed before, requires a reference of such class.

```
...
//Creates two components
final MediatorProtocol component = new
   MediatorProtocolWeavelet();
final MediatorProtocol anotherComponent = new
   MediatorProtocolWeavelet();
...
//Creates a new nested component
component.Mediator mediator = component.new
   MediatorBinding(label);
...
//This is not allowed because the families are different
anotherComponent.Colleague colleague = component.new
   ColleagueBinding(button);
....
```

Listing 3.7: Component creation

3.5 Wrapper Recycling operator

With the new language constructions shown in this chapter, it is possible to express generic functionalities and bind them to a particular application, but there is a problem concerning the object creation yet.

Section 3.3 states that binding classes may wrap abstractions from the base code. It helps developers to join component roles with the application without changing the

modular structure of the base code. Hence in spite of avoiding the problem identified in the AspectJ's solution, it gives means to specify more sophisticated mappings since binding classes are not needed to wrap explicitly one abstraction but can define their own logic for a particular concept in the application which may not be declared in an abstraction.

However, using wrappers instead of changing the structure of the code leads to a problem. As the remodularization is not introduced into the base application but just decorates abstractions or concepts, whenever the remodularization must be applied in certain collaboration, new instances of the wrappers have to be created, even if the the abstraction or abstractions which play a certain role in the collaboration have already been wrapped by the component.

This problem causes that wrappers lose their state and identity, therefore using wrappers in this way the components would not be able to maintain internal structure losing important features of the model.

In order to avoid this problem, Caesar provides a new operator to create wrappers: the *wrapper recycling operator*. This operator plays the same role of the `new` Java's operator, but they only create new instances of the wrappers, if the objects passed to the constructor have not been wrapped before by the component of that collaboration.

The wrapper recycling operator is activated whenever the developer declares a statement that calls a constructor as a normal method call, without using the operator `new`. For example `component.MediatorBinding(label)` in listing 3.9.

This operator is available for all nested binding classes, even if it does not explicitly declares a `wraps` clause. The semantics of the wrapper recycling operator in these cases is the same as in the cases that the bindings are explicitly declared wrappers. If the parameters passed have already been passed once, it does not create a new reference of the binding class.

The wrapper recycling operator is object-oriented. If there are two objects of the same class and these two objects use the wrapper recycling operator to instantiate objects of their nested classes, even that the references passed to the constructor point to the same object, new wrappers will be created. For example, `mediator` and `mediator4` in listing 3.9 are references to distinct objects.

Before presenting an example of using wrapper recycling, a new binding of `Mediator` is introduced because both bindings `MediatorBinding` and `ColleagueBinding` declared in section 3.3 are explicitly declared wrappers. Hence it would be not possible to show the use of the operator for ordinary binding classes. Listing 3.8 shows this new binding `SpecialMediatorBinding`. Note that it is declared inside `MediatorProtocolBinding` that was presented before. All other abstractions remain the same as declared.

```
public class MediatorProtocolBinding
  binds MediatorProtocol {
  ....
  public class SpecialMediatorBinding
    binds Mediator {
    private Label label;
    private String name;
    public SpecialMediatorBinding(
      Label label, String name)
    {
      this.label = label;
      this.name = name;
    }

    public void colleagueChanged(
      Colleague colleague) {

      label.setText(name + ``:'' + colleague.getState()
          + ``clicked'');
   }
  }
  ...
}
```

Listing 3.8: New binding for Meditor

Now with this new binding, some examples of wrapper recycling can be shown. In listing 3.9, some references of `MediatorBinding` and `SpecialMediatorBinding` are defined. For example `mediator` and `mediator2` which are initialized with the return of the wrapper recycling call. If after these statements the references are compared (`mediator == mediator2`) the result of the computation is true, they are references of the same object. It happens also if `specialMediator` and `specialMediator2` are compared, however it will be false if `specialMediator` is compared with `specialMediator3`. As discussed, the wrapper recycling is object-oriented, therefore the references `mediator` and `mediator4`, for example, point to different objects.

Hence, using this new operator, instances of the components can be maintained allowing the components to have internal state and remain their identity. Therefore, it avoids the problems outlined in the beginning of this section without changing other parts of the model described here.

```
...
final MediatorProtocol component = new
   MediatorProtocolWeavelet();
```

```
final MediatorProtocol anotherComponent = new
   MediatorProtocolWeavelet();

//Creates a new nested component
component.Mediator mediator =
  component.MediatorBinding(label);
//Here, it will not create a new MediatorBinding
component.Mediator mediator2 =
  component.MediatorBinding(label);
//It creates a new MediatorBinding again
component.Mediator mediator3 =
  component.MediatorBinding(new Label());
//It also creates a new MediatorBinding
anotherComponent.Mediator mediator4 =
  anotherComponent.MediatorBinding(label);

//It is true
mediator == mediator2;
//It is false
mediator2 == mediator3;
//It is false as well
mediator == mediator4;

String name = ``Button'';
//Creates a new SpecialMediatorBinding
component.Mediator specialMediator =
  component.SpecialMediatorBinding(label, button);
//Does not create a new reference
component.Mediator specialMediator2 =
  component.SpecialMediatorBinding(label, button);
//Creates a new reference again
component.Mediator specialMediator3 =
  component.SpecialMediatorBinding(new Label, button);
//It is true
specialMediator == specialMediator2;
//It is false
specialMediator == specialMediator3;

...
```

Listing 3.9: Using wrapper recycling operator

Beyond creating the wrappers in this way, developers are able to disassociate a base

object from its respective wrapper. It allows wrappers to be instantiated even if they have already been created once. This is achieved by means of the new *wrapper destructor operator*: `~`. Syntactically, using this operator in the language is very similar with the wrapper recycling operator. The only difference is that before the wrapper type to be created the operator `~` must be inserted.

The semantics of this operator is that if the parameters passed to the operator have already been used to create a wrapper, it destroys this wrapper. Therefore, when a new wrapper recycling call is performed with those parameters, a new wrapper is created. Otherwise, if the parameters have not been passed yet, it does not have any effect.

For example, listing 3.10 presents the use of such operator. After creating `mediator`, it calls the destructor of such wrapper, thus `mediator2` is a reference of new wrapper. Note that it destructs the wrappers only in the context of the component that they were created. For example, after creating a wrapper using `anotherComponent`, it destroys the wrapper instantiated using the `component` reference. If the wrapper recycling is used again in the context of `anotherComponent`, it does *not* instantiate a new wrapper.

```
...
final MediatorProtocol component = new
   MediatorProtocolWeavelet();
final MediatorProtocol anotherComponent = new
   MediatorProtocolWeavelet();

//Creates a new wrapper
component.Mediator mediator =
  component.MediatorBinding(label);
//It removes the wrapper of label
component.~MediatorBinding(label);
//Creates again!
component.Mediator mediator2 =
  component.MediatorBinding(label);
//Now it does not instantiate the wrapper
component.Mediator mediator3 =
  component.MediatorBinding(label);

//It is false now!
mediator == mediator2;
//But it is true
mediator3 == mediator2;


//Creates because it is anotherComponent
```

```
anotherComponent.Mediator anotherMediator2 =
  anotherComponent.MediatorBinding(label);

//Destroys the wrapper, but from the reference component
component.~MediatorBinding(label);

//Now it does not instantiate the wrapper
anotherComponent.Mediator anotherMediator3 =
  anotherComponent.MediatorBinding(label);
//Removes the wrapper
antoherComponent.~MediatorBinding(label);
//Nothing happens here
antoherComponent.~MediatorBinding(label);
...
```

Listing 3.10: Using wrapper recycling operator

Using wrapper recycling operator developers can manage the wrapper creation avoiding the problems of lose identity and state of such wrappers. Besides that, the wrapper destructor operator gives more control to them, allowing destroying the wrappers when it is desired, thereby the wrapper destructor operator can be used to release the wrapper objects for garbage collection.

Chapter 4

Aspect interfaces in the Caesar compiler

This chapter presents how aspect interfaces and their concepts presented in the last chapter have been implemented in Caesar using the notions of virtual types and delegation as presented in chapter 2.

The virtual types and delegation presented in section 2 are the basis for the implementation that has been performed during this thesis. However these mechanisms themselves were not totally adapted to be directly used for the implementation of the aspect interfaces. Therefore, this chapter discusses also the improvements that have been implemented concerning virtual types and delegation implementations.

In the following, section 4.1 defines how the delegation mechanism has been used. Section 4.2 describes the transformations in the Abstract Syntax Tree to allow using the FamilyJ's concepts in this implementation. The next section 4.3 shows how the components are created at runtime. Section 4.4 discusses the improvements in the FamilyJ's implementation. In section 4.5 the approach used to implement the wrapper recycling operator is presented. Finally, section 4.6 gives an overview of the source code additions in the compiler.

4.1 Using the delegation mechanism

One of the main mechanisms for the concepts presented in this thesis is delegation. Delegation is an important concept for implementing aspect interfaces because it allows class hierarchies to be assembled at runtime without loosing late binding features. This section shows how the delegation mechanism available in FamilyJ has been used to

implement components using two separated abstractions.

As described in chapter 2, FamilyJ uses delegation to allow virtual classes to execute the right method implementations when such classes do not know their super type implementation at compile time. It permits, for example, a super class of a virtual class to be overridden, because the class hierarchy is assembled depending of the context in which the virtual class is instantiated.

This feature of FamilyJ's model gives a clue of how aspect interfaces can be implemented. But before showing the approach used to allow aspect interfaces in the model, some features of the delegation mechanism available in FamilyJ are refreshed.

In that model, every method call that is not supported by the protocol of the receiver of the message, but for the protocol of its super class, is delegated to the super class instance that is defined at runtime. The super class implementation may be any class that implements the clean interface of such a super class. Hence, it can obviously be an instance of the super class itself, but it can also be an instance of any subclass of such class.

When delegating a method call, FamilyJ provides an object to play the role of the receiver of calls to methods defined in the same scope that the method is being executed. This object, `self` in FamilyJ, is the first receiver of the message that has not been performed in this scope (`this` in Java, for example).

The first thought which could arise when providing such object is passing itself to the next method implementation if it is the first receiver of the message, or the object that has been received in methods that have already been called through delegation. However, as discussed before it would imply that, in some configurations, some methods of the protocol would become disabled.

Therefore, in FamilyJ the object, and its class representation, to play this role is created at runtime depending on the context that the method is being executed. The new class created to represent this object implements the clean interface of the parent reference of the object that is currently executing the method, providing implementation of the methods of such interface by delegating the method to the right receivers.

For example, if the clean interface of the parent defines a method that is not defined in the clean interface of the first receiver, it delegates the method to the current parent reference, otherwise it delegates to the first receiver or the first object in the runtime hierarchy that implements that method.

Now with these features elucidated, let's try to map these features to the requirements of execution of the component as they were described in this work. On one hand, binding or implementation classes provide only part of the protocol of aspect interfaces,

but on the other hand they can use the whole aspect interface protocol as their own.

Even though binding and implementation classes do not know about the existence of each other, they know that they become available to be instantiated only when they are woven with a class which implement the other part of their protocol. Besides that, they have a common super type: the aspect interface. Hence an instance of the implementation class, for example, can be set as parent of a binding object.

By setting the implementation object as parent of a binding instance, calls to provided methods that are not implemented by the binding class are delegated to the parent reference. The parent reference is an instance of the implementation class which does implement the method. In this way, the requirement of binding classes call provided methods can be achieved, but how can implementation instances call expected methods?

Note that the first receiver of method calls is always the binding instance because it is the *child* object and is passed to the implementation object by means of the method dispatcher instance. Therefore this object acts as `self` for all calls to methods that are defined in the aspect interface. Thus all method calls for `self` that are defined in the aspect interface performed by the implementation instance reach the binding instance that executes the method if it is defined on it or calls the implementation to execute otherwise. In this way, the expected methods are also bound to the right implementation. So this is how implementation instances can call expected methods.

In order to show an example of these collaborations using delegation, a more complex collaboration than that defined in the mediator pattern example is presented. The expected method `isAvailable()` was added to the aspect interface `MediatorProtocol`. This method is called by the implementation class in the method `setMediator(Mediator)` to ensure that the `Colleague` is available. Listing 4.1 shows the new implementation of `setMediator(Mediator)` as well as the implementation of the new expected method.

With this new implementation, it is possible to present a collaboration between a binding and a implementation instances using the approach introduced in this section. Figure 4.1 presents the collaboration diagram of an hypothetical `setMediator(Mediator)` execution.

In this diagram, a client of the component sends the message `setMediator(Mediator)` to the binding reference. This method is not implemented by the binding class since it is a provided method, thus the method is delegated to the parent reference, which was set with the implementation instance. When the `setMediator(Mediator)` method calls `isAvailable()`, this message is sent to the binding instance that now has the implementation of the method and executes it.

```
//A special implementation with
//a more complex collaboration
public class SpecialMediatorProtocolImpl
  provides MediatorProtocol {

  public class Colleague {
    private Mediator mediator;
    public void setMediator(Mediator mediator) {
      if (isAvailable())
        this.mediator = mediator;
    }
    ...
  }
}

public class MediatorProtocolBinding
  binds MediatorProtocol {

  //Implementation of the new protocol
  //providing the new expected method implementation
  public class SpecialColleagueBinding
    binds Colleague
    wraps Button {

    public boolean isAvailable() {
      return wrappee.isEnabled();
    }
    ...
  }
  ...
}
```

Listing 4.1: New implementation and binding classes

Note that it is not passing directly the first receiver of the method (`this`) as `self` parameter, but it is passing a method dispatcher - created by means of the method `getDispatcher(object, object)` - reference instead. In this case it does not have any implication since all messages will be dispatched to the binding object. However, if the implementation class had defined a public method that is not in the aspect interface, the implementation instance would be able to call this method without the dispatcher reference.

For example, the implementation class `MediatorProtocolImpl.Colleague` shown in

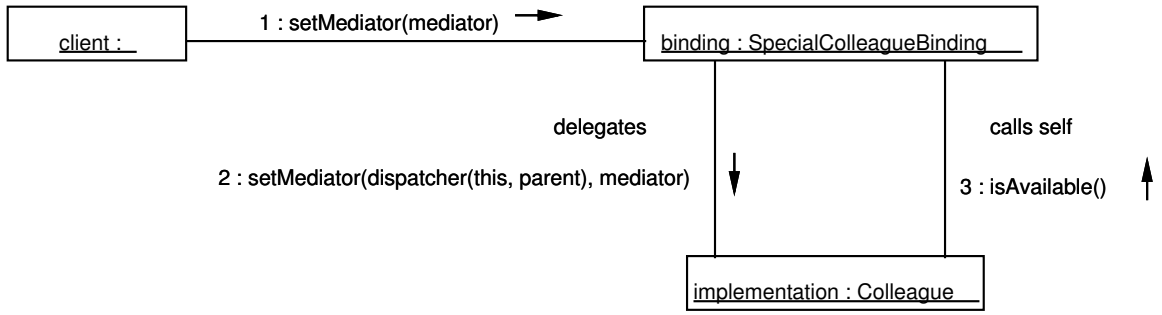

Figure 4.1: Collaboration between binding and implementation using delegation.

listing 4.2 declares the method `isMediatorSet()` that is not declared in the aspect interface `MediatorProtocol`. When it calls this method, the method dispatcher sends this method to the implementation instance instead of the binding object.

```
//A special implementation with
//a more complex collaboration
public class SpecialMediatorProtocolImpl
  provides MediatorProtocol {

  public class Colleague {
    private Mediator mediator;
    public void setMediator(Mediator mediator) {
      if (isAvailable() && isMediatorSet())
        this.mediator = mediator;
    }
    //This method is not declared in the interface
    public boolean isMediatorSet() {
      return mediator != null;
    }
    ...
  }
 }
}
```

Listing 4.2: Implementation class declaring a new public method

Listing 4.3[1] presents a theorical implementation of the method dispatcher in the case shown in figure 4.1 where the method `setMediator(Mediator)` is delegated from the binding object to the implementation instance using the code shown in 4.2. The concep-

[1]This code does not show the real implementation of the dispatcher, it is just for illustrating the concept, for more details about the real implementation please refer to [16].

tual class `Dispatcher` implements the interface `MediatorProtocolImpl.Colleague`[2] delegating the method calls to `self` if the method is declared in the aspect interface or for `parent` otherwise. Note that when dispatching the method `isMediatorSet()` it needs to provide a method dispatcher similar to `Dispatcher` presented.

```
public class Dispatcher
  implements MediatorProtocolImpl.Colleague {
  //Parent on that method dispatch
  private MediatorProtocolImpl.Colleague parent;
  //Self on that method call (it will be 'this' in this
     case)
  private MediatorProtocolBinding.ColleagueBinding self;

  public void setMediator(Mediator mediator) {
    self.setMediator(mediator)
  }
  public Mediator getMediator() {
    return self.getMediator();
  }
  public String getState() {
    return self.getState();
  }
  public boolean isMediatorSet() {
    //Here it is dispatched to the parent object
    return parent.isMediatorSet();
  }
}
```

Listing 4.3: Theoretical implementation of the method dispatcher

In summary, the approach used to allow binding and implementation classes to call methods of each other is by using delegation. The implementation object is set as parent of the binding instance and using delegation they can call provided or expected method as their own. In order to use this mechanism, some transformations have to be performed to adapt the new abstractions presented in this work to the FamilyJ implementation. The next section shows these transformations and section 4.3 describes how the objects are created to assemble this structure at runtime.

[2]The creation of this interface is shown in section 4.2.

4.2 Transformations

In the last chapter, the approach used in the compiler to implement binding and implementation classes using FamilyJ's delegation mechanism has been presented. In order to enable the approach, the implementation carried out during this thesis adds new transformations to be performed before FamilyJ's transformations shown in chapter 2. These new transformations adapt the new abstractions of the model to FamilyJ's abstractions allowing direct use of delegation as discussed in the previous section. This section describes these transformations and states the restrictions that come with them.

As this section makes extensive use of FamilyJ's concepts described in chapter 2, before starting to show the transformations some terms are elucidated: *clean classes*, *virtual classes* and *override classes*. Clean classes are classes that have a clean interface and use delegation in place of common inheritance. Virtual classes are inner classes with clean interfaces and use delegation as well, in addition to that, they can be overridden by subclasses of the enclosing class and their instantiation is late bound. Override classes are virtual classes that override other virtual class.

Now with these terms the transformations are shown. In the following, how the new abstraction of Caesar presented in chapter 3 are transformed to FamilyJ's abstractions is described.

4.2.1 Aspect interface transformations

All transformations performed to aspect interfaces are realized in the beginning of the compilation. After parsing a compilation unit and creating the respective *Abstract Syntax Tree*, AST for short, the compiler performs the transformations needed for these interfaces.

Hence in this turn, if an interface declared with the modifier `aspect` is found in the AST, this interface is replaced by a clean class if it is not a nested interface; otherwise it is substituted by a virtual class.

As described in chapter 3, nested aspect interfaces do not need to be defined with the modifier `aspect` because if they are in the scope of an aspect interface, they are implicitly declared aspect interfaces. It is because all interfaces declared in the scope of an aspect interface are transformed to virtual classes. However aspect interfaces can have subtypes and their subtypes can override their nested interfaces. Therefore, nested interfaces that override a super aspect interface are transformed to override classes.

In order to turn interfaces into classes, it is necessary to provide implementation for their methods. Thus when it transforms aspect interfaces, it inserts empty bodies for

the interface methods. The problem is that methods that return a value cannot have an empty body; they need at least the return statement. Therefore, when the method defines a return type that is not `void`, it is injected a return statement in the empty method body. The return values of these statements are: `null` if the method returns a reference type; `0` if the return is a numeral; and `false` if it returns a `boolean`.

This transformation is performed to maintain the consistency of the binding and implementation classes. As binding and implementation classes do not implement the whole protocol, they would not be valid Java classes. Hence even though the generated class has empty methods, these methods will never be executed at runtime.

Listing 4.4 shows how the aspect interface `MediatorProtocol` presented in the last chapter looks like after this transformation.

```
public aspect clean class MediatorProtocol {

  public aspect virtual class Mediator {
    public expected void colleagueChanged(
      Colleague colleague) {
    }
  }

  public aspect virtual class Colleague {
    public provided void setMediator(Mediator mediator) {
    }
    public provided Mediator getMediator() {
      return null;
    }
    public expected String getState() {
      return null;
    }
  }
}
```

Listing 4.4: Aspect interface transformations

In this code, it can be seen that the modifier of aspect interfaces `aspect` remains after the transformation, and all nested interfaces receive this modifier marking that the class is the representation of an aspect interface. In addition to that, the implicit modifiers of interfaces (`abstract` and `interface`) and interface methods (`abstract`) have been removed.

Keeping the `aspect` modifier is important for the compilation process. The compiler must perform special checking and semantic analysis for aspect interfaces. Since the

aspect interface is transformed to clean classes without this information the compiler would not know when it is compiling this kind of abstraction.

Using this approach brings some restrictions to aspect interfaces concerning subtype relations. Aspect interfaces can be subtype only of other aspect interface. Besides that, they can declare a subtype relationship with just one aspect interface. These two restrictions appear because the interfaces are transformed to classes and have therefore to follow the rules of ordinary Java classes. Another restriction is that they cannot be declared as nested interfaces of ordinary interfaces or classes. This restriction comes from the clean classes that they cannot be declared nested.

4.2.2 Implementation class transformations

As well as aspect interfaces, implementation classes are transformed to clean classes. Furthermore, the *providing* relation - the relationship created by the `provides` clause - is changed to a subclass relationship that after is changed to a subtype relationship as discussed before for clean classes. Hence implementation classes are changed to be subclasses of the transformed aspect interface that it provides.

After these transformations, nested implementation classes can be transformed. As implementation classes are subclasses of classes that contains virtual classes (the transformed aspect interfaces), their nested classes are able to override the nested virtual classes of their super class. Therefore, nested implementation classes are turned into override classes, and thus they have the implicit subclass (subtype after FamilyJ's transformations) relationship with the aspect interfaces.

This transformation shows why nested implementation classes do not need to define the provides clause and must be declared with the same name of the aspect interfaces that they provide. Furthermore, as all nested implementation classes are replaced to override classes, they *cannot* declare other nested classes than those ones defined by their aspect interfaces. It would cause an error during the compilation because it would override a class that does not exist.

After these transformations `MediatorProtocolImpl` described in section 3.2 of the last chapter would look like the code in listing 4.5.

Note that in this code the `providing` modifier has been added to the classes. As well as it happens to aspect interfaces it is inserted to allow providing classes to be identified during the compilation process. However, it is different of aspect interfaces because this modifier is not available in the language, it is used only for the internal structures of the compiler; developers therefore are not able to use that.

With these transformations implementation classes now override the provided methods of the transformed aspect interface using common method overriding available in Java. Hence, they provide a real implementation of the provided contract defined in the aspect interface, hiding the empty methods declared in those classes.

```
public providing clean class MediatorProtocolImpl
  extends MediatorProtocol {

  public providing override class Colleague {
    private Mediator mediator;
    public void setMediator(Mediator mediator) {
      this.mediator = mediator;
    }
    public Mediator getMediator() {
      return mediator;
    }
  }
}
```

Listing 4.5: Implementation class after transformations

Transforming implementation classes into clean classes implies that implementation classes must follow the rules of clean classes described in chapter 2. Hence, as clean classes can only have sub clean classes, subclasses of implementation classes have to be *explicitly* declared clean using the `clean` modifier available in FamilyJ. As well as they cannot be declared inside the scope of an ordinary class.

Besides that, as described before, they cannot declare other inner classes than those declared in the aspect interface, though it is not true for their subclasses, which are able to declare any nested class the developer wants. It is because nested classes of sub implementation classes do not need to be transformed to override classes by the compiler, the developer have to explicitly declare it if it is desired by using the `override` modifier available in FamilyJ.

4.2.3 Binding class transformations

The transformations performed to binding classes are very similar to those presented for implementation classes. Binding classes are also transformed to clean classes. As well as the *binding relationship* (created by means of the `binds` clause) is changed to a subclass relation.

However, as a binding class may provide more than one implementation of the same nested aspect interface using different names, they cannot be transformed to override

classes. Thus, nested binding classes are turned into virtual classes with an explicit subclass relationship with the transformed super nested aspect interface.

```
public clean binding class MediatorProtocolBinding
  extends MediatorProtocol {

  public clean binding class MediatorBinding
    extends Mediator {
    //New field wrappee
    private Label wrappee;
    public MediatorBinding(Label wrappee) {
      this.wrappee = wrappee;
    }

    public void colleagueChanged(
      Colleague colleague) {
      _getWrappee().setText(colleague.getState() + ``
         clicked'');
   }

   public Label _getWrappee() {
     return wrappee;
   }
  }

  public clean binding class ColleagueBinding
    extends Colleague {

    private Button wrappee;
    public ColleagueBinding(Button wrappee) {
      this.wrappee = wrappee;
    }

    public String getState() {
      return _getWrappee().getName();
    }
    public Button _getWrappee() {
      return wrappee;
    }
  }
}
```

Listing 4.6: Binding class after transformations

In this way they can also override the empty super methods using the common method overriding from Java, because it will use delegation instead of ordinary inheritance to dispatch methods even that it does not override the super aspect interface.

As described before, binding classes may wrap other classes. Therefore, some transformations have to be performed if a binding class explicitly defines the `wraps` clause. This declaration leads the compiler to insert a new field called `wrappee` and add - or change if the constructor is already declared - the constructor to receive the `wrappee` reference. Beyond that, the `_getWrappee()` method is introduced in the class to allow sub classes to use `wrappee` in the language also. Due to that, all references to `wrappee` are changed to common method calls that will reach this new method.

Similar to implementation classes, a modifier that is not enabled in the language is added to binding classes. Thus, binding classes are marked with the modifier `binding` for their identification in the compilation process. Listing 4.6 shows the binding class `MediatorProtocolBinding` defined in section 3.3 after these transformations.

As it can be seen in this code, now the other part of the contract (the expected contract) from the aspect interface is implemented, and the empty methods that have been inserted in the aspect interface are overridden by the binding class. Therefore empty method implementations of the interface are no longer called, since the implementation class has overridden the provided methods, and the binding classes have hidden the expected contract. Now method calls are performed as described before in section 4.1.

Defining binding classes in this way brings some restrictions as well. Binding classes must declare a clean interface, since they are transformed to clean classes, therefore they have the same restrictions describe to implementation classes concerning clean classes. However, as discussed, they *can* declare other inner classes that are not defined by the aspect interface. Another restriction arises when the `wraps` clause is declared: the constructors *cannot* be defined with parameters, because it could bring inconsistency to the wrapper recycling operator.

4.2.4 Weavelet class transformations

Likewise other abstractions presented in this section, weavelet classes are transformed to clean classes. However, as they do not have nested classes, there is no special transformation for nested classes of this abstraction.

Weavelet classes are turned into subclasses of the binding class that is defined in the special `extends` clause of such abstractions. Hence, weavelet classes remain subclasses of the aspect interface, since the binding class is a subclass of the aspect interface. Furthermore, objects of this type can be used to create nested binding classes. These

nested classes hence are the classes able to be initialized using the factory methods inserted in the binding class for their virtual classes. Note that nested binding classes are transformed to virtual classes; it implies that their enclosing classes define a factory method for their creation. These creations are clarified in section 4.3.

In the same way that binding and implementation classes have a special modifier that are not allowed in the language, weavelet classes are also marked with a special modifier, `weavelet`, for allowing their identification in the compilation process. The code in listing 4.7 presents the weavelet class `MediatorProtocolWeavelet` declared in section 3.4 after these transformations.

```
public clean weavelet class MediatorProtocolWeavelet
  extends MediatorProtocolBinding {
}
```

Listing 4.7: Weavelet class of MediatorProtocol

The first implication of this transformation is the same as discussed before for classes that are turned into clean classes, so weavelet classes must respect the rules of such abstraction. Another implication of these transformations is that the constructors declared with parameters by binding classes must be explicitly declared in the weavelet classes. It is because as they are subclasses of binding classes they must respect the rules of the Java constructors in subclass relationships.

4.3 Hierarchy construction

Now with the classes transformed in this way, the model is able to use FamilyJ's delegation mechanism as described in chapter 4.1. For doing so, objects must be constructed in such way that represents that structure at runtime.

Hence, other transformations are needed in order to adapt the aspect interfaces and the other concepts to the delegation mechanism. These transformations change the object constructors of these new abstractions to instantiate the right parents and children.

As described before, the approach used by Caesar to execute the components is using delegation, where an implementation instance is set to be parent of a binding object. FamilyJ's implementation inserts a new constructor for every constructor declared in the clean or virtual classes which receives as first parameter the reference `parent` to be set for the object that is being created. The type of this reference is the same set by the developer to be super class of the current class. This is the way that instances of subclasses of the same class can be passed to this new constructor and therefore can be set parent of each other.

Thus, after performing the transformations described in the last section, where aspect interfaces, as well as their binding and implementation classes, are turned into clean classes and binding and implementation classes become subclasses of the transformed aspect interface, the new constructor introduced by FamilyJ allows implementation objects to be set parent of binding instances since they have a common super clean class.

In this way, they could already be directly created by developers, however they would have to provide information about which binding and implementation class should be used in every instantiation. For avoiding that, weavelet classes have been inserted in the language where developers declare it only once, in the weavelet declaration, and after that they can instantiate it as an ordinary Java class.

Weavelet constructors are responsible to create the runtime hierarchy structure of the components. They are transformed to instantiate the binding and implementation objects that compound the whole functionality, and set their respective parents. Listing 4.8 shows the weavelet constructor after this transformation. The second constructor declared in this class will be explained soon; by now the first one is focused.

```
public clean weavelet class MediatorProtocolWeavelet
  extends MediatorProtocolBinding {
  //The constructor is changed
  //to create the class hierarrchy
  public MediatorProtocolWeavelet() {
    super(new MediatorProtocolBinding(new
       MediatorProtocolImpl());
  }
  //Constructor provided by FamilyJ
  //that is changed too.
  public MediatorProtocolWeavelet(MediatorProtocolBinding
     parent) {
    super(new MediatorProtocolBinding(new
       MediatorProtocolImpl(parent));
  }
  ...
}
```

Listing 4.8: Weavelet class of MediatorProtocol

By taking a look in this new constructor a restriction of the model arises. One of the two abstractions (binding or implementation) cannot define constructors with parameters. Since this constructor calls directly both constructors, it would be not possible to know for which of them the parameters should be passed. Hence in Caesar, implementation classes *cannot* declare constructors with parameters. It seems to be more acceptable

to have binding class able to declare constructors because they are related with the application. Since they adapt the component to the base code, they need to define links to base objects. Besides that, if implementation classes need to create objects, it could be specified expected methods in the aspect interface that would be implemented by the bindings that would provide such objects.

Having this restriction respected by implementation classes, the model can already create the class hierarchy needed to enable delegation as described in section 4.1. Figure 4.2 shows a collaboration diagram illustrating the runtime class hierarchy created when a weavelet class is instantiated. Note that the instance responsible to receive all messages from the client is not the binding object as it was described in section 4.1, but the weavelet. It does not imply any change in the structure since the weavelet object delegates first to the binding object, but it brings a feature to weavelet classes, now they can override any method of the aspect interface.

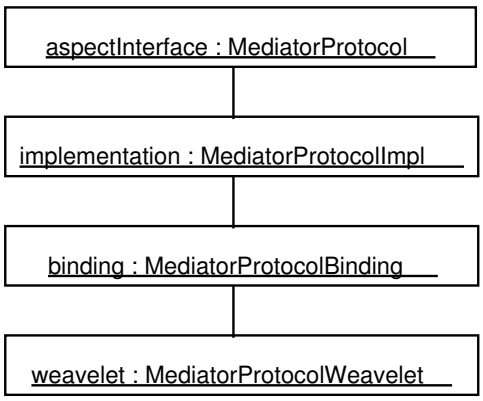

Figure 4.2: Runtime hierarchy of the component.

This approach is used to create the most outer components, but creating this structure already enables instantiation of nested classes of the component. As discussed before, for each constructor of the virtual classes a factory method is inserted in the enclosing class of such virtual class and all virtual class instantiation is changed to call these factory methods instead of directly create the objects by using the `new` operator. With that, FamilyJ allows late bound instantiation of virtual classes. This combined with the structure created for enclosing instances is everything that is needed here.

This is because factory methods use other factory methods to obtain the parent reference of virtual and override classes. In the case of override classes, the parent reference is gotten by calling the same factory method in the super class, like a common super call. In the case of virtual classes that are explicitly declared to extend another virtual class, the parent reference is obtained through the factory method of that super virtual class.

Thus, as in the structure created objects always use delegation to invoke methods, and factory methods become common methods in the classes, they are called by using delegation as well. In this way, if the structure is already supporting the execution of the methods using delegation, the right objects are created.

Figure 4.3 shows a conceptual collaboration diagram of the instantiation of nested components. Note that details of delegations are not shown in this diagram. The weavelet instance just delegates the method call to the binding object since its class does not declare this method. The binding instance creates a `MediatorBinding` passing parent as the result of the `createMediator()` method since `MediatorBinding` is set subclass of `Mediator`. This method call reaches the implementation object because it is the first object in the hierarchy whose class declares this method. So, in this method the parent reference is obtained by calling `createMediator()` for its parent, `aspectInterface`, since it is creating an override reference. The `aspectInterface` reference just returns a new instance of `Mediator` because it is not a override class and there is no super class directly declared for this virtual class. With this object, the implementation reference can return a new instance of its `Mediator` override class with the right parent set.

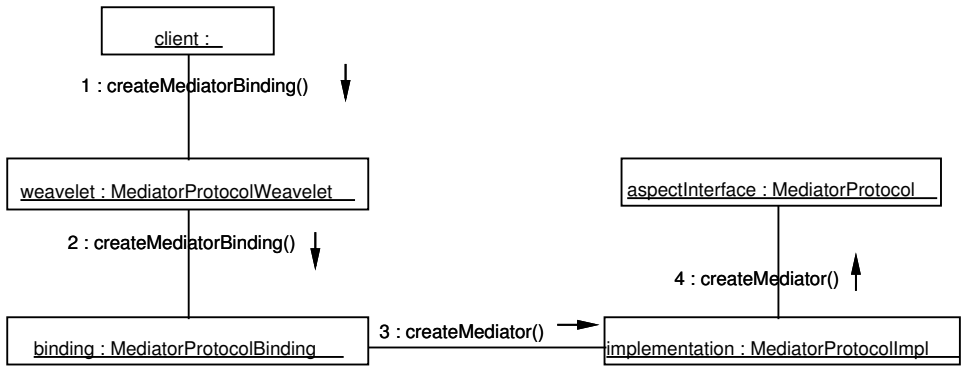

Figure 4.3: Nested components creation.

After this collaboration the runtime hierarchy created is as shown in figure 4.4. It looks like the runtime structure of enclosing components. The only difference is that nested components do not have weavelet because the role of weaving nested components is played also by the weavelet of the enclosing objects as it has been shown in figure 4.3. Therefore, it creates the right structure of nested objects as well, with a binding object delegating to an implementation instance.

With this transformation, simple configurations of the component as shown here can already be used. But there are problems yet concerning subclasses of binding classes. Before describing these problems a more complex example is shown extending the classes that have been created until here. Listing 4.9 shows this new configuration.

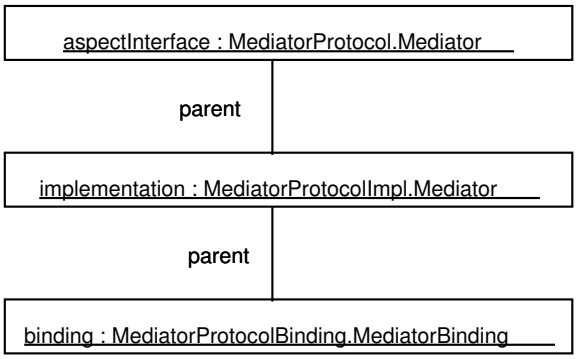

Figure 4.4: Runtime structure of nested components.

```
//The weavelet now weaves other
//implementation with another binding
public class MediatorProtocolWeavelet
  extends MediatorProtocol(SpecialMediatorProtocolImplSub,
    SpecialMediatorProtocolBindingSub)  {
}
//This is a subclass of MediatorProtocolBinding
public clean class SpecialMediatorProtocolBinding
  extends MediatorProtocolBinding {
  ...
}
//One more level
public clean class SpecialMediatorProtocolBindingSub
  extends SpecialMediatorProtocolBinding {
  ...
}
//This is a subclass of MediatorProtocolImpl
public clean class SpecialMediatorProtocolImpl
  extends MediatorProtocolimpl {
  ...
}
//Another level for implementation too
public clean class SpecialMediatorProtocolImplSub
  extends SpecialMediatorProtocolImpl {
  ...
}
```

Listing 4.9: New configuration for MediatorProtocol

Using only the transformations in weavelets constructors presented in this section with the new configuration shown in listing 4.9 , it would not create the right runtime hierarchy. Besides that it would generate an error during the compilation. It is related with the constructors inserted by FamilyJ in the abstractions, so before showing why these errors would occur, a closer look in these constructors is taken.

As described before, FamilyJ enables one more constructor for each constructor declared by the developer inserting the parent reference. The type of this new reference is the super class explicitly declared by the developer. Besides that the body of the new constructor is copied from the old constructor that now just calls this new constructor passing as `parent` a reference of an object directly created in this constructor.

Inserting this new constructor and creating an object in the old constructor enable the creation of the hierarchy when calling the old constructor, because it always creates a fresh reference to act as parent. However, if the new constructor is called directly it does not create the hierarchy, because it assumes that the hierarchy has already been created before.

Now, let's go back to the weavelet constructors. The constructor created in the weavelet classes calls the constructor of the binding class passing as parent a reference of the implementation class. Using the code presented in listing 4.9, it is

```
new SpecialMediatorProtocolBindingSub(new
   SpecialMediatorProtocolImplSub())
```

The problem concerning the error in the compilation arises because the constructor inserted in the binding class `SpecialMediatorProtocolBindingSub` expects a reference of the type `SpecialMediatorProtocolBinding`, however the reference passed is of the implementation type instead, so the compiler cannot find the constructor called by the weavelet.

The other problem is due to how the constructor added by FamilyJ is implemented. Using this constructor means that the class hierarchy has already been initialized, but this is not the case here since the hierarchy passed is the implementation hierarchy, but the binding hierarchy has not been created yet.

These problems do not happen with binding nested classes because they are created through factory methods which allow late bound instantiation. Therefore the structure does not need to be created by hand as non-nested classes that must be directly created by the weavelet class.

In order to avoid these problems a new constructor is injected into explicit or implicit subclasses of non-nested binding classes. The parent parameter of this constructor is of the aspect interface type. Before calling the constructor introduced by FamilyJ it directly creates a new object of the super class passing the received parameter as

parent. Hence, when creating this new object the constructor call reaches this same kind of constructor in the super class if it is not the real binding class yet, or the constructor inserted by FamilyJ if it is an explicitly declared binding class. With that, it creates the binding structure as well, because when it arrives in a binding class, the structure is already created.

Listing 4.10 shows the constructors created by FamilyJ and the new constructors inserted to allow the creation of the right structure.

```
public clean class SpecialMediatorProtocolBinding
  extends MediatorProtocolBinding {
  //Constructor inserted by FamilyJ
  public SpecialMediatorProtocolBinding(
    MediatorProtocolBinding parent) {
    ...
  }
  //New constructor inserted
  public SpecialMediatorProtocolBinding(
    MediatorProtocol parent) {
    this (new MediatorProtocolBinding(parent));
  }
  ...
}
//One more level
public clean class SpecialMediatorProtocolBindingSub
  extends SpecialMediatorProtocolBinding {
  //Constructor inserted by FamilyJ
  public SpecialMediatorProtocolBindingSub(
    SpecialMediatorProtocolBinding parent) {
    ...
  }
  //New constructor inserted
  public SpecialMediatorProtocolBindingSub(
    MediatorProtocol parent) {
    this (new SpecialMediatorProtocolBinding(parent));
  }
  ...
}
```

Listing 4.10: New constructors added by Caesar

The second constructor introduced in listing 4.8 has been inserted just to maintain consistency of clean classes, in addition to that it might allow other kind of structure to be created. Duo to this, the new constructor added into binding classes are also inserted

in subclasses of implementation classes, because without that the problems discussed for sub binding classes would arise also for sub implementation classes, since the type of the parameter received is a binding type and it would not create the implementation hierarchy.

In summary, the constructors of weavelet classes are transformed to create the runtime object hierarchy. For avoiding the problems that would occur by using just the constructors added by FamilyJ, new constructors are inserted in the non-nested binding classes, as well as for no-nested implementation classes to maintain the consistency of clean classes.

4.4 Enhanced virtual types and delegation

Although virtual classes and the delegation mechanism available in FamilyJ enables means to express almost everything that has been needed in the implementation of aspect interfaces presented in the last sections, some cases were not contemplated yet. This section presents these cases and how they have been implemented in this thesis.

The next subsections show two problems that have been detected and the solution that have been found to avoid them.

4.4.1 Enclosing object reference problem

Ordinary nested classes in Java have an internal field introduced by the compiler that contains the reference of their enclosing object at runtime. Through this reference, nested objects can access methods of their respective enclosing instances. Since FamilyJ's implementation does not take special care with this case, it implied a wrong behaviour at runtime.

In common Java nested classes the constructors are responsible to initialize this new field. In order to do so, the compiler changes the constructors of nested classes by inserting a new parameter in all constructors of such classes. Hence, when the compiler finds a nested class creation, it inserts the code needed to provide this new parameter to that constructor call. For example an object creation like

```
this.new Mediator();
```

is translated to something like

```
new Mediator(this);
```

and the constructor of the nested class `Mediator` initializes the reference to the enclosing object with `this`.

Since clean and virtual classes use delegation to bind the messages to the right method implementations and they may declare nested virtual classes in their body, special care must be taken in order to virtual class instances call the right methods in their enclosing objects when they are instances of clean or virtual classes.

Using the *outer this* initialization as it has been presented for ordinary nested classes would imply that method calls from virtual class objects to their enclosing reference reached the object in the hierarchy which contains the factory method that has created the virtual class object. Though it is correct when the enclosing object does not use delegation, it could execute the wrong method implementation if the outer object does use delegation.

Figure 4.5 shows how a component created as described in the last sections would have their relations with their enclosing objects. `outerImplementation` is an instance of the enclosing implementation class and `outerBinding` is the enclosing binding object. Together they constitute the outer component that has created the nested component compound by `nestedImplementation`, the implementation object, and `nestedBinding`, the binding instance. Therefore, they all use delegation instead of common inheritance, because their classes have been transformed as discussed in section 4.2

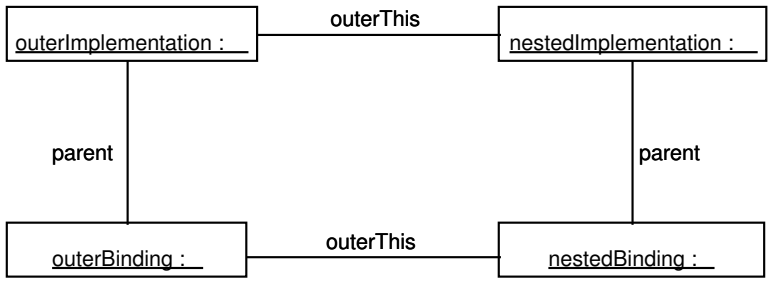

Figure 4.5: Outer reference problem.

The problem that can be seen in this diagram is that as `outerImplementation` and `outerBinding` dispatch their methods through delegation, methods called by `nested-Implementation` and `nestedBinding` to these objects should follow delegation rules. In other words, methods called by `nestedBinding` and `nestedImplementation` to their outer reference, which are declared in the aspect interface of the outer component, should reach before the binding reference, or `outerBinding` in this example. But in this case, methods called by `nestedImplementation` to its enclosing reference will always reach `outerImplementation`.

In order to avoid this problem, the instantiation of virtual classes has been improved

as well as their internal reference to the enclosing objects has been changed. The instantiation of virtual classes are always performed by factory methods that are inserted in their enclosing classes. As factory methods are just common methods in the clean or virtual classes, they are transformed to receive also a reference to the right dispatcher depending of the context that they have been called.

Therefore, as this dispatcher has been created to call the right methods for the context that the virtual object is instantiated, or in other words, as it dispatches the method calls performed by the factory method to the right method implementation, this dispatcher can play the role of the enclosing reference of the new virtual class instance that is being created as well.

Hence, the reference of the dispatcher received in the factory method is the reference passed to the constructor of the nested virtual instance that is created by the factory method. In this way, the constructor of the nested virtual class initializes the virtual class field responsible to be the receiver of method calls from this class to its enclosing object with the reference of such dispatcher.

In order to allow it, the type of the enclosing reference in the nested classes has been changed. Before showing why it has been changed, let's refresh the transformations performed for clean and virtual classes described in chapter 2. Clean and virtual classes are broken up into three new abstractions: an implementation class, a proxy class and a clean interface. The abstraction responsible to enclose the implementation classes created through virtual classes is the implementation class of the enclosing class.

Now that it has been elucidated, the problem can be shown. In a common nested class, the type of the outer reference is the type of the enclosing class of the nested class. In this case it would be the type of the implementation class of the enclosing clean or virtual class that is the real enclosing class of this class. The problem is that the reference that is now passed to the constructor is a reference of the dispatcher that is not of the same type of the real enclosing class. Though the dispatcher has not exactly type of the enclosing class, it implements its clean interface as discussed before in chapter 2.

Thus, the internal reference type of virtual classes has been changed to the clean interface of its enclosing class. After these changes, the runtime structure showed before in figure 4.5 looks like the conceptual collaboration diagram in figure 4.6. Now the nested classes point their reference to the enclosing objects to the dispatcher, and method calls to these dispatchers reach the right object in the runtime hierarchy.

Though this solution deals well with the problem when calling public methods from nested objects to their respective enclosing objects, private methods or fields of the enclosing class cannot be accessed by nested virtual classes anymore. It is because the type of their reference has been changed to the clean interface of their enclosing class, and interfaces cannot define private methods or declare non-static fields.

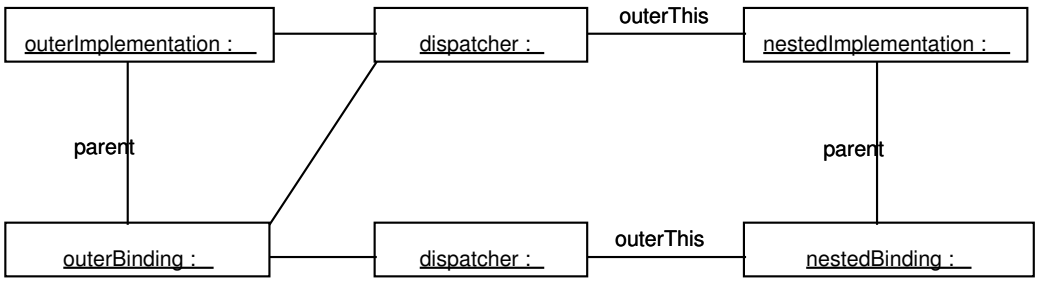

Figure 4.6: The right runtime configuration.

This restriction remains in the current implementation. A possible solution for this problem would be inserting another field in the virtual classes to access public methods as discussed in this section. Leaving then the reference for the enclosing object as it is carried out for common nested classes in Java. Then public methods would be accessed through this new field, and private methods and fields could be invoked using the reference to the real outer object passing the reference contained in this new field to play the role of `self` in such private methods.

Another possible solution would be using the information of the real outer reference contained in the dispatcher set as the enclosing reference of the virtual class. As shown in figure 4.6 the dispatcher maintains this information. The dispatcher keeps this reference because it needs to know how dispatch public methods that are not declared in `outerBinding` for example. Hence this information could be used in order to allow virtual classes to invoke private methods or access fields of their real outer reference.

4.4.2 Parameter types problem

In FamilyJ, methods defined with parameter, whose type is a virtual class that has been overridden in the same scope that the method is being executed, have their bodies changed by the compiler. It introduces downcasts to the right type in that context as first statements of such methods.

For example, in listing 4.11 `Mediator` is overridden to introduce a new method that is not declared in the `MediatorProtocol.Mediator` aspect interface. Now this method should become available to be called in the context of `MediatorProtocolImpl`.

In order to allow it in the method `setMediator(Mediator)`, the parameter `mediator` is renamed and a new local variable is introduced using the old name. This new local variable is initialized by down casting the parameter received to the class `Mediator` overridden in this context. Hence, it allows `setMediator(Mediator)` to call this new

method declared only in the scope of the `MediatorProtocolImpl` class.

```
public providing clean class MediatorProtocolImpl
  extends MediatorProtocol {

  public providing override class Mediator {
    //This method is not declared in the aspect interface
    public boolean isAssignable(Colleague colleague) {
      ...
    }
    ...
  }
  public providing override class Colleague {
    ...
    //The contract of the method was defined in the aspect
        interface
    //then the parameter type is the super type
    public void setMediator(MediatorProtocol.Mediator
       _renamedMediator) {
      //Downcast introduced by the compiler
      MediatorProtocolImpl.Mediator mediator =
        (MediatorProtocolImpl.Mediator) _renamedMediator;

      //Now it can access this method
      if (mediator.isAssignable(this))
        this.mediator = mediator;
    }
    ...
  }
}
```

Listing 4.11: Downcast problem

The problem with this approach is that in some hierarchies, like the hierarchy of the components presented in this chapter it causes a runtime error. For elucidating this problem let's remember how hierarchies of components are created in Caesar. Regarding only nested components, which is where the problem arises, the first object in the hierarchy (that object which receives method calls) is a binding object. The implementation objects are in the hierarchy but always playing the role of parent of some binding. Hence binding objects are the objects available to be passed as parameters because they are the objects available to developers maintain references in the program.

Therefore, the references of nested components passed to methods are always references of binding objects. When executing the downcast in the methods declared in the imple-

mentation classes, it would generate a `ClassCastException` since the parameter passed is not of the implementation type but of the binding type.

Due to this problem a new method is now inserted in virtual and override classes enclosed in clean, virtual or override classes. This new method (`_adapt<TypeName>(Object enclosing)`) looks up to the right object reference in the hierarchy for the context that it gets applied, hence the cast can be performed without raising exceptions.

This method uses the information of the real enclosing reference available in the dispatcher set as outer reference of virtual references as discussed in the end of section 4.4.1. With this information the method can discover the level in the hierarchy that the virtual class is and therefore can return the right object for the context it has been invoked.

In listing 4.12, this new method[3] is inserted in the virtual and override classes. As it can be seen in this listing the method has a different behaviour for virtual and override classes

In the case of virtual objects, they just return themselves, because it is not possible to go further in the hierarchy since it is the last object. On the other hand, when the object is an override reference it has to check if it is the right object to return. In order to do so, it checks if its real enclosing reference is the same as the reference passed (`enclosing`). If it is the same, it means that the method has arrived in the same level in the hierarchy that it has been called, thus it is the right object to be returned, then return itself; otherwise it calls its parent. In summary, the method consists in delegate the method calls until it finds the right object or a virtual object.

Actually the methods in both cases return a dispatcher created for the context that the method is being executed because methods called in this reference must be dispatched following the delegation principals as well, but as it is a conceptual method it has been hidden to simplify their presentation.

With this method, before down casting the parameter inside the method bodies it adapts the parameter for the context of the execution of such method. It can be seen in listing 4.12. In the body of the `setMediator(Mediator)` method, before down casting the parameter `_renamedMediator` as in listing 4.11, it calls the new method in the parameter reference passing its real enclosing reference. Therefore, the problem disappears because the parameters are now also dependent of the context which they get used.

[3]It is a conceptual implementation, details like the method used to get the real reference are not shown.

```
public aspect clean class MediatorProtocol {
  public aspect virtual class Mediator {
    //It is a virtual class, it does
    //not have super with the same type
    public Object _adaptMediator(Object enclosing) {
      return this;
    }
    ....
  }
  ...
}
public providing clean class MediatorProtocolImpl
  extends MediatorProtocol {
  public providing override class Mediator {
    //It is an override class, so first test
    //if it is the right reference for the context
    public Object _adaptMediator(Object enclosing) {
      //MediatorProtocolImpl.this here means the
      //real outer reference
      if (MediatorProtocolImpl.this == enclosing)
        return this;
      //It is not, so try the parent reference
      return parent._adaptMediator(enclosing);
    }
    public void setMediator(MediatorProtocol.Mediator
       _renamedMediator) {
      //Here it first adapt the parameter, after realize
         the down cast
      //MediatorProtocolImpl.this shall be seen as the
         reference to the real
      //enclosing object
      MediatorProtocolImpl.Mediator mediator =
        (MediatorProtocolImpl.Mediator)
          _renamedMediator._adaptMediator(
            MediatorProtocolImpl.this);
      ...
    }
    ....
  }
  ...
}
```

Listing 4.12: Method to adapt parameters to certain context

4.5 Wrapper recycling implementation

Using the wrapper recycling operator is very similar to common method invocation. Syntactically it does not differ from method calls. In the Caesar compiler, the node representation of such expression in the AST is the same as a method call, the only difference is that the invoked method is a constructor instead of an ordinary method.

Therefore, this expression must follow the same scoping rules from Java method calls. In other words, the class of the reference that receives the wrapper recycling call must define a nested class, explicitly or implicitly by inheriting, with that name which activates the call, and this nested class must declare a constructor with parameters that match with the parameters passed.

With those rules, the compiler can use the same approach used by FamilyJ to define factory methods. A method is inserted in the enclosing class of a nested binding class for each constructor declared in that nested class. The only exception is when the constructor has no parameters at all. It is because, it would not make sense to enable wrapper recycling operator for constructors without parameters since it would create always a new wrapper instance that is the semantics of the `new` operator.

In order to allow the semantics of the wrapper recycling, the references of the wrappers need to be maintained by their enclosing objects. It is because the enclosing objects are the responsible to create the wrappers if they have not been created yet through the method inserted on it. For doing that, for each nested binding class is added a correspondent map into its outer class which links the parameters passed to the wrapper recycling operator to the wrapper instances.

Hence the method inserted for the wrapper recycling implementation consists in lookup in the map if the wrapper has already been created for the parameters passed; if it finds the wrapper for the current parameters it just returns this object, otherwise it creates a new wrapper, links it to the parameter references in the map and returns the new object.

Listing 4.13 shows how this method is implemented in Caesar. Note that it returns `Object` instead of the wrapper type. The reason for doing so is the same as discussed for FamilyJ's factory methods. Since the types can be overridden, this method can be overridden as well; if the return type is set with the type declared in this class, it would cause an error when the type is overridden because this method would be overridden too but changing the return type. Therefore, the compiler inserts automatically casts to the right type of the objects that can be known only in the context where the wrapper recycling method is invoked.

Although this method calls factory methods and instances of the right type would be

created even if the wrapper recycling method was not redefined in the subclasses, there would be the problem concerning constructor parameters that can be changed when nested binding classes are overridden. For this reason, for every binding nested class - even if it is not explicitly declared binding but just overrides a binding class - is inserted a wrapper recycling method for each constructor declared in that nested class.

```
public clean binding class MediatorProtocolBinding
  extends MediatorProtocol {
  //The map for mediatorBindings
  private HashMap mediatorBindingMap = new HashMap();
  ...
  //Wrapping recycling operator method
  public Object _getMediatorBindingWrapper(Label label) {
    Object result = mediatorBindingMap.get(label);
    if (result == null) {
      result = _createMediatorBinding(label);
      mediatorBindingMap.set(label, result);
    }
    return result;
  }
  ...
}
```

Listing 4.13: Wrapper recycling method

With this method implementation, the wrapper recycling operator works only for nested binding classes that wrap explicitly a base abstraction and those classes which declare constructors with just one parameter. It is because in these cases there is only one reference to link to the wrapper in the map. But as this operator can be used with any number of parameter a new class, `WrapperKey`, has been inserted in the runtime environment of Caesar. Instances of this class compound an undefined number of objects creating a unique representation of them. Hence, for constructors with more than one parameter, an object of this class is created to link the parameters to the wrapper object.

For example, in listing 4.14 the new binding class `SpecialMediator` has been inserted, so the wrapper recycling method must be added in the class `MediatorProtocolBinding` as well as its respective map. But as this class declares a constructor with two parameters, the `WrapperKey` class must be used in the wrapper recycling method.

As it can be seen in the code in listing 4.14, `WrapperKey` groups all parameters in an `Object` array and now represents the key of all parameters. This key then can be used in the map to link the wrapper instances to the group of parameters passed. Hence when exactly the same parameters are passed one more time to the wrapper recycling method,

it has already their representation in the map, through the instance of `WrapperKey`, and therefore the wrapper is not instantiated again.

```
public clean binding class MediatorProtocolBinding
  extends MediatorProtocol {
  ...
  //It is a binding too since its super class is a binding
  public clean class SpecialMediator
    extends MediatorBinding {
    ...
    public SpecialMediator(Label label, String name) {
      ....
    }
  }
  ...
  //Wrapping recycling operator method of
  //a constructor with more parameters
  public Object _getSpecialMediatorWrapper(
    Label label, String name) {
    //Creates a compound key
    Object key = new WrapperKey(new Object[]{label, name})
       ;
    Object result = mediatorBindingMap.get(key);
    if (result == null) {
      result = _createSpecialMediator(label, name);
      specialMediatorMap.set(key, result);
    }
    return result;
  }
  ...
}
```

Listing 4.14: Wrapper recycling method

Although using this approach the requirements of the wrapper recycling operator are achieved, it can cause problems concerning memory management. It is because wrapper objects remain linked with the base object through the maps after the scope that they were created is no longer valid. For example, if a local variable in a method is assigned with the result of the wrapper recycling operator, the entry in map created to link the wrapper to the base objects remains even after that the method execution ends. With that, objects from the base application that would be able to be collected are no longer eliminated because the entry in the map which links it to its wrapper still point to that object.

A first attempt to resolve this problem was create *weak references* available in Java when using the standard API class `WeakHashMap`. Unlike strong references, weak references do not inhibit collection of the object it points. Hence, this hashtable-based implementation maintains its entries while keys of the entries are in ordinary use. Therefore, when other objects no longer point to the key, the entry in the map is automatically removed allowing the garbage collector to remove the objects.

But maintaining the wrappers in `WeakHashMap` instead of common hashtable-based maps as shown here does not work as desired in both cases (using the base object as key, or a compound key using `WrapperKey`).

The first case where the base objects become keys in the maps seems to be the exact application of weak references. But if a closer look is taken in the binding implementations, the problem arises.

The most common implementations of binding classes keep a reference to the base object that it wraps, often using the `wraps` facility available in the language. With that, this reference disenables the entry in the map to be removed since there is still this strong reference to the key, or wrappee, object.

In the second case using the compound object as key is even worse. The new object created to be the key is pointed by a strong reference only in the context of the wrapper recycling method. When the execution of this method ends, the entry is able to be removed from the map and the semantics of the wrapper recycling operator is no longer achieved.

Duo to this problem, a common map is used (`HashMap`) and the new operator discussed before (`~ <NestedBindingType>(...)`) has been introduced in the language to allow manually detaching of the wrapper with the base object. Note that it is not necessary to be used. The base objects will be collected when the wrappers can be discarded. In other words, when the outer object that contains the maps can be removed. However, if developers desire to control the memory, they can use this new operator for detaching base objects from their respective wrappers.

The implementation of this operator is very similar to the wrapper recycling operator. For each wrapper recycling method added in the binding classes, it is inserted a method (`_remove<NestedBindingType>Wrapper(...)`) to remove the entries in the map using the keys as presented in this section. Thus, like the wrapper recycling operator, this operator is replaced to an ordinary method call that reaches this new method.

Although this new operator has been inserted to deal with memory management problems, it brings more control to developers when using wrappers in their applications. With this new operator, it can be decided if the wrapper should be recreated for certain object that may be useful for some kinds of application.

4.6 Implementation details

This section gives a very brief overview of the implementation of the compiler realized in this thesis. It shows only the main classes introduced in the compiler as well as some classes that have been changed during the implementation.

The Caesar compiler has been implemented on top of the open source Java compiler KOPI [1]. Therefore all abstractions introduced in the compiler somehow extend the abstractions implemented in KOPI. As introduced before, the concepts presented in this thesis have been realized in an extension of the work carried out in context of Andreas Wittmann's thesis [16] and Jürgen Hallpap's thesis [4]. However the code structure followed in this work is that defined by Hallpap.

In order to introduce the new language support presented in chapter 3, the scanner and parser had to be changed. These changes have been done in the files *Caesar.t*, *Caesar.g* and *CaesarMessages.msg*. In the file *Caesar.t* the new key words have been inserted. *Caesar.g* is the definition of the grammar, so all rules needed to allow the concepts presented in this thesis in the language have been introduced in this file. The other file *CaesarMessages.msg* is the file that contains the error messages generated by the compiler, so the new error messages introduced in the compiler are in this file.

After having the language accepting the new abstractions and operators, the source code of the compiler have been modified. These changes have been realized to allow the representation of those abstractions and operators in the AST, as well as introduce semantic analysis for them. Besides that, new *visitors* have been introduced or changed, these visitors are the classes responsible to traverse the AST performing some modifications, transformations or checking in the elements of the AST.

Inn the following, the main classes introduced in the compiler during this thesis, as well as the main FamilyJ's classes that have been modified, are presented with a brief discussion of their roles:

- `org.caesarj.compiler.util.AspectInterfaceTransformation`: This class has been introduced to perform the transformations discussed in section 4.2, as well as to insert the internal structure of explicitly declared wrappers. It is a visitor which traverses the AST transforming the abstractions that are needed.

- `org.caesarj.compiler.util.MethodTransformationFjVisitor`: It is another visitor built in the FamilyJ's implementation. The constructor's transformations presented in section 4.3 are carried out in this class. Besides that, this class is responsible to insert the wrapper recycling structure discussed in section 4.5.

- `org.caesarj.compiler.ast.CaiClassDeclaration`: This class represents a class

declaration in the AST that may be an implementation or binding class. It has been introduced on top of FamilyJ's hierarchy of class declarations.

- `org.caesarj.compiler.ast.CaiWeaveletClassDeclaration`: This class represents a weavelet class in the AST.
- `org.caesarj.compiler.ast.FjCleanClassDeclaration`: This class has been created in the FamilyJ's project, but it had to be adapted to allow binding and implementation classes.
- `org.caesarj.compiler.ast.CaiInterfaceDeclaration`: This class has been introduced to allow interfaces to be declared as aspect interfaces.
- `org.caesarj.compiler.ast.CaiWrappeeExpression`: Objects of this class are created in the AST when a `wrappee` expression is found in the source code
- `org.caesarj.compiler.ast.FjMethodCallExpression`: It is a FamilyJ's class that has been adjust for the implementation of the wrapper recycling and wrapper destructor operators.

This is the main files and classes that have been introduced or changed in the compiler. For more details, please refer to the source code documentation that has been realized using Javadoc in the points of the code that have been modified, or in those classes that have been inserted in this implementation.

Chapter 5

Summary and future work

This chapter gives an overview of the concepts presented in this thesis. Besides that it makes an evaluation of the implementation of the model showing how it avoids the problems discussed in chapter 1. In addition to that, it presents what could be carried out to improve the current implementation.

During this thesis, it has been presented so far new language support to facilitate the description of generic functionalities of applications as independent components that become reusable and extendible achieving then goals of modularity.

In addition to that, new support has been described in order to integrate these new components into base applications without changing their modular structure. It permits the component implementations to be decoupled from application's particularities, thereby allowing them to be applied into many applications or even into many abstractions in the base code.

The key concept presented here is the notion of aspect interfaces. These new abstractions allow describing the type of components not only by representing the contract provided by the component implementation as common interfaces do, but they can also express what implementations of the components expect from of the applications that they get integrated into. In other words, these interfaces permit separating the expected and provided contract of the component.

An important feature of aspect interfaces is that they can be nested. This feature gives means to describe component types as a set of abstractions that together define the functionality as a whole.

This new interface leads to two new abstractions that are responsible to provide implementation for it: implementation and binding classes. Implementation classes realize the generic parts of the functionality, in which binding classes integrate into a particular

application.

Although implementation classes are totally decoupled from the application, they can communicate with the base objects of the application that they get integrate into. It is realized through the binding classes using the clearly defined interface described in the aspect interface that they implement.

On the other hand, binding classes are also decoupled from a specific implementation class and as well as implementation classes can communicate with the implementation of the component by using the aspect interface.

With binding and implementation classes decoupled from each other these classes can be freely combined by means of the new abstraction provided by Caesar: the weavelet classes. These classes then join a binding and an implementation class building up a complete implementation of the aspect interface.

The mappings of the component to the base application are carried out through the binding classes. These classes may be just a wrapper of a base abstraction, however they may also be mapped to a concept of the base application that not necessarily is an abstraction in such application.

In order to use wrappers without losing their internal state and identity, the wrapper recycling operator has been presented. This operator avoids the creation of wrappers of base objects that have been already wrapped once allowing hence wrappers to maintain state and identity. Besides that, base objects can be detached from their respective wrappers with the wrapper destructor operator. This operator permits hence the creation of new wrappers for the same base object.

The wrapper recycling operator combined with binding classes and the dynamic JPI presented in chapter 2 enable the application of on demand remodularization presented in chapter 1. Using the JPI model available in Caesar, binding classes can declare *joinpoints* that intercept the execution of the base application, becoming hence *aspects* as they are known from AspectJ for example.

Using that, binding classes can introduce the behaviour contained in the component into the application using these joinpoints for intercepting the execution of the base code that shall be *decorated* with the component behaviour. In this way, the introduction of the component into the application would be applied just in the code surrounded by a `deploy()` block where the binding instance would be applied. In addition to that, by using the wrapper recycling operator to create the binding instance, this new aspect (binding instance) can maintain state and identity.

Evaluating the implementation of the model presented in this work, it covers the problems identified in the AspectJ's solution described in chapter 1.

It enables support to express the expected contract of the component through aspect interfaces, thereby avoiding the problems that become in languages like AspectJ that provide only little assistance to express the expected contract.

Unlike in AspectJ, or even in Object Teams and ACC presented in chapter 1, binding classes are totally decoupled from any implementation of the component. They are linked to an specific implementation just by the common aspect interface. It increases reusability of the abstractions because as well as implementation classes, binding classes become reusable.

Binding classes *wrap* base objects rather than physically change their structure as in the AspectJ solution. Meaning that the model does not have the invasive nature presented by AspectJ, therefore it does not suffer of the problems identified in the AspectJ solution when it injects a new type in the modular structure of the base application.

Though binding classes may be directly mapped to one abstraction of the base code using the `wraps` clause, it is not restrictive. Through binding classes, components can be integrated into applications even if such application does not define any abstraction to play certain role of the component. With that, it provides the sophisticated mappings that are not possible in the other models presented, AspectJ, Object Teams and ACC.

Though the implementation of the model presented improves the expressiveness of the language technology to facilitate the separation of an application into reusable modules and avoids the problems outlined in chapter 1, it is an ongoing work and there is work to be done yet.

A point that could be tackled is the class hierarchies of implementation and binding classes. As discussed in chapters 3 and 4, classes that declare the `binds` or `provides` clause cannot have a subclass relationship with other classes.

Allowing this subclass relationship would increase reusability and extendibility in the language. As aspect interfaces can be defined in a hierarchy of interfaces, when implementing the contract of a sub aspect interface, the contract of the super aspect interface must be provided as well. Hence, with this relationship, binding or implementation classes that implement the contract of a sub aspect interface could reuse possible implementations of the super aspect interface, by extending such classes.

Another concept that could be improved in the current implementation of the compiler is the wrapper recycling operator. First by implementing the notions of *most specific wrappers* introduced in [10] that is not available in this implementation. With this kind of wrapper recycling operator the wrapper to be created is chosen based on the runtime type of the parameters passed to the operator. This operator would be valid just when nested bindings are declared with the same name, which is another point that is not allowed in the current version and could therefore be another future work.

Concerning the wrapper recycling operator yet, the memory management of wrappers could be improved. As described in chapter 4, the memory management of wrappers is not a simple task and the current implementation has a simple implementation of that because of the problems described in such chapter. Maybe a possible solution could be extending the implementation of `WeakHashMap` allowing that whenever the base object are released to garbage collection by the base application its correspondent entry in the map that link it to a wrapper is removed, releasing it also to be collected. On the other hand maintaining it even if the key is not directly pointed by a strong reference and the base objects cannot be collected yet.

These are some future works concerning the implementation of the model, obviously the model is also an ongoing work, therefore it is being improved as well, but as this thesis talk about the implementation of the model rather than the model itself, these future works are not presented here. For further information about that please refer to [10] and [9].

Bibliography

[1] DMS Decision Management Systems GmbH. The KOPI Project. http://www.dms.at/kopi/index.html, 2003.

[2] Erik Ernst. Family polymorphism. In *Proceedings of ECOOP 2001 – Object-Oriented Programming*, pages 303–326. Springer-Verlag, 2001.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[4] Jürgen Hallpap. Towards Caesar: On Dynamic Deployment and Aspectual Polymorphism. Master's thesis, Technische Universität Darmstadt, Germany, 2003.

[5] Jan Hannemann and Gregor Kiczales. Aspect-oriented design pattern implementations. http://www.cs.ubc.ca/~jan/AODPs/, 2002.

[6] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173, 2002.

[7] Karl Lieberherr, David Lorenz, and Johan Ovlinger. Aspectual collaborations: Combining modules and aspects. *Computer Journal of the British Computer Society*, 2003. to appear.

[8] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 edition, 1997.

[9] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[10] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD '03), Boston, USA*, 2003.

[11] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[12] Harold Ossher and Peri Tarr. The need for on-demand remodularization. Position paper in ECOOP, 2000.

[13] Johan Ovlinger. From aspect-oriented model to implementation. watch out for impedance mismatch. Position paper in AOSD, 2003.

[14] The AspectJ Project. http://www.eclipse.org/aspectj, 2003.

[15] Object Teams. http://objectteams.org/, 2003.

[16] Andreas Wittmann. Towards Caesar: Family Polymorphism for Java. Master's thesis, Technische Universität Darmstadt, Germany, 2003.