Vrije Universiteit Brussel – Belgium

Faculty of Sciences

In Collaboration with Ecole des Mines de Nantes – France

2005

A wrapper API for Reflex

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: Richa GUPTA

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoter: Jacques Noye (Ecole des Mines de Nantes)

ABSTRACT

Reflex is a versatile kernel for multi-language Aspect-Oriented Programming (AOP) which also provides building blocks to help implement different AO languages. The objective of this AOP kernel is to support and combine a large range of Aspect approaches. Reflex has mainly focused on ‘behavior-oriented’ aspect languages. In order to verify the versatility of AO approaches, it is also necessary to consider aspect languages which are interested in the structure of aspects, such as Caesar. Caesar has focused on the issue of structuring aspects and the relationship between this structure and the structure of the base program. Thus Caesar gives the structural point of view. It is a language which uses the concept of wrappers and collaboration to achieve binding between aspects and components. Wrappers are not directly supported by Reflex. This thesis exposes the development of a wrapper API which was integrated with Reflex in order to extend the Reflex API so that wrappers can be directly supported. An example was also developed to show how this wrapper API can be used to develop an aspect and component language.

ACKNOWLEDGEMENT

First and foremost I would like to thank GOD for always being there for me and guiding me whenever I wanted HIS guidance.

Second, I would like to thank my grand parents for their blessings and good wishes.

A very big thanks to my parents and my brother, specially my parents for giving me this opportunity to come here and study. It would not have been possible to come here without their help. They were always a constant source of inspiration for me. Although I am so far away from them, they were always there for me in difficult as well as happy times. I would not have been able to enjoy the wonderful time I spent in EMOOSE if they were not there for me. I love them a lot.

I would also like to extend my sincere thanks to all the professors who came to teach us and share with us their knowledge and experience. A special thanks to my advisor, Jacques Noye, who was always ready to help me out whenever I needed his advice and guidance. And also a big thanks to Annya who was like a mother to us and also Sylvie who was always ready to help us out.

And last but not the least, a very special thanks to my classmates, Javier, Jorge, Daniel and Harmin who were always there to boost my morale whenever I felt down and out and who were always ready to lend me a helping hand.

CONTENTS

Chapter 1. INTRODUCTION

Aspect-Oriented Programming (AOP) [4] was introduced by Gregor Kiczales et al. in the year 1997. It is a programming technique developed to address *crosscutting concerns*. These concerns are modularized into modular units called *aspects*.

Since the conception of AOP, many different approaches to AOP, together with the corresponding aspect languages, have been developed which are worth exploring and experimenting with. Each aspect language is developed differently using different tools. This means that aspects are developed using different tools. When different aspects affect the same program points, they interact [39]. But when aspects are developed using different tools, then it is not easy to resolve which aspect should be applied first because tools are incompatible with each other and the interaction between aspects is blindly handled [25]. Hence the resulting semantics of the program depend on the order in which these tools are applied. Furthermore, tools to implement a specific model of AOP have to be developed from scratch. This gave rise to the following two basic needs:

1. having a common platform on which different AOP approaches can be applied and experimented with and which also

2. provides building blocks for the implementation of different AO languages using different AO approaches.

Reflex was developed to fulfill the above mentioned needs. Reflex is a versatile kernel for multi-language AOP which also provides building blocks to help implement different AO languages. The objective of this AOP kernel is to support and combine a large range of Aspect approaches. Reflex has mainly focused on *behavior-oriented* aspect languages. In order to verify the versatility of AO approaches, it is also necessary to consider aspect languages which are interested in the *structure* of the aspects. One such language is Caesar [3].

Caesar is an interesting language which is worth exploring. It has focused on the issue of structuring aspects and the relationship between this structure and the structure of the base program. Thus Caesar gives the structural point of view. Whereas AspectJ, which is the de facto standard for AOP languages has focused on aspects as modifying the behavior of the base program, that is, it gives the behavioral point of view. Caesar language is based on the following two main ideas:

1. Wrappers – Wrappers in Caesar are a mid-level abstraction.
2. Collaborations – Collaborations in Caesar come into play when the aspect has to be bound to the base program.

Wrappers are a well established technology in the Object-Oriented Paradigm. Many patterns have been developed using wrappers. People have successfully implemented wrappers in Smalltalk [10]. Dynamic wrappers are implemented in JAC (Java Aspects Components) framework [7] to handle the composition issue. Dynamic wrappers are implemented with the help of a dedicated MetaObject Protocol (MOP) wherein calls to wrappee are trapped by the wrappers. Component wrapping also is a well known structuring technique [29]. Wrappers have been used for integrating Off-the-Shelf (OTS) components with the rest of the system. Component wrapping has also been used for wrapping COTS components so that the

interaction between the component and the rest of the system can be understood [31]. Now, the concept of wrappers has been used to develop aspect languages like Caesar [3]. Wrappers are a powerful mechanism for introducing new functionality without changing the original source code. Hence the idea of wrappers is worth experimenting with.

Another idea which is worth exploring is the notion of *collaboration*. Many languages have been developed on the notion of collaboration. Component language like Java Layers [5] talks about layers (or components) for building new types and Jiazzi [32] talks about units and compound units. These units and layers are implementations of a collaboration. The composition between units and layers are implemented using mixin-based inheritance. Aspectual Collaboration [16] is a language for integrating aspects and modules. But none of the above mentioned languages really integrates Aspects and Components. Such a language is yet to be developed. Hence the idea of collaboration can be exploited still more to develop such a language.

This suggests that it is worth experimenting with the notion of wrappers and collaboration to build a language which integrates aspects and components and use Reflex as a platform for these experiments.

This gives rise to the objectives of my thesis.

Objective

The objective of my Master thesis is twofold:

1. Build a wrapper API for integrating aspects and components using the notion of collaboration.

2. Bridge the gap between Reflex and available AO approaches by integrating a Wrapper API in Reflex. This API will act as a building block for developing new aspect- and component-oriented languages based on the notion of wrappers.

Contribution

1. A wrapper API integrated with Reflex. This API supports:

    a. The creation of specialized wrapper classes for wrapping components or Java classes.

    b. Inheritance and forwarding techniques for binding of aspect and components using wrappers.

    c. Methods for modifying / changing the behavior of a wrappee.

    d. Methods for inserting a method and an instance variable.

2. Examples to show how this API can be used, together with annotations, to build aspect- and component-oriented languages.

3. This work has also tested the Reflex API for structural transformation and suggested some improvements.

Thesis Structure

The structure of the thesis is as follows:

Chapter 2 is the state of the art which is dedicated to the necessary background concepts regarding aspect-oriented programming, Reflex, components, and wrappers. Chapter 3 gives technical concepts which are necessary to understand some terminology used in the thesis. Chapter 4 gives an idea of how wrapper classes are generated and how binding is achieved between aspects and components to accomplish collaboration. Examples are also given in this chapter to show how wrappers can be put to different uses. Chapter 5 takes up the issue of integration of the wrapper API with Reflex. It also shows with the help of an example how this API can be used to achieve collaboration between components using the concept of wrappers. Chapter 6 is a conclusion showing the contribution of the thesis and pointing to future work.

Chapter 2. STATE OF THE ART

2.1 Overview of Aspect-Oriented Programming

Aspect-oriented programming is a new programming technique developed to address crosscutting concerns. AOP helps to modularize these concerns by introducing a*spects*. Aspects encapsulate behaviors that affect multiple classes into reusable modules. Here is a small example to concretize the main idea behind AOP.
Suppose there is a banking application that implements the logging functionality to satisfy transaction. The code to trigger this transaction functionality is scattered throughout various modules. In the object-oriented paradigm, a module is a class. In other words, the code for logging intertwines with code whose primary responsibility is something else. In this case it is Banking. This type of behavior, wherein the code is scattered through various classes, is termed *crosscutting concerns*. Thus a change in the logging functionality will have a ripple effect in all the modules which calls this logging functionality. AOP was developed to overcome these crosscutting concerns. These concerns are modularized into units called *aspects*, and hence the name *Aspect-Oriented Programming*.
The AOP-based implementation of an application consists of: (i a) a component language with which to program the components (i b) one or more aspect languages with which to program the aspects (ii) an aspect weaver for the combined languages (iii a) a component program, that implements the components using the component language and (iii b) one or more aspect programs that implement the aspects using the aspect languages [4]

2.1.1 Examples of AOP Languages and Approaches

AspectJ

AspectJ is a simple and practical aspect-oriented extension to Java. Although there were a number of AOP languages developed before AspectJ, but all of them were domain specific. AspectJ was the first AOP language which is a general-purpose language. Hence it has become a de facto standard for all AOP languages and sets the standard for other frameworks to follow.
AspectJ uses a new unit of modularity called *aspects* to implement crosscutting concerns modularly. It supports two kinds of crosscutting implementations – *behavioral crosscutting* which helps define additional behavior to run at certain well defined points, and *structural crosscutting* which helps modify the static structure of a program. Well-defined points in the execution of the program where program behavior is extended by introducing new crosscutting behavior are called *join points.* The static counter part of these join points found in the code is called the join point *shadow*. A *pointcut* is a group of join points which is used to specify the places at which the crosscutting concern or the aspect actually affects the base application. The crosscutting behavior that should be applied upon occurrences of join points matched by a given pointcut definition is called an *advice* [2]**.** Advices are method-like constructs defining the additional behavior to be executed at particular join points. An advice has to be explicitly bound to a pointcut. Advices are of five kinds – `before` (executed

before the execution of the join point), `after` (executed after the execution of the join point), `after throwing` (executed after the join point execution, returning with an exception), `after returning` (executed after the join point execution, returning normally) and `around` (replace the join point execution).

Caesar

This AOP model and language was proposed by Mira Mezini and Klaus Ostermann. Cesar puts forward two ideas – expressing aspects as a set of collaborating abstractions and then structuring the interaction between the two parts of an aspect: aspect implementation and aspect binding into a particular code base [3]. This model was developed because according to the authors, the existing model of AspectJ is not sufficient for modular structuring of aspects. The basic feature of Caesar is *Aspect Collaboration Interfaces (ACIs)*. An ACI is basically an interface which contains mutually recursive nested interfaces. Each nested interface represents an abstraction. Inside those interfaces, methods are declared with the keywords *provided* and *expected*. Methods declared with *provided* are methods that the aspect provides in the context in which it is applied and *expected* are methods that the aspect expects from the context in which it will be applied. The a*spect implementation* implements all the methods declared in the interface with the provided facet and the *aspect binding* implements all the expected methods defined in the nested interfaces. New classes called *Weavelets* are also defined. They relate the implementation of an ACI with an aspect binding.

Event-Based Aspect Oriented Programming

The developers of EAOP are interested in defining aspects as a general structuring mechanism which is applicable to a large variety of crosscutting concerns. The main characteristics of EAOP are that aspects are defined in terms of events emitted during program execution, c*rosscuts* relate sequences of events and once a crosscut has been matched, an associated *action* is executed.

Comparison with other AOP models

Explicit Aspect Composition – Aspect composition operators are supported by EAOP which helps in ordering and re-ordering of aspects. Thus aspects can be prioritized.

Aspects of Aspects – One aspect can be applied to another aspect by simply allowing the former aspect to refer to runtime events generated by the second one. In order to ensure the correct application of aspects in presence of aspects defined on top of other aspects, aspect composition operators can be used.

Aspect Instantiation – In EAOP, aspects can be instantiated just like any other ordinary classes and provide a corresponding per-aspect state.

Summary

AspectJ was developed as a basis for an empirical assessment of aspect-oriented programming. Hence AspectJ became the standard framework for the development of other AOP languages. Caesar was developed for better structuring of aspects which was lacking in AspectJ. Lacking support for multi-abstraction aspects, sophisticated mapping, reusable aspect bindings and aspectual polymorphism were some of the drawbacks in AspectJ which was overcome in Caesar. EAOP model was developed for better generalization of crosscuts in terms of sequence of events. In AspectJ, crosscut was defined as a specific point but in EAOP crosscuts are defined in terms of sequences of runtime events.

Here are just a few examples of different AOP languages and approaches. Each approach has its own set of pros and cons. The variety of approaches was the motivation factor for the development of the versatile kernel in Reflex which supports the development of different AOP languages based on different AOP approaches.

2.2 Reflex

2.2.2 Initiation

The Reflex project was started in 2001 with the objective of applying reflective extension of Java to enhance mobile agent systems with regards to the way the resources attached to mobile agent are handled upon migration. But none of the reflective extensions available could satisfy the need of portability and the ability to attach a metaobject to only some specific instances of a given class. Furthermore, the high-level reflective extensions provide hard-wired choices about MOP definition and hook introduction. This gave rise to the need of developing an extension which would be able to define a generic MOP. Thus Reflex, an *open reflective extension of Java* was conceived [21]. The main ideas of Reflex are:

1. definition of a generic MOP.
2. reification of the code transformation process as an extensible entity.

To put generic MOP into practice, hooks must be inserted where needed. The corresponding code transformation is reified as an extensible entity, which are called class builders. These class builders set up appropriate hooks within a given class. Subclassing is used when the class cannot be modified.

Reflex is built on top of Javassist [34].

2.2.2 Reflex and Partial Behavioral Reflection

This open extension of Java for behavioral reflection evolved into a model to support *partial behavioral reflection*. Partial behavioral reflection is an approach to more efficient and flexible behavioral reflection relying on the notion of avoiding useless reifications [25]**.** Flexibility is accounted for by the fact that the selection of *what* has to be reified can be done

and efficiency is accounted for by the fact that reifications can be turned off or on depending on the reflective needs of the object. Thus one can select *when* reifications are to be applied. The evolution from open extension of Java for behavioral reflection to partial behavioral reflection took place due to some of the following limitations in the previous model.

1. The idea of generic MOP, though appealing was not practical because it was too permissive and inefficient and most of the time would have required a MOP specialization. Thus the existing class builders would have to be extended and new ones created.
2. The existing model of class builders was too coarse grained to be really appropriate for user extensibility.

2.2.3 Reflex and AOP

Kojarski et al. [33] stated "Aspect-Oriented programming (AOP) is in essence a computational reflection mechanism. The join point model reflects a program's behavior: a join point provides the ability to introspect; advice provides the intercession (manipulation) capability". This means that everything in AOP precipitates down to semantic alterations of the application written in base language. The reflective model for structural and behavioral alteration can be used for describing semantic alterations. Thus there are clear connections between reflection and AOP.

Furthermore, there are many different approaches to AOP which are available which should be explored and experimented with but each aspect language is developed differently using different tools. This means that aspects are developed using different tools. When different aspects affect the same program points, they interact. But when aspects are developed using different tools, then it is not easy to resolve which aspect should be applied first because tools are incompatible with each other and the interaction between aspects is blindly handled [25]. Thus aspects will be handled in an unordered way. This gives rise to the need of having a common platform on which different AOP approaches can be applied and experimented with.

The previous model of Reflex handled partial behavior reflection and this new idea of a common platform gave rise to a Versatile AOP Kernel. The foundation of this kernel was based on the idea that the transformation process of the base program is very similar in all the AOP systems. This transformation can be factored out into a versatile Kernel which can then support various AOP languages based on various approaches.

2.2.4 Present Scenario

Reflex is now a versatile kernel for multi-language AOP. It provides building blocks for the implementation of different AO languages. This facility helps in experimenting with various AOP concepts and languages. Reflex also supports the detection and resolution of interactions between aspects written in different languages

The architecture of the AOP kernel consists of three layers as shown in Fig 3.1. [22]**.**

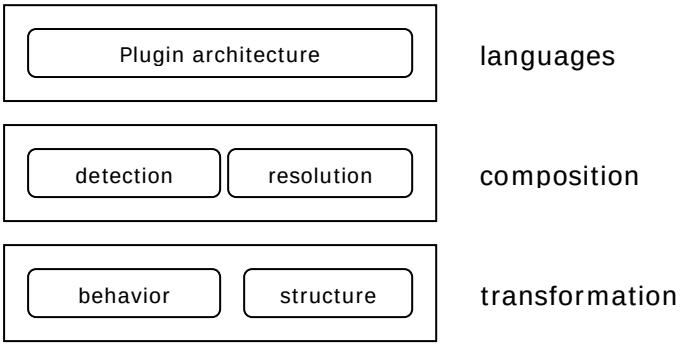

Fig-3.1 Architecture of an AOP Kernel

Reflex relies on the notion of an explicit link binding a cut to an action. A cut determines where an aspect has to be applied and an action specifies the effect of the aspect. Links are a mid-level abstractions, in between high-level aspects and low-level code transformation [22].

The *transformation layer* is in charge of basic weaving, supporting both structural and behavioral modifications of the base program

The *composition layer* is responsible for the detection and resolution of interactions between aspects

The *language layer* is responsible for modular definitions of aspect languages.

There are two types of links:

1. A *Structural Link* (S-link) binds a structural cut to the related action

2. A *Behavioral Link* (B-link) binds the behavioral cut to the related action. B-links are associated to hooks that are inserted in the program at appropriate places to delegate control to the metaobject.

S-links are setup at load time and B-links are applied at runtime.

Summary

Reflex provides a platform for implementing various AO approaches. It provides building blocks for developing new AO languages. Hence the API has to be as complete as possible to enable it to support a large range of aspect languages. It contains quite a rich API for structural and behavioral modifications.

So far experiments with Reflex and AOP have mainly focused on “behavior-oriented” aspect languages. In order to verify the versatility of AO approaches, it is also necessary to consider aspect languages which are interested in the structure of the aspects, like Caesar. Considering Caesar, wrappers look like a mid-level abstraction, which is not directly supported by Reflex.

2.3 Wrappers

Wrappers are a type of software "glueware" that are used to attach together other software components. They are mechanisms for extending the behavior of the object that is wrapped. This is done by introducing new behavior which is executed before, after or in place of the existing methods.

One of the first papers published was [9] in which wrappers were introduced as encapsulators. This paper talks with reference to Smalltalk-80. In order to encapsulate an object, one had to send the encapsulator class the message *object:* with the name of the object to be encapsulated as an argument. When a message is sent to the encapsulated object, a pre action is performed before the object's method is executed and then a post action is performed before the result is received.

[10] examines several approaches, again with reference to Smalltalk, which help in intercepting and augmenting the behavior of existing methods in order to 'wrap' new behavior around them. A summary of these approaches are as given below:
Source code modification in which the new code is inserted directly in the original method's source and then the resulting code is re-compiled. The other method is *byte code modification* wherein the byte codes are directly inserted into the compiled method. This removes the need for re-compiling the method. In the *new selectors* approach, the original method is moved to a new selector, a new method is created that executes the before code, sends the new selector and then executes the after code. Another approach is *class wrapper* wrapping in which subclassing is used to specialize the behavior. A new method is created which executes the before code, calls the original method using super and then executes the after code. In *instance wrapper* wrapping approach, method wrapping is done on a per instance basis.

Dynamic wrappers form a key feature of JAC (Java Aspect Components) [19]. This dynamic wrapper is a stand-alone object that defines behavior to extend the behavior of regular objects. Factoring out advice from base object or aspect class into these stand-alone objects allows performing dynamic weaving because advices can be added and removed at run time. Also, with dynamic wrapper, the wrapper does not change the wrappee reference as is required to be done with classical wrappers using forwarding technique. The simplest way of implementing dynamic wrappers is to use a Meta-Object Protocol (MOP). The JAC framework uses a dedicated MOP for supporting dynamic wrappers which helps in trapping calls to the wrappee by the wrappers. Dynamic wrappers have also been used to solve the composition issues [7] namely compatibility issue, ordering issue and optimization issue.

Nowadays wrappers are a well established technology in the object-oriented paradigm. Many patterns have been developed using wrappers and wrappers have been put to use in many areas. The main attraction of a wrapper is that the behavior of an object can be changed without changing its source code. Hence wrappers are particularly used in component-based software engineering.

In [28], wrappers are used for detecting interface violations among components. Stephen H. Edwards et al. have developed a framework wherein instead of hard wiring the code for checking violations of interface contracts into the base level component, it is layered on top of the component so that it can be turned off or on selectively for one or more components. The idea behind this framework is that the component is encased in a wrapper whose sole

functionality is to implement violation detection checks. In order to detect these violation checks, the wrapper performs run-time checks both before and after each method invocation.

The concept of wrappers has also been applied for wrapping Off-the-Shelf (OTS) components. The need for wrapping components stems from the fact that it is very expensive or difficult to change the components. So in order to change the functionality of the component, it is easier and less expensive to add new features to the component by incorporating them into wrappers. In [29], a general approach for developing protective wrappers has been given which helps integrate OTS items with the rest of the system without reducing the system dependability.

Summary

Wrappers have been exploited to quite a great extent. The main attraction for the use of wrappers is that the wrappee behavior can be modified or changed without having to change the source code. Wrappers have been introduced in component based software engineering also. They have been used for integrating Off-The-shelf components with the rest of the system and for achieving collaboration between components. This idea of wrappers can be exploited further for developing an integrated aspect and component language, based on the notion of collaboration, where wrappers play an important part in binding of aspects and components.

2.4 Overview of Components

2.4.1 Definition of a Component

A formal definition of a component is still under discussion. However a component can be defined as “a reusable aggregation of functionality with a well-defined interface”. According to [FeRT1999, S.34] the term component is defined as follows:
A component consists of different (software) artifacts. It is reusable, self-contained and marketable, provides services through well-defined interfaces, hides its implementation and can be deployed in configurations unknown at the time of development [38]

A component is generally developed for a specific domain like a business domain. At the time of development of a component, a component developer does not know how the components will be deployed in the application. Hence she / he do not have control over the deployment and execution environment. Exact prediction of the development of component-based software is a difficult task. A number of concerns must be realized as separate software components. It is often difficult to encapsulate these concerns using traditional modularization techniques. In [34] several approaches for overcoming these inadequacies of existing modularization techniques have been proposed. They are meta-level architectures, aspect-oriented programming, multidimensional separation of concerns and component containers.

2.4.2 Component Models

There are 3 main component models available today. They are:

1. OMG's CORBA Component Model (CCM).
2. Microsoft's (D)COM / COM+ family.
3. SUN Microsystem's JavaBeans and Enterprise JavaBeans.

Elements of Component Models

1. *Interfaces* – An interface serves as a contract between a component and its clients. It specifies the services that a client can request from a component and the component has to provide an implementation of these services.

2. *Metadata* – It stores the information about interfaces, components and their relationships. This information provides the basis for scripting and remote method invocation and is used by composition tools and reflective programs.

3. *Interoperability* – A software component composition is possible only if components from different vendors can be connected and are able to exchange data and share control through well defined communication channels.

4. *Customization* – It is the ability of a consumer to adapt a component prior to installation or use.

5. *Composition* – The component model must define how to design interfaces to support such composition.

6. *Evolution support* – A new version may not only have a different implementation but may provide modified or new interfaces. Existing clients should be affected as little as possible.

7. *Packaging and Deployment* – A component model must describe how components are packaged, so they can be independently deployed, that is installed and configured, in a component infrastructure.

2.4.3 Component Containers

In this approach, in order to overcome the problem of separation of concerns, the appropriate implementations are embedded and executed as part of the architecture itself. Thus when an instance of the component is created by the component container, all the logical environment is automatically embodied in the component just because the component was implemented inside the container. In other words, a component container is just like a wrapper but is implemented at the architectural level. The basic idea behind the component container approach is that the infrastructural services are associated with the components at deployment time rather than explicitly addressing them in component implementation at design time.

Implementation of a Container (w.r.t EJB)

When a client calls a method on the component interface, the call is replaced by the *proxy* (a kind of wrapper) object provided by the container. All the calls for the component are now intercepted by the container that hosts that component and additional behavior that has to be performed is applied. The deployment descriptor is used for associating a bean or component with infrastructural services.

2.4.4 Aspects vs. Containers for Separation of Concerns [17]

1. Container provides a standard set of services. As such it is difficult to extend a container with additional services. Aspects, on the other hand allow us to easily add infrastructural services to base object. Thus, in the case of containers, we can only use those services that are provided by the container. Using aspects, one can easily customize or add services.
2. As far as reusability of containers is concerned, it is much stronger as compared to reusability of aspects. Containers can be easily re-used in different applications for different platforms with eventually different implementations of infrastructural services. If authorization checks are required for different components, then all that needs to be done is to deploy the two components along with their deployment descriptor. The container then automatically associates the proper services. In the case of aspects, it is a little more difficult. In order to support reusability, abstract aspects with abstract pointcuts can be provided. These abstract aspects can be extended according to the needs. But this poses some drawbacks. First, in order to extend an abstract aspect, the source code should be easily available, easy to understand and modify. Second, using concrete aspects as connectors may easily lead to a complex and bloated aspect hierarchy which is hard to understand, extend and maintain. Third, if aspect instance states are defined, then it is difficult to apply abstract aspects.

Chapter 3. BACKGROUND INFORMATION

3.1 Javassist

Java programming language supports reflection. But this reflection is restricted to introspection which means that the data structures used in a program like classes can only be introspected. The ability to change the behavior of a program is very limited. Most of the extension proposed to address this limitation of JAVA enable only behavioral reflection which is the ability to intercept an operation and change the behavior of that operation. Behavioral reflection only provides the ability to alter the behavior of operations in a program but does not provide the ability to alter data structures used in the program, which are statically fixed at compile time. On the other hand, structural reflection allows a program to change the definition of a class, a record etc on demand. Thus as an extension of Java, to support structural reflection, *Javassist* was developed [30]. Javassist is a class library for enabling structural reflection in Java.

Javassist enables structural reflection at load-time, that is, only before a program is loaded into the JVM or a runtime system. After a java class is compiled, the bytecodes are stored in separate class files. Javassist performs structural reflection by translating alterations by structural reflection into equivalent bytecode transformation of the class files [30]. Please see Code-3.1 for a very basic example of how a program in Javassist is written.

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("test.Rectangle");
cc.setSuperclass(pool.get("test.Point"));
cc.writeFile();
```

Code-3.1

Basic Terminology

`CtClass` – Also called compile-time class. A `CtClass` object is the reification of the bytecodes of a class loaded in the JVM. This reification is step is the first step that has to be performed in order to be able to access the class from the program. In other words, a Javassist `CtClass` is an abstract representation of a class file. A `CtClass` object is a handle for dealing with a class file [31]**.**

`ClassPool` – A `ClassPool` object controls bytecode modification with Javassist. A `ClassPool` is a container of all `CtClass` objects. A `ClassPool` object can read a a class file on demand in order to generate a `CtClass` object. Once a `CtClass` object is created, it is recorded in the `ClassPool` forever. This is done because the `CtClass` object may have to be accessed later when the source code of the class, which is represented by this `CtClass`, is re-compiled by the Javassist embedded recompiler.

`detach()` - As mentioned earlier, a `CtClass` object is recorded in the `ClassPool` forever. This may result in large memory consumption. Thus, to prevent out os memory problems, a `CtClass` object that is not used anymore or is redundant can be explicitly removed from the `ClassPool` by giving the detach command.

`writeFile()` – After structural changes are done on the class file, they have to be written down so that the next time the changes are reflected. For this a `writeFile()` method is available which writes a class file represented by this `CtClass` object in the current directory.

3.2 Java Annotations

Annotations are a new feature introduced in J2SE 5.0 (Tiger). With annotations one can add metadata to Java. Although metadata had become one of the latest trends in programming, annotations provide a more versatile way of adding metadata. Annotations are modifiers you can add to your code and apply to package declarations, type declarations, constructors, methods, fields, parameters, and variables [32].

Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program. Annotations can be read from source files, class files, or reflectively at run time [31].

Annotations are represented as `@` sign followed by the name of the annotation. Annotations can be supplied data values also wherever needed.

There are three types of annotations:

1. **Marker Annotations** – These annotations have no variables. No data needs to be supplied to marker annotations and they just appear as names. For instance, `@MarkerAnnotation`
2. **Single value Annotations** – These annotations have just a single data member. Only one value is supplied by these annotations. For instance `@Wrap("Point")`
3. **Full Annotations** – These annotations provide multiple data members. For instance. `@FullAnnotation(var1="value1", var2="value2")`

There are three built-in annotations namely `@Override`, `@Deprecated` and `@SuppressWarnings`. Besides these ready made annotations, a user can also build custom annotations.

3.3 Reflex – Structural Cuts and SMetaobject

A basic idea of what reflex is has already been given in Chapter 2. Reflex was basically developed for behavioral modification. It initially included the possibility of structural reflection via Javassist (embedded in Reflex in order to modify method bodies to implement behavioral reflection) but it has been recently refined in order to better support interactions

between structural and behavioral modification [22]. But now-a-days it is being extended to support structural modifications. Here I will talk about how structural modifications are achieved in Reflex.

Reflex relies on the notion of defining a `link` explicitly which binds a *cut* to an *action.* Cuts and actions can be either structural or behavioral. A structural link or an S-link binds a structural cut to an action. An S-link is applied, that is, its associated action is performed at load-time [22]. A structural cut in Reflex is a class set defined by a class selector.
The following snapshot shows a class selector defining a cut consisting of classes that contains an annotation called `Wrap` (Please refer to section 3.2 of this chapter for annotations).

```
public void initReflex() {
      ClassSelector wrapperSelector = new ClassSelector() {
            public boolean accept(RClass aClass) {
                  Annotation annotation =
                        aClass.getAnnotation("theWrapperAPI.Wrap");

                        if (annotation == null)
                              return false;
                        else
                              return true;
                  }
            };
      }
}
```

Code-2.2

The above code shows just one class selector, but there can be many class selectors implemented in `initReflex`. Similary, for each class selector there has to be a corresponding structural link. And also for each structural link defined, there has to be a corresponding `SMetaobject` (structural metaobject). Thus a cut is defined by class selectors and the action is defined by the corresponding metaobject, the `SMetaobject`.

Reflex operates in two phases at load-time, the SLA (S-link application) phase and the BLS (B-Link setup) phase. B-links are *set up* at load-time in the BLS phase and S-links are *applied* at load-time in the SLA phase.

Summary

These sections give some basic idea of about the tools that I have used for my thesis project.
The first section gives a brief idea of what Javassist is and some of the methods or terms that are frequently used in my thesis. The second section talks about Reflex, with emphasis on structural modification and how it is achieved in Reflex. The third section gives a very basic idea about what Java annotations are. The purpose of this chapter was to make the reader aware of some of the common terms that are used in the project so that reader may know what those terms mean and what they are.

Chapter 4. The WrapperAPI

Structure

Section 4.1 is meant to give an overview of some of the terms that will be used in the thesis. It will lay the foundation for my thesis and will give an idea of what is to be expected from the following sections.
The integration of aspects and components takes place in two steps – i) Generating wrappers for wrapping components and then ii) binding aspects and components using these wrappers. Section 4.2 'Wrapping Components' and section 4.3 'Binding Aspects and Components' are devoted to the two above mentioned steps, respectively. These two sections expose the API that is used for accomplishing wrapping and binding along with small examples. Finally section 4.4 gives examples to show the different uses of wrappers.

4.1 Building Blocks for Creating Wrappers

4.1.1 General View of a Component

We consider here components implemented using a standard class-based object-oriented language.

We define a component as a composition of classes in which all the classes work in co-ordination with each other to implement a specific functionality and all these classes provide together the methods declared in the interface. The component interface is the union of all the interfaces implemented by these classes. All the components are black boxes, that is, the source code is not available to us. Hence the names of the classes are not known. The only thing that is known to us is the contractually specified interface of the component. In other words, the functionality of a component is hidden behind the interface of the component. Please see Fig-4.1 below for a general view of the component.

Thus in order to wrap a component, its interface has to be known. Also, the wrapper class has to accept an array of objects belonging to different classes that are encapsulated in the component and all these objects have to be wrapped one by one. Furthermore, if there is a call for a specific method, then the wrapper has to know to which class the method belongs to, so that it can be forwarded accordingly.

Hence in order to make things simpler, I am considering that all the individual classes that are present in the component are linked to each other in a common class which will be responsible for forwarding method calls to appropriate classes. These classes will also implement the same interface as the component. Thus at the moment, I am considering a normal Java Component. For me a Java Component is a class file implementing an interface. This interface is a simple Java Interface with no provided and expected methods (all the methods are provided methods) (See Fig-4.2). But there are drawbacks of this approach. First, there is an extra level of forwarding of method calls. Instead of having the wrapper calling the method of the class, enclosed in the component, directly, it first calls the common

class and that class in turn forwards the method calls to the appropriate class (See Fig-4.3). The classes A, B and C are therefore wrapped twice (by an “implementation wrapper” ABC and by a component wrapper). Secondly, interfaces are not contractual interfaces as defined for components.

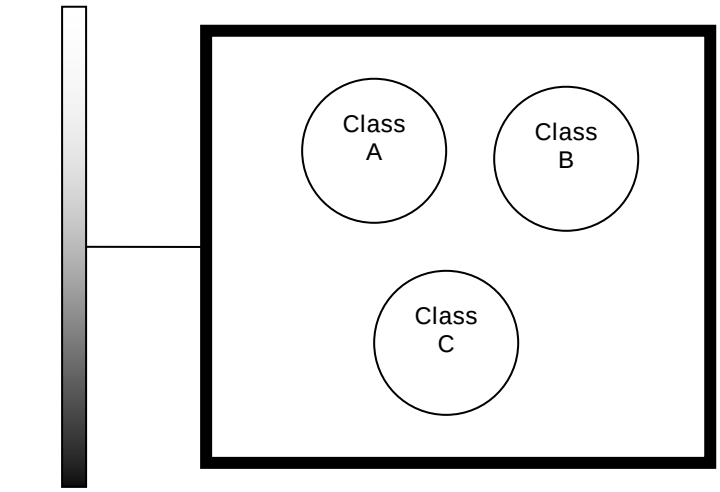

Fig-4.1 General View of a Component

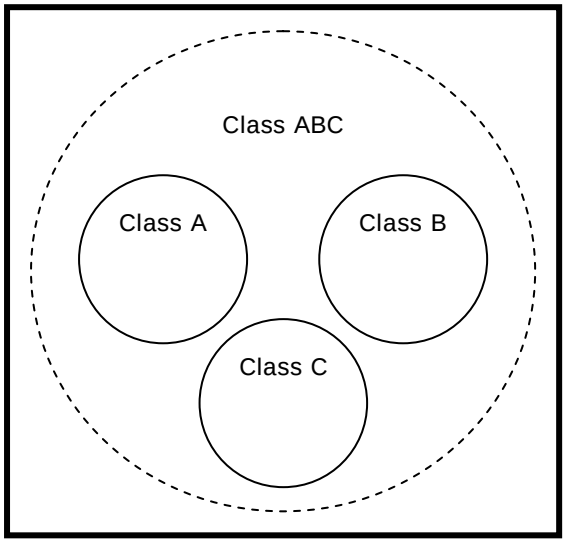

Fig-4.2

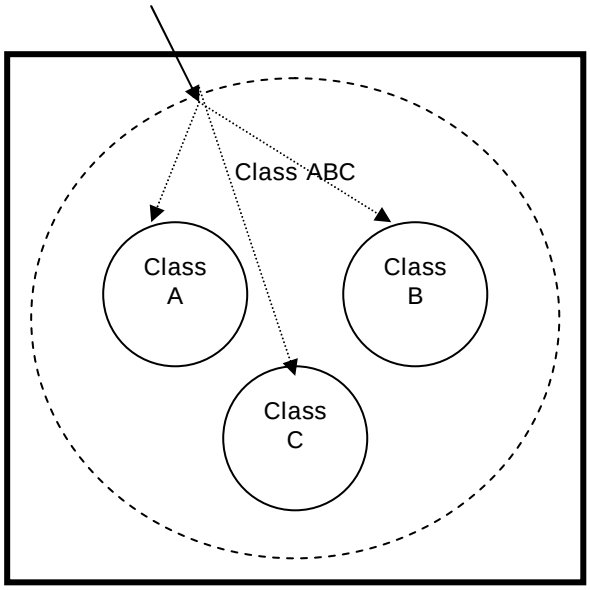

Fig-4.3

4.1.2 Overview

The foundation of wrappers is laid after taking into consideration the following requirements:

1. Components are black boxes. Hence *aspects* should not rely on the source code of the component.
2. Aspects and Components have to be considered two separate entities and the language should ideally provide facilities to integrate aspects with components at runtime.
3. Aspects should not be hardwired into the source code.
4. Aspects should be easily reusable with different components.

Thus following are the two main entities at the user-level:

1. Components
2. Aspects

Components – As said earlier, the component encapsulates a particular functionality which is hidden behind the interface. In my case, it is hidden behind a Java Interface. I will be considering that only the name of the Java Interface is known and the name of the class file is unknown. This has been assumed so that properties of a component (represented here as a Java interface) can be incorporated in the wrapper API.

Aspects – Aspects are modules or units for dealing with crosscutting concerns. The structuring of aspects is based on the idea of the aspect structure given in the Caesar model [3]. Hence, an aspect is composed of the following three parts:

1. Aspect Interface
2. Aspect Implementation
3. Aspect Binding

The aspect implementation is written as a normal Java class. This class implements a Java interface which in the above case is the aspect interface. Aspect binding is also a Java class whose main function is to bind the aspect implementation with the component.

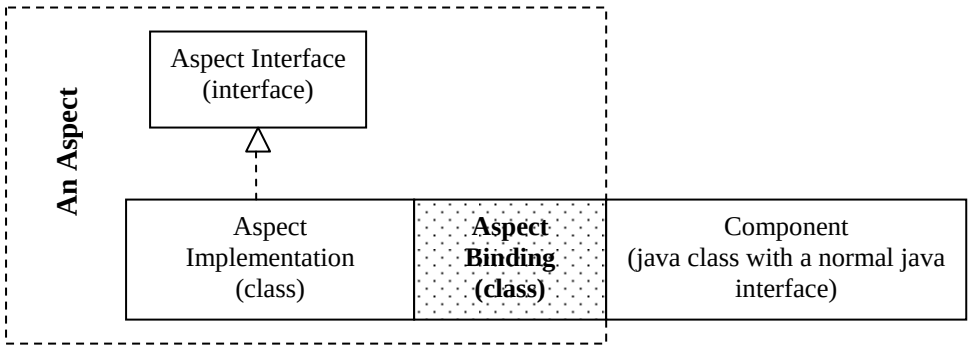

Fig-4.4 General Structure of an Aspect

The above structure of an aspect shows one component bound to the aspect implementation. But when many components have to work in collaboration with each other, then aspect

implementation and aspect interface together defines a *role* that a component has to play (See Fig-4.4). In this case, the function of an aspect binding class is to bind the roles to one or more components. All aspect implementation classes implement an interface and they have a naming convention which is as follows:

InterfaceName + "`Impl`"

For instance, if the name of the interface is `Subject`, then the aspect implementation class implementing this interface will be named `SubjectImpl`. The reason behind this naming convention will be dealt with in section 4.3.2. The aspect interface `Subject` and the aspect implementation class `SubjectImpl` together defines the role `Subject`.

After defining what a component and an aspect is, the next question arises – How do we integrate aspects with components? How do we achieve collaboration between components? Here, wrappers come into play.

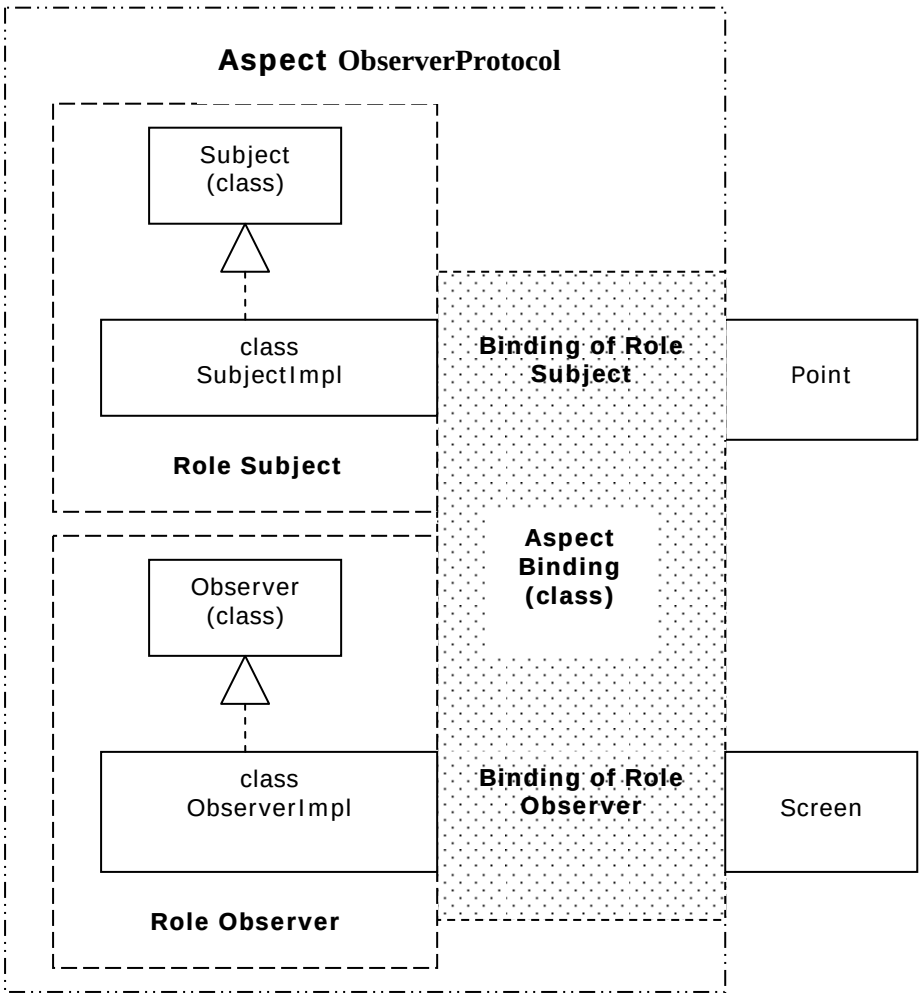

Fig-4.4. Structural view of an Aspect supporting collaboration

Wrappers

Wrappers are entities responsible for the integration of aspects and components. Thus at runtime, only wrappers are present and aspects and components lose their presence at this stage. In my thesis, wrappers are basically class files, which are built implicitly by Javassist.

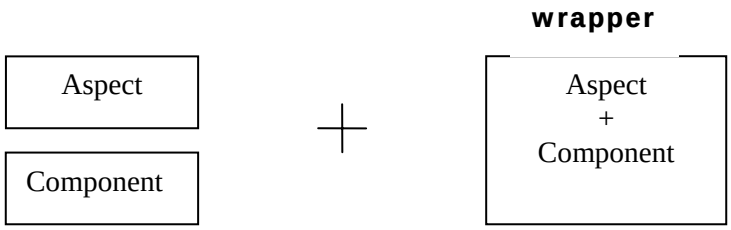

Fig-4.5 Wrappers for integrating Aspects and Components

The integration of aspects and components is a two step process.

1. Generation of Wrappers (covered in Section 4.2)
2. Binding aspects and components using these wrappers (covered in Section 4.3)

All the methods required for wrapping a component and binding aspects and components are present in a class called `WrapperAPI`.

The wrappers generated by this class can be used for the following purposes:

1. Changing / modifying the behavior of the component
2. Implementing Aspects
3. Achieving collaboration between components

These have been explained in detail in section 4.4 with examples.

The wrapper API is generic enough to wrap either Java class files / component implementation or Java interfaces / component interfaces. Furthermore, the API can either generate new wrappers or convert existing classes into wrappers.

Generation of new wrapper classes will be dealt in the next section which will give an overview of how wrappers are generated and implemented. Conversion of classes into wrappers will be dealt with, in chapter 5.

Section Summary

The main purpose of this section was to give a bird's eye view of the various ingredients that have gone into my project and also to give an idea of what is to be expected from this thesis. This section is like a very general introduction to the following sections.

4.2 Wrapping Components

4.2.1 Available Approaches

Inheritance

Inheritance means extending or modifying the base class with some new behavior or functionality without destroying it. In Java, inheritance is accomplished using the 'extends' keyword. The syntax for inheriting a class is as follows:

```
class NewClass extends BaseClass{ }
```

`NewClass` is the class defining the new behavior which has to be added to the `BaseClass`. From the above given syntax we can see that the `extends` clause requires the name of the class which has to be extended.

As mentioned earlier, we only know the name of the interface of the component. Thus inheritance cannot be used for wrapping components because the classes are encapsulated in a component and hence class names are unknown to us.

Superimposition

Superimposition is basically copying the behavior of one class into another class. If we take the example given above in inheritance, the behavior of the `BaseClass` will be copied into the `NewClass`. In a component, we don't have access to the source code. Hence this approach also cannot be applied.

Forwarding

Forwarding is the act of sending a message to another location. In computer terminology, forwarding is the act of sending a message received by one object to another object. This seems the most workable approach because the interface of the component is known. Also when a component is instantiated, the resulting object has the type of the component interface. The interface can be used to copy the signature of the methods into the wrapper and the object can be used for forwarding method calls to the component.

4.2.2 Description of the API

The class WrapperAPI can create wrappers for wrapping either java classes or components. Hence either the interface name of the component or the class name can be passed as a wrappee, the only condition being that it has to be a string.

There are two methods available for wrapping a component:

1. `wrap(String wrappee)`
2. `wrap(String wrapperName, String wrappee)`

The `wrap` method takes in as a parameter the name of the interface of the component which needs to be wrapped and generates a specialized wrapper class for that component. The programmer can choose whether to supply the name of the wrapper or not. If the wrapper name is supplied, then the wrapper is created with the given name otherwise the wrapper name is automatically generated. The automatic generation of wrapper names follows the following naming convention – *InterfaceName* + "`Wrapper`".
For example, if the name of the interface is `IPoint`, then the name of the wrapper will be `PointWrapper`.

In general, there are four steps involved in the creation of wrapper classes for components using method `wrap`:

1. Creating a constructor which receives as a parameter the object of type interface.

2. Inserting an instance variable which is of the proper interface type.

3. Copying the interface of the component.

4. Setting the body of all methods such that method calls are forwarded to the component.

4.2.3 Illustration for Generation of Wrapper Classes using an Example

Although the method for wrapping components and classes is same, here we will talk about wrapping Java components. Please see Fig-4.6.

Component Name: `Point`
Interface implemented by Component Point: `IPoint`
Wrapper Name: `PointWrapper`

When the name of the interface is passed as a parameter, the `CtClass` (compile time class) is retrieved from the pool (please refer to chapter 3 section 3.1 for Javassist terminology).

The wrappers are created at the byte code level.

Now let us take a look at how a wrapper class is generated. The name of the wrapper to be created is `PointWrapper`. Following steps are involved in the creation of the wrapper class:

1. Creation of a class file

2. Inserting the instance variable `wrappee` which will be of type interface. In the above given example, it will be of type `IPoint`.

```
class PointWrapper{

      IPoint wrappee;
}
```

Code-4.1 class PointWrapper after Step 2

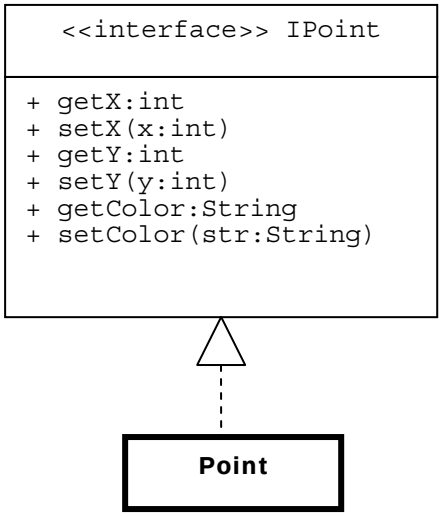

Fig. 4.6 Component Point implementing interface IPoint

3. Inserting the constructor – To wrap a component implementing the interface `IPoint`, the wrapper class receives an object of `type IPoint` (the interface name). In general, the constructor of the wrapper class receives an object of *type interface*. A class can also be passed as a wrappee. In that case the parameter type and instance variable will be of *type class*. This means that either:

   `Point p = new Point();` or `IPoint p = (IPoint) new Point();`

   can be executed once the wrapper class has been generated. After the insertion of the constructor and the instance variable, the wrapper class will look as follows:

```
class PointWrapper{

      IPoint wrappee;

      public PointWrapper(IPoint wrappee){
            this.wrappee=wrappee;
      }
}
```

Code-4.2 class PointWrapper after Step 3

4. copying the interface of the component – All the method signatures present in the component interface are copied into the wrapper class.

```
class PointWrapper{

      IPoint wrappee;

      public PointWrapper(IPoint wrappee){
            this.wrappee=wrappee;
      }
      public int getX();
      public void setX(int x);
      public int getY();
      public void setY(int y);
      public String getColor();
      public void setColor(String str);
}
```

Code-4.3 class PointWrapper after Step 4

5. Filling in the method body of each method for forwarding calls to the component – In this step, the method body of each method is defined. The method body is basically forwarding the method call to the wrappee.

```
Class PointWrapper{

      IPoint wrappee;

      public PointWrapper(IPoint wrappee){
            this.wrappee=wrappee;
      }

      public int getX(){
            wrappee.getX();
      }
      ……
      ……
}
```

Code-4.4. Class PointWrapper after Step 5

It is clear from the above process for the generation of wrappers that the component object or the wrappee resides in the body of the wrapper object (Fig-4.7).

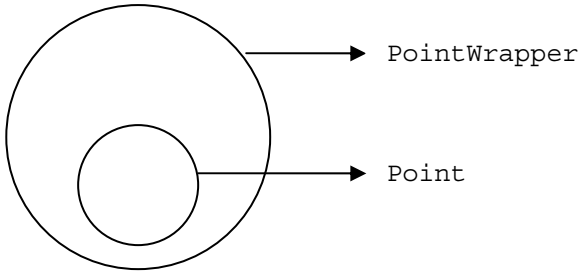

Fig-4.7 Wrappee point encapsulated within wrapper PointWrapper object

Here is a snapshot of the code that uses the wrap method to build wrappers.

```
class CreateWrappers{
      public static void main(String args[]){
            WrapperAPI w = new WrapperAPI();
            w.wrap("PointWrapper","IPoint");
      }
}
```

Code-4.5 Usage of 'wrap' method of wrapper API

The method `wrap` of the API accepts two parameters of type 'String'. The first parameter is the name of the wrapper and the second parameter is the name of the wrappee. In the above code snippet, the name of the interface is passed as a wrappee. The component object that will be received will be of type '`IPoint`'.

4.3 Binding of Aspects and Components

4.3.1 Available Approaches

For binding of aspects and components, there are also three available approaches. These three approaches are as mentioned in section 4.2.1 above. All these approaches can be used for binding an aspect implementation class to different component classes. But each approach has its pros and cons.

Inheritance

If one component has to be statically bound to only ONE aspect implementation class, then inheritance is the best approach. It has following advantages and disadvantages:

Advantages

- The wrapper class extends the base class and hence no new methods or classes have to be redefined in order to reuse the methods of the base class.
- It is efficient because there is no indirection of method calls to the wrappee.

Disadvantage

- It can only be applicable when one component object has to be bound to only one aspect implementation class. In the case of collaboration where the same component class has to play multiple roles, inheritance approach fails because Java does not support the concept of multiple inheritance.
- It is static. The binding cannot be undone at runtime.

Superimposition

Advantage

- It is quite efficient because the behavior is copied into the body of the wrapper class and hence there is no indirection of method calls.

Disadvantage

- If the aspect implementation class changes, then all the copies made from this class will have to be updated.
- The reuse of the class is not good.

Forwarding

Advantage

- The aspect implementation class is reused. There is no need to redefine or copy the methods.
- It is dynamic

Disadvantage

- The method calls are first intercepted by the wrapper and then forwarded to the wrappee. Hence there is an indirection which brings down the performance.

Hence this approach is feasible when reuse of the class is more important than performance. It is the only choice if the binding has to be undone at runtime.

Thus it can be seen that there are tradeoffs of all the three approaches. Each approach can be good for a particular situation. Hence the wrapper API should be able to support all the three approaches. At present it supports forwarding and inheritance approaches for binding of aspects and components.

4.3.2 Description of the API

The binding of an aspect and a component is basically the integration of two Java classes (one implementing the aspect and one implementing the component). But in case of an aspect supporting a collaboration, it is more complicated as each role part of the aspect has to be assigned to the corresponding component. Here we will talk about collaborations and how roles can be assigned.

Assigning Roles

Section 4.2.3 illustrated using an example how wrapper classes are created. Once a specialized wrapper class for a component is generated, it can be adapted to play a specific role / roles. For this purpose, the wrapper API contains two methods:

1. `assign(String role, String wrapperName)`
2. `forward(String role, String wrapperName)`

The `assign` method implements the inheritance approach and the `forward` method implements the forwarding approach. These methods take in two `String` parameters:

1. the name of the role that the component has to play which is equivalent to the interface name and

2. the name of the wrapper class that was generated for that component.

The code snippet for assigning roles is as follows:

```
class AspectBinding{
      public static void main(String args[]){
            WrapperAPI wapi = new WrapperAPI();
            wapi.forward("Subject","PointWrapper");
      }
}
```

Code – 4.6 Assigning roles using 'forward' method of the API

In Code-4.6, `Subject` is the name of the aspect interface implemented by the class `SubjectImpl` and `PointWrapper` is the wrapper class that was generated for the component which has to be wrapped.
When the interface name is passed as a parameter, the method `assign` and `forward` searches for the class implementing this interface. The class implementing this interface, by

naming convention, is *InterfaceName* + "`Impl`". Thus if the interface name is `Subject`, then the class implementing this interface will be `SubjectImpl`. When the class is found, the wrapper will either inherit this class or forward method calls to this class depending on the method used for assigning the roles.

If the inheritance approach is used, then the methods that need to be overridden in the superclass can be inserted in the wrapper using the `insertMethod`. This method takes in two `String` parameters – the name of the wrapper class to which the method has to be added and the complete method, i.e method signature and method body, as a string. For example, if we want to add a method `display` to the wrapper class `PointWrapper`, then the following has to be done:

```
insertMethod("PointWrapper","public void display()
                {System.out.println(\"In display()\");}");
```

In the case of forwarding mechanism, if the method body has to be replaced with another method body, then `around` method can be used. This method accepts three `String` parameters. The first parameter is the name of the method whose body has to be replaced, the second parameter is the complete method and the third parameter is the name of the wrapper class. The usage of this method has been shown in the next section.

4.3.3 Illustration for Binding Aspect and Components using an Example

In section 4.2.3 we saw with the help of an example how a wrapper class is created. Now in this section we will see how a wrapper class will be adapted to play the role of a `Subject`. The following example is just a part of a more elaborate and complete example of collaboration. The main purpose here is to show how roles are assigned to component objects using the forwarding approach.

Wrapper name: `PointWrapper`

Role: { Interface Name: `Subject`
Class implementing interface Subject: `SubjectImpl`

Please see Fig-4.8

The first step to assign a role is to call the `forward` method as shown in Code-4.6. When this method is called, it automatically retrieves the class `SubjectImpl` from the `Classpool` and the method calls are forwarded to class `SubjectImpl`. Following four steps are required for implementing forwarding:

1. Modifying the constructor of the wrapper to accept a parameter of type role. In the example it is of type `Subject`.

2. Inserting an instance variable `role` which will also be of type role (See Code-4.7)

3. Copying the aspect interface

4. Inserting method body to forward method calls (See Code-4.8)

The interface `Subject` contains a method called `getState` (refer to Fig-4.8). This method is supposed to be implemented in the wrapper class which will return the current state of the wrappee (this is actually defined in aspect binding). In our example it has to return the current color of the wrappee. But the class `SubjectImpl` implements the interface `Subject`, and hence this method has to be present in the class `SubjectImpl`. In Fig-4.8, it can be seen that this method returns an empty string. So when the wrapper class forwards method calls to the class `SubjectImpl`, the method `getState` will return an empty string. So a new method body has to replace the method body that is implemented in `SubjectImpl`. This can be done with the help of the method `around`. The following has to be done:

```
around("getState","public String getState(){return
                          wrappee.getColor();}","PointWrapper");
```

After the insertion of the method, the wrapper class will look as given in Code-4.9.

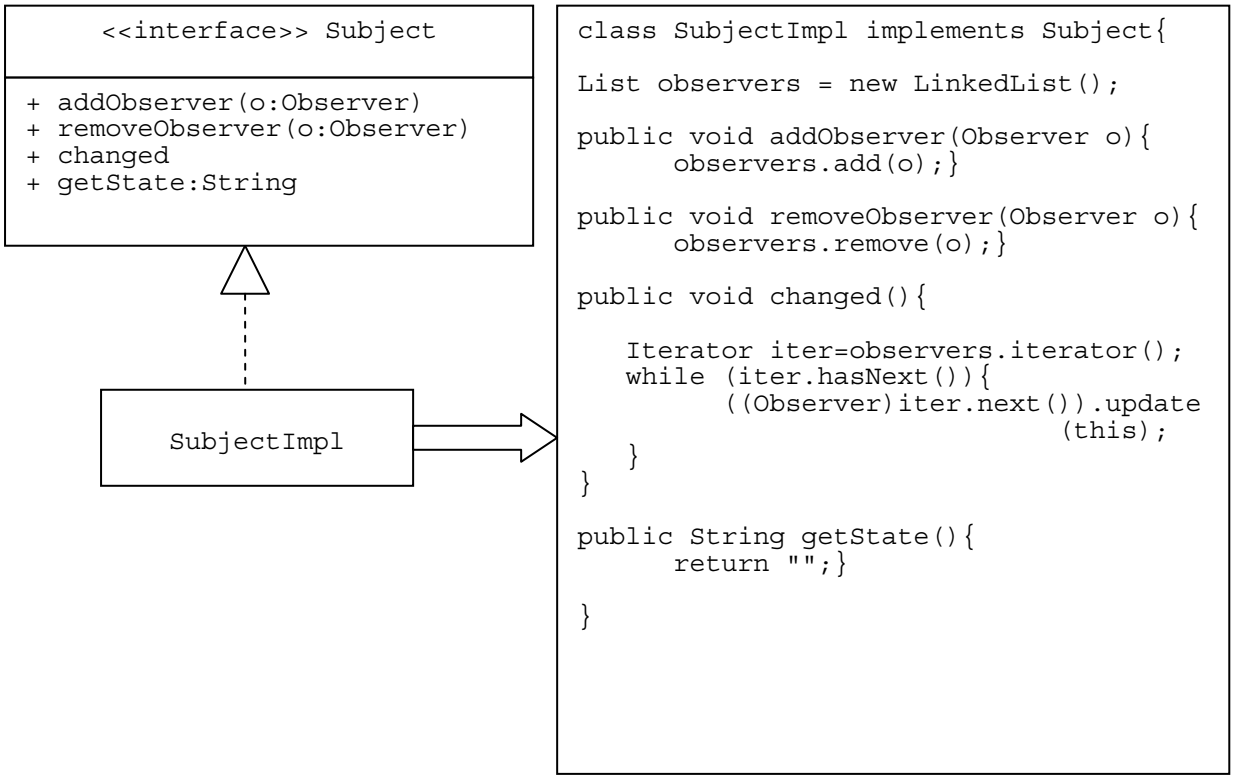

Fig-4.8 Role Subject

```
class PointWrapper{

      IPoint wrappee;

      Subject role;

      public PointWrapper(IPoint wrappee){
            this.wrappee=wrappee;
      }

      public PointWrapper(IPoint wrappee, Subject role){
            this(wrappee);
            this.role=role;
      }

      public int getX(){
            wrappee.getX();
      }
      ……….
      ……..
}
```

Code-4.7 New constructor added to support 'forwarding'

The wrapper class now is fully adapted to play the role of a Subject. All the wrappees wrapped by this wrapper class will play the role of `Subject`.

Using the forwarding technique, we can see that the instance of the role class (aspect implementation class) resides inside the body of the wrapper object. Thus the wrappee and the role class object both reside in the body of the wrapper object.

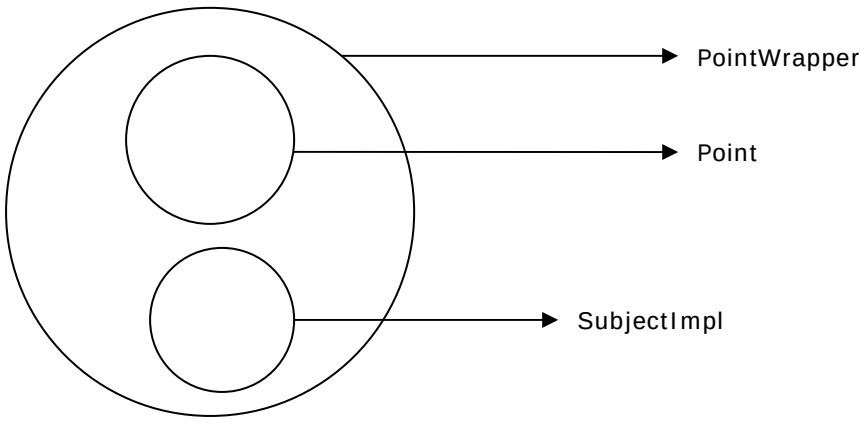

Fig-4.9 Wrappee Point and role object SubjectImpl residing inside the body of the wrapper object

```
class PointWrapper{

      IPoint wrappee;

      Subject role;

      public PointWrapper(IPoint wrappee){
            this.wrappee=wrappee;
      }

      public PointWrapper(IPoint wrappee, Subject role){
            this($1);
            this.role=$2;
      }

      public void getX(){
            wrappee.getX();
      }
      ……….
      ……..
      public void addObserver(Observer o){……..}

      public void removeObserver(Observer o){……..}

      public void changed(){
            role.changed();
      }

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

Code-4.9 class PointWrapper after inserting the method 'getState' using around advice

4.4 Case Study – Various Uses of Wrappers

4.4.1 Wrappers for Changing / Modifying the Behavior of a Wrappee

In this section we will see how wrappers can help in changing / modifying the behavior of a wrappee.

Problem Statement

There is a component `Point` which implements an interface called `IPoint`. Now we want that if the color of point is set to Blue, then the color should be changed to Yellow. Let us see how it can be accomplished.

Solution

Interface Name: `IPoint`
Component which implements this interface: `Point` (See Fig-4.6)

First, a wrapper for Point is generated as explained in section 4.2.3. The generated wrapper is called `PointWrapper` (please refer to Code-4.5).

The next step is to extend the behavior of the wrappee using this wrapper. We want to insert a piece of code which evaluates whether the color of the wrappee is set to Blue or not. If it is set to Blue, then the color should change to Yellow.

For inserting such a piece of code which changes the behavior of the wrappee, three methods are provided in the API.

`Before` – It inserts the given piece of code before the execution of the given method.
`After` – It inserts the given piece of code after the execution of the given method.
`Around` -It inserts the given piece of code in place of the given method.

The color of the point is set using the method `setColor(String color)`. We want to intercept calls to this method, and check whether the color is set to Blue or not. If it is Blue, then the new color Yellow should be set. The method that returns the current color of the wrappee is `getColor`. So after the color has been set, we want call `getColor` to retrieve the current color. If the current color is Blue, then we want to set the color to Yellow using the `setColor` method. This can be achieved in the following way:

```
class ColorChanger{
      WrapperAPI wapi = new WrapperAPI();
      wapi.after("setColor","{if (wrappee.getColor()==\"Blue\")
                        wrappee.setColor(\"Yellow\");}","PointWrapper");
}
```

Code – 4.10 Usage of 'after' advice to change the behavior of the wrappee

The method `after` receives three String parameters. The first parameter is the name of the method after which the given piece of code has to be executed. This given piece of code is the second parameter. The last parameter is the name of the wrapper wrapping `Point`.
After executing the above class, whenever the color is set to Blue, it is changed to Yellow. The main class shown in Code-4.11 produces the following output:

```
E:\EMOOSE\Research\Javassist\javassist-3.0\Examples>javac MainClass.java

E:\EMOOSE\Research\Javassist\javassist-3.0\Examples>java MainClass

The Color is set to Red
Current Color = Red

The Color is set to Blue
Current Color = Yellow

E:\EMOOSE\Research\Javassist\javassist-3.0\Examples>
```

```
public class MainClass{
      public static void main(String args[]) throws Exception{

            Point p = new Point();

            PointColor pc = new PointColor(p);

            System.out.println("The Color is set to Red");
            pc.setColor("Red");
            System.out.println("Current Color = " + pc.getColor());

            System.out.println("");
            System.out.println("The Color is set to Blue");
            pc.setColor("Blue");
            System.out.println("Current Color = " + pc.getColor());
      }
}
```

Code – 4.11 Class to show how wrappers are used to change the behavior of a wrappee

4.4.2 Wrappers for implementing Aspects

Here we will see how wrappers can be used for implementing aspects.

Problem statement

There are two components – Point and Line each implementing interfaces `IPoint` and `ILine` respectively as shown in Fig-4.9. They both have `setColor` and `getColor` methods for setting and getting the color of Point and Line and they both have the same implementation for the `setColor` method, that is, whenever the color is set to Blue, they log it. In my case they do a `System.out.println()` instead of logging it in a file. This indicates that they both implement a kind of logging functionality. This logging functionality crosscuts the class structure, and hence it can be factored out into modular units called aspects. According to our aspect structure, we have to develop the following three parts:

1. The Aspect Interface named `Logger`
2. The Aspect Implementation which will be a class called `LoggerImpl` implementing `Logger`
3. The Aspect Binding which will be a class called `AspectBinder` which will bind aspect implementation class called `LoggerImpl` to the two components and also implement advices to extend the behavior of the wrappee so that the desired result can be achieved.

Please see Fig – 4.10.

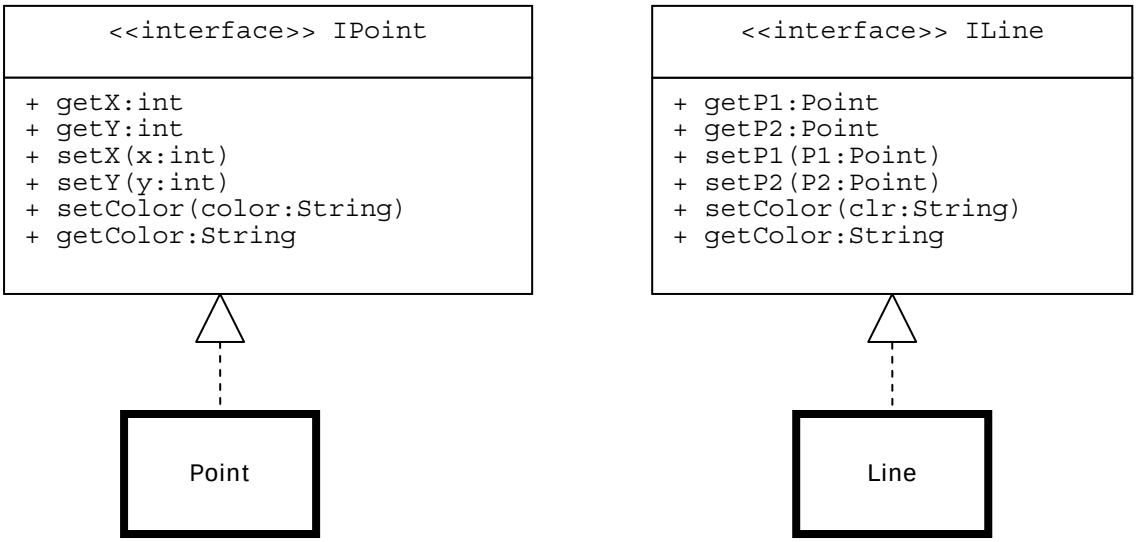

Fig-4.9 Interfaces implemented by Components Point and Line

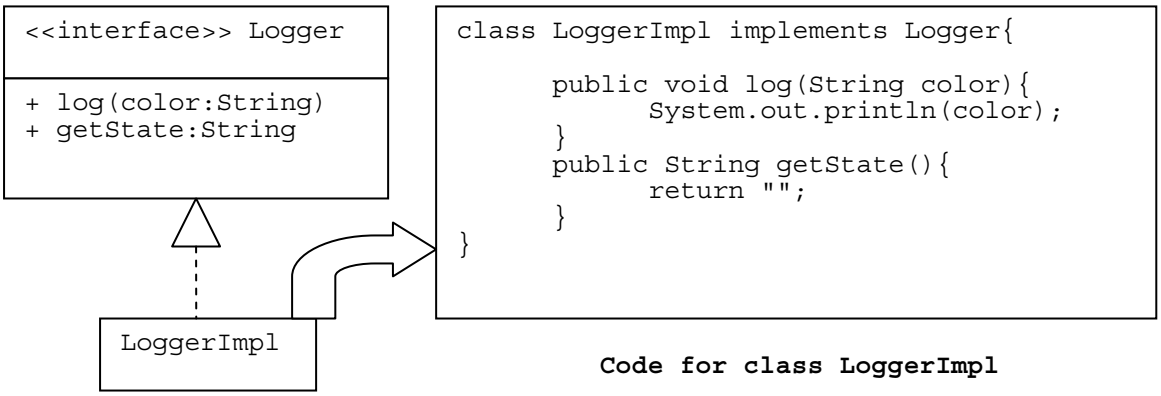

Fig – 4.10 Role Logger and its implementation

The first step is to create wrapper classes for `Point` and `Line`. The wrappers created are `PointLogger` and `LineLogger`. After this, the aspect implementation has to be bound to components `Line` and `Point` using wrappers as shown in Code-4.12.

When the binding phase is over, the wrappers are ready to execute the aspects.

```
class AspectBinder{

      WrapperAPI wapi = new WrapperAPI();

      wapi.forward(“Logger”,“PointLogger”);

      wapi.assign(“Logger”,”LineLogger”);

      wapi.after("setColor","if (wrappee.getColor()==\"Blue\")
                              log(wrappee.getColor());","PointLogger");

      wapi.after("setColor","if (wrappee.getColor()==\"Blue\")
                              log(wrappee.getColor());","LineLogger");
}
```

Code – 4.12 Binding of Aspect Implementation class to Components Point and Line using wrappers

The following main class:

```
public class MainClass{
      public static void main(String args[]) throws Exception{
            Point p = new Point();
            Line l = new Line();
            Logger lImpl = (Logger) new LoggerImpl();
            LineLogger ll = new LineLogger(l, lImpl);
            PointLogger pl = new PointLogger(p, lImpl);

            System.out.println("Setting Color to Red in Point");
            pl.setColor("Red");

            System.out.println("Setting Color to Blue in Point");
            pl.setColor("Blue");

            System.out.println("Setting Color to Red in Line");
            ll.setColor("Pink");

            System.out.println("Setting Color to Blue in Line");
            ll.setColor("Blue");
      }
}
```

Code – 4.13 Class showing how wrappers are used for implementing aspects using forwarding approach

produces the following output. It can be seen that whenever the color is set to Blue, logging is done.

```
E:\EMOOSE\Research\Javassist\javassist-3.0\Examples\WrappersWithAspects>java Mai
nClass
Setting Color to Red in Point

Setting Color to Blue in Point
Logging....Blue

Setting Color to Red in Line

Setting Color to Blue in Line
Logging....Blue


E:\EMOOSE\Research\Javassist\javassist-3.0\Examples\WrappersWithAspects>
```

4.4.3 Wrappers for Collaborations

This section is meant to show a complete example of an aspect supporting a collaboration. Part of the example has already been discussed in sections 4.2 and 4.3.

Problem Statement

There are three components namely `Point, Line` and `Screen`. We want these components to work in collaboration with each other, that is, whenever a new color is set in either component `Line` or component `Point`, the changes should be reflected by component `Screen`. This means that the `Screen` object has to observe the color changes in `Point` and `Line objects`, and as soon as the color is changed, the `Screen` object should be updated and it should display the new color.

Solution

Here we will implement the Observer Pattern. According to the Gang of Four, the idea behind the observer pattern is to "Define one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."[26]

In this pattern, there are two roles – `Subject` and `Observer`. To define a role we need an aspect interface and a class implementing that interface which is aspect implementation. The role `Subject` has already been defined and can be referred to in Fig-4.8. The role `Observer` is defined as shown in Fig-4.11.

The components `Point` and `Line` have to play the role of `Subject,` and `Screen` has to play the role of `Observer`. First wrappers have to be generated for each component. These wrappers are named `PointSubject`, `LineSubject` and `ScreenObserver` respectively (refer to Code-4.14)

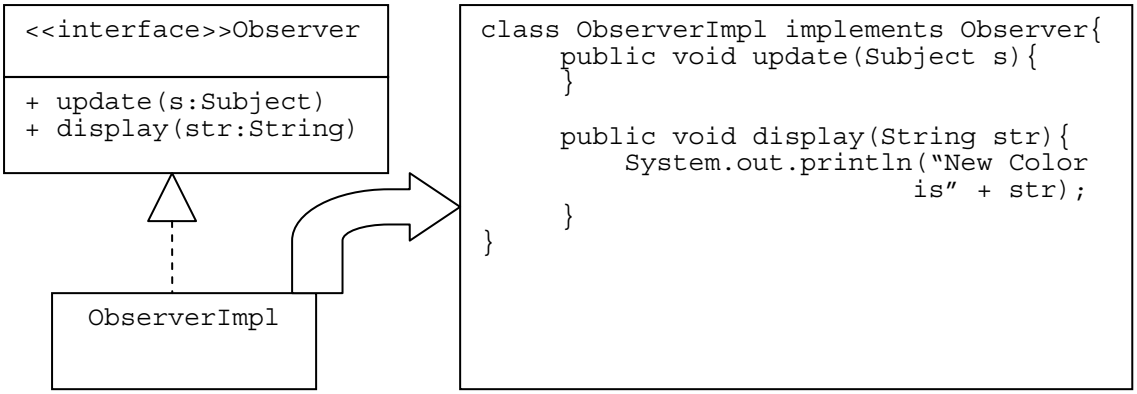

Fig – 4.11 Role Observer

```
public class ClassWrappers{
      public static void main(String args[]) throws Exception{

            WrapperAPI wapi=new WrapperAPI();
            wapi.wrap("PointSubject", "Point");
            wapi.wrap("LineSubject", "Line");
            wapi.wrap("ScreenObserver", "Screen");
      }
}
```

Code – 4.14 Generation of Wrappers for Components Point, Line and Screen

Once wrappers are generated, different roles have to be bound to different component objects so that the wrappees can play their expected roles. As seen in Code-4.15, `assign` method has been used to assign roles (unlike the example of section 4.3.3 which was using forwarding). As mentioned earlier, this method uses the inheritance approach to adapt a wrapper class to play a specific role.

As can be seen in Fig-4.8, `SubjectImpl` has a method `getState`. This method is supposed to be implemented in wrapper classes, playing the role of Subject, which returns the current state of the wrappee. In this example, it has to return the current color of the wrappee. When the `PointSubject` and `LineSubject` wrapper classes extend `SubjectImpl`, all methods are inherited including `getState`. But this method returns an empty string. Hence this method has to be overridden in both the wrapper classes so that it can return the current color of the wrappees. For method overriding, `insertMethod` can be used. Also an `after` advice is used for inserting a call to method changed which will update the `Screen` object. The binding phase comprises all these steps.

```
public class AspectBinding{
      public static void main(String args[]) throws Exception{
        WrapperAPI wapi=new WrapperAPI();

        wapi.assign("Subject","PointSubject");
        wapi.assign("Subject","LineSubject");
        wapi.assign("Observer","ScreenObserver");

        wapi.insertMethod("PointSubject","public String
                                getState(){return wrappee.getColor();}");

        wapi.insertMethod("LineSubject","public String
                                getState(){return wrappee.getColor();}");

        wapi.insertMethod("ScreenObserver","public void update(Subject
                                s){wrappee.display(s.getState());}");

        wapi.after("setColor","changed();","PointSubject");
        wapi.after("setColor","changed();","LineSubject");
      }
}
```

Code-4.15

The main class shown in Code-4.16 produces the following output:

```
E:\EMOOSE\Research\Javassist\javassist-3.0\PrototypeIII>java MainClass
Setting Color to Blue in Point
The color is Blue

Setting Color to Green in Line
The color is Green

Removing Observer from Point
Now setting Color to Pink in Point

Setting Color to Yellow in Line
The color is Yellow

E:\EMOOSE\Research\Javassist\javassist-3.0\PrototypeIII>
```

It can be seen clearly that when the observer is removed from Point, nothing is reflected on the console. Thus the added collaboration between components is working.

```
public class MainClass{
      public static void main(String args[]) throws Exception{

      Point p=new Point();
      Line l=new Line();
      Screen s=new Screen();

      PointSubject ps= new PointSubject(p);
      LineSubject ls = new LineSubject(l);
      ScreenObserver so = new ScreenObserver(s);

      ps.addObserver(so);
      ls.addObserver(so);

      System.out.println("Setting Color to Blue in Point");
      ps.setColor("Blue");
      System.out.println("");

      System.out.println("Setting Color to Green in Line");
      ls.setColor("Green");
      System.out.println("");

      System.out.println("Removing Observer from Point");
      ps.removeObserver(so);

      System.out.println("Setting Color to Pink in Point");
      ps.setColor("Pink");

      System.out.println("");
      System.out.println("Setting Color to Yellow in Line");
      ls.setColor("Yellow");

      }
}
```

Code – 4.16 Class showing how wrappers are used to accomplish collaboration

4.5 Chapter Summary

This chapter is devoted to showing how wrapper classes are generated and then how these wrapper classes help in binding of aspects and components. The examples are meant to show the various uses of wrappers as well as give an overview of methods available in the wrapper API. One thing important to note out here is that the wrappers are being generated explicitly. Until and unless wrappers are generated, binding cannot be achieved. The binding is done in a separate class which has been named AspectBinding. The next chapter will show a sample aspect binding class and how it can be parsed by Reflex to generate wrappers implicitly.

Chapter 5. Wrapper API in REFLEX

Chapter 4 introduced the various building blocks available in the wrapper API which are necessary for the creation of wrapper classes and how these wrapper classes can be adapted to play different roles in a collaboration. In this chapter we will first see how the wrapper API was integrated with Reflex and then we will see with the help of a small example how the Wrapper API can be used to build a language which uses the concept of wrappers for binding aspects and components like the Caesar.

Structure

Section 5.1 is devoted to explaining how the wrapper API was integrated with Reflex. Section 5.2 shows a Sample Aspect Binding class that was designed together with Java annotations, to give a feel of how the wrapper API can be used. Section 5.3 is an example which uses the binding class to achieve collaboration.

5.1 Integrating Wrapper API with Reflex

Overview

The Wrapper API was built in Javassist. This API now had to be integrated with the Reflex API. The Reflex API for structural modifications is fairly new. This API basically encapsulates the Javassist API. So the first idea was to leave the Wrapper API built in Javassist as it is because Reflex is built on top of Javassist. But this idea had to be dropped because Reflex works with `RClasses` and not `CtClasses` (`RClass` is very close to a `CtClass` as it actually encapsulates an `RClass` but with added Reflex functionality). Hence the procedure of integrating the Wrapper API using the Reflex API started. The first problem encountered here was that not all the features provided in Javassist API was available in Reflex API for structural modifications. Generation of wrapper classes was not possible. Therefore it was decided that `RClasses` that are retrieved from the `RPool`, the pool of `RClasses`, would be converted to `CtClasses`. All the modifications that are required to build a wrapper class will be done using Javassist API, but the rest of the other methods available in the Wrapper API will be integrated using the Reflex API.

Implementation Details

The Reflex API has a method called `asCtClass` in class `RClassImpl`. This method converts `RClasses` to `CtClasses`. Hence, first `RClass` is retrieved from the `RPool`. Then, `asCtClass` method is used to retrieve the `CtClass` of the given `RClass`. Thus, the generation of wrapper classes takes place with the help of Javassist API but the wrapper class that is returned is the `RClass` wrapper class and not the `CtClass` wrapper class. The procedure has been depicted in Code-5.1 given below. There is a class called `DummyClass` which has a method `convertToCtClass` which receives a parameter `aClass` of type

String `aClass` is the name of the java class file. In line number 5 a default `RPool` is retrieved. In line 6 the `RClass` of `aClass` is retrieved from the `RPool`. Once the `RClass` is there, the `asCtClass` method can be used to retrieve the corresponding `CtClass` as can be seen in line number 7.

```
1 public class DummyClass{
2     public RClass convertToCtClass(String aClass){
3           RClass aRClass;
4           CtClass aCtClass;
5           RPoolImpl pool = RPoolImpl.get();
6           aRClass = pool.get(aClass);
7           aCtClass = ((RClassImpl) aRClass).asCtClass();
8           return(aRClass);
9     }
10 }
```

Code-5.1 Dummy Class to show the conversion of RClasses to CtClasses

5.2 Sample Aspect Binding Class

As mentioned earlier, there are two steps for integrating aspects and components:

1. Generation of wrappers.
2. Binding of aspects and components.

In chapter 4, it can be seen that the wrappers were being generated explicitly. But this was not our aim. Our aim is to build wrappers implicitly, without letting the user know that they exist. In other words, the aspect binding class should automatically generate wrappers for integrating aspects and components. We need some language semantics to let the compiler know that the wrappers have to be generated. With Reflex, one can define structural cuts and the corresponding action that has to be taken. This action is defined as a MetaObject (MO). This facility of Reflex can be used for generating wrappers implicitly. The following sample aspect binding class was designed:

```
Class OuterClass{
      @Wrap("Wrappee1") @Bind("Role1") @Before("methodName","body")
      @Around("methodName","body") public class InnerClass1{ }

      @Wrap("Wrappee2") @Bind ("Role2") @After("methodName","body")
      @Insert("methodName","body") public class InnerClass2{ }
}
```

Code – 5.2 Sample Aspect Binding class

The above code shows an outer class called `OuterClass` and two inner classes called `InnerClass1` and `InnerClass2`. All the inner classes have annotations `@Wrap` and `@Bind`. These are single value annotations. The value passed to the wrap annotation

specifies the class that has to be wrapped (wrappee). For example, if there is an annotation `@Wrap("Point")`, then this means that the class `Point` has to be wrapped by the wrapper. Similarly, the bind annotation specifies the role that the wrappee has to play.
Now that we have a way of knowing the wrappee and the role that the wrappee has to play, we still do not know the names of wrapper classes. What names should be given to the wrappers that are created? To answer this query, the name of the inner class is used as the name for the wrapper. All the inner classes that are compiled follow a naming convention which is `OuterClass$InnerClass`. So according to Code-5.2 above, three Java class files will be compiled as follows:

1. OuterClass.class
2. OuterClass$InnerClass1.class
3. OuterClass$InnerClass2.class

The inner files will be the wrapper classes wrapping the wrappee which is passed as a value to the `@Wrap` annotation.

For example: `@Wrap("Wrappee1") public class InnerClass1`

means that class `Wrappee1` has to be wrapped by a wrapper named `OuterClass$InnerClass1.`

It is important to note out here that the wrapper classes that were generated by the class WrapperAPI in Chapter 4 were developed from scratch. This means that a new Java class file was being developed as a wrapper. But here we already have compiled classes that are supposed to be wrappers. Hence we will have to convert these compiled classes into wrapper classes. These wrapper classes will wrap their respective wrappee passed as a value in annotation `@Wrap`. *Thus the wrapper API had to be refined further to provide a facility for converting already existing classes into wrapper classes.*

Also in Fig-5.1, there are tags `@After`, `@Before`, `@Around` and `@Insert`. These tags accept two parameters – the name of the method and the body. If we talk about the `@After` annotation, then the first parameter is the name of the method after which the given body of the method has to be inserted. These annotations can be retrieved in a *binding* metaobject (there is also a *wrap* metaobject). If the annotations are present, then their values can be retrieved and respective API methods can be called to implement the advices or insert a method.

Refinement of the Wrapper API

It was decided that the already existing method for the creation of wrapper classes will be left as it is. Instead the wrapper API was extended to incorporate the facility of converting existing classes into wrapper classes.

The instances of all inner classes reside in an instance of the outer class. This means that for each inner class that is compiled, the compiler automatically inserts a default constructor which receives a parameter of type `OuterClass`. According to Code-5.2, the constructor of the inner classes will be as follows:

```
public OuterClass$InnerClass1(OuterClass oc){ }
```

Please note here that I have entered the parameter name `oc`. In reality, we don't know the name of the parameter because the parameter name is implicitly generated by the compiler.

Coming back to the topic, in order to convert this inner class into a wrapper class, the constructor should be modified to receive a parameter of type wrappee. If the class to be wrapped is Point (the wrappee), then the constructor should be as follows:

```
public OuterClass$InnerClass1(OuterClass oc, Point wrappee){ }
```

Unfortunately, there were no methods available in Javassist API or Reflex API which could help add a new parameter to the already existing constructor. This meant that a new constructor had to be inserted which will contain previous parameter types as well as the new parameter types. The code for constructing the constructor is shown in Code-5.3. In line 6 it can be seen that the CtConstructor takes in two parameters - a list of the parameter types and the class to which the created method is added. CtConstructor does not accept the parameter names. These names are however necessary to define the body of the constructor constructorBody. It turns out that Javassist provides metavariables `$1, $2...` to represent these names with `$i` representing the ith parameter name. In the case of the example, the body of the parameter is therefore defined by: `this($1); this.wrappee=$2;`

```
1 public void createConstructor(CtClass wrapper, String constructorBody,
2     CtClass[] paramTypes) throws NotFoundException, IOException,
3                                         CannotCompileException {
4
5
6           CtConstructor construct = new CtConstructor(paramTypes,
7                                                       wrapper);
8           construct.setBody(constructorBody);
9           wrapper.addConstructor(construct);
10    }
```

Code-5.3 Code to show how a constructor is created and added to the wrapper class

The rest of the procedure for building wrapper classes is as explained earlier.

5.2.1 Implicit Generation of Wrappers in Reflex

Here we will see how the above given language semantics is parsed by Reflex to generate wrappers implicitly. As mentioned earlier, for structural modifications there has to be a

structural cut and a corresponding action. The reification of the structural cut and the action is the S-Link that is defined. For the cut, we need to define a class selector. Classes are selected based on the presence of annotations. If the parser finds an annotation `@Wrap`, then that class is selected. The code for the selection of classes by the class selector is shown in Code-5.4 (see the definition of the method `accept`).

Once a class selector is defined, the second step is to define the action to be taken when this cut takes place. This action is defined in a metaobject class which implements `SMetaobject`. `SMetaObject` is the common interface implemented by all the metaobject classes that are defined. Each metaobject class has a method called `handleClass`. This method defines how the selected class should be dealt with. The MO for the creation of wrappers is defined as shown in Code-5.5.

```
public class Config implements IReflexConfig {
      public void initReflex() {
            ClassSelector wrapperSelector = new ClassSelector() {
                  public boolean accept(RClass aClass) {
                        Annotation annotation = aClass
                              .getAnnotation("theWrapperAPI.Wrap");
                        if (annotation == null) {
                                    return false;
                        } else {
                                    return true;
                        }
                  }
            };
      SLink wrapLink = API.links().addSLink(wrapperSelector,
                        new WrapperGenerator(), "wrapperLinkId");
      }
}
```

Code-5.4

```
public class WrapperGenerator implements SMetaobject {

      public void handleClass(RClass aClass) {

        Annotation annotation =
                        aClass.getAnnotation("theWrapperAPI.Wrap");

        MemberValue memberValue = annotation.getMemberValue("value");

        String weeName = ((StringMemberValue) memberValue).getValue();

        String str=new String(aClass.getSimpleName());

        WrapperAPI wapi = new WrapperAPI();
        try {
                  wapi.wrap(str,weeName);

        } catch (Exception e) {
                  e.printStackTrace();
        }
      }
}
```

Code-5.5

The wrappee is retrieved from the wrap annotation and the name of the class selected is also retrieved. These two values are passed as parameters to the wrap method defined in the wrapper API. The `wrap` method generates the respective wrapper class.

The link between the structural cut and the metaobject is defined as shown in Code-5.4. It has been highlighted. The link is the link between the structural cut and the corresponding action that has to be taken when the cut occurs. Hence the link that is defined accepts two parameters, the name of the class selector which defines the cut and the metaobject which defines an action for the cut. The third parameter is optional which is the identifier of the link.

In the above case (Code-5.4) it can be seen that `wrapperSelector` is the name of the class selector, `wrapperGenerator` is the Structural metaobject as shown in Code-5.5 and `wrapperLinkId` is the id of the link.

5.2.2 Binding of Roles

The value passed to the `@Bind` annotation specifies which role the wrappee has to play. The structural cut is defined in the same manner as was defined for wrappers. One more class selector is defined which select classes that have `@Bind` annotation as follows (Code-5.6):

```
ClassSelector BindSelector = new ClassSelector() {
      public boolean accept(RClass aClass) {

            Annotation annotation = aClass
                              .getAnnotation("theWrapperAPI.Bind");
                  if (annotation == null) {
                        return false;}
                  else{
                        return true;}
            }
      };
```

Code-5.6 Class selector for selecting classes having @Bind annotation

Then a metaobject is defined which specifies how the selected class should be handled. This is shown in Code-5.7 given below:

```
public class Binding implements SMetaobject{
      public void handleClass(RClass aClass) {

            Annotation annotation =
                              aClass.getAnnotation("theWrapperAPI.Bind");

            MemberValue memberValue =
                                    annotation.getMemberValue("value");

            String roleName = ((StringMemberValue)
                                                memberValue).getValue();

            String str = new String(aClass.getSimpleName());
            WrapperAPI wapi = new WrapperAPI();
            wapi.assign("role1",str);
      }
}
```

Code-5.7 Binding metaobject for handling classes selected in Code-5.5 above

In Code-5.7, the name of the role and the class selected by the class selector is retrieved and passed as parameters to `assign` or `forward` methods which are defined in WrapperAPI. These methods, as mentioned earlier, are responsible for binding roles to the components. The only difference is the approach they take – `assign` uses the inheritance approach and `forward` uses the forwarding approach. At the moment, it is being hardwired into the metaobject. But the selection of the technique should be dynamic.

The annotations `@After, @Before, @Around` and `@Insert` are also checked for in the same MetaObject. If any of these annotations is found, then the corresponding method name and body is retrieved and passed as parameters to either `after, before, around` or insert methods of the API, respectively.

5.3 Case Study

The wrapper API was designed to implement a Caesar-like language which uses the notion of wrappers. Hence this section will show how to implement the Observer Pattern example as given in [3] using the sample aspect binding class given above.

The roles `Subject` and `Observer` are as defined in Fig-5.1 below:

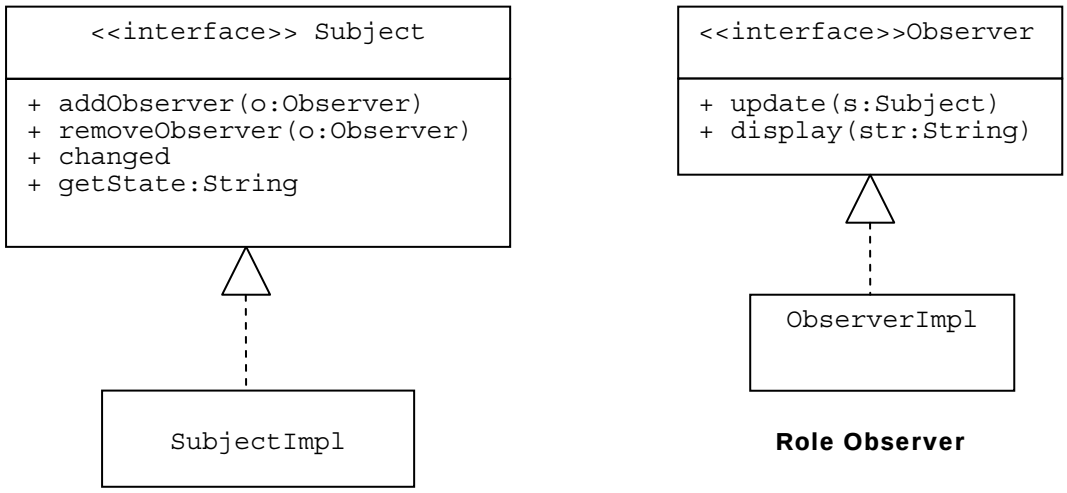

Fig – 5.1

The role `Subject` has to be linked to `Point` and `Line` and role `Observer` has to be linked to `Screen`. Now let us see how aspect binding is achieved using the sample aspect binding class presented earlier. Please note that here we will be using only `@Wrap` and `@Bind` annotations. The functionality of `@After`, `@Before` and `@Insert` is not implemented. It is directly hardwired into the metaobject protocol.

```
class ColorObserver{
      @Wrap("Point") @Bind("Subject") public class PointSubject{ }

      @Wrap("Line") @Bind("Subject") public class LineSubject{ }

      @Wrap("Screen") @Bind("Observer")public class ScreenObserver{ }
}
```

Code-5.8 Aspect Binding (class ColorObserver)

The structural cuts are defined based on the presence of `@Wrap` annotation and `@Bind` annotation. The code for selecting the classes based on `@Wrap` and `@Bind` annotation is the same as given in Code-5.4 and Code-5.6 respectively.

Once the cuts are defined, the selected classes are handled in the respective `SMetaobject`. The metaobject for the generation of wrappers is as follows (Code-5.9):

```
public class WrapperGenerator implements SMetaobject {
      public void handleClass(RClass aClass) {

          Annotation annotation =
            aClass.getAnnotation("wrappersForCollaboration.Wrap");

          MemberValue memberValue =
                               annotation.getMemberValue("value");

          String weeName = ((StringMemberValue)
                                          memberValue).getValue();

          String str=new String(aClass.getSimpleName());

          WrapperAPI wapi = new WrapperAPI();
            try {
                  wapi.wrap(str,weeName);
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }
}
```

Code-5.9 Metaobject for the generation of wrapper classes

The metaobject for binding of aspects and components is as shown in Code-5.10. The checking for role has been applied here because the implementation for annotations `@After`, `@Before`, `@Around` and `@Insert` have not been done. Once the checking for these annotations is incorporated as new structural links, the checking of roles can be removed.

Note: This implementation is fairly unsafe with the current implementation of Reflex. Indeed it assumes that the `WrapperGenerator` metaobject applies before the `Binding` metaobject but there is currently no way to guarantee this. Although, there are rules to order the application of behavioral links [22] but such rules are not available for structural links. Our work has pointed out this limitation of the current version of Reflex.

```
public class Binding implements SMetaobject{
      public void handleClass(RClass aClass) {

        Annotation annotation =
            aClass.getAnnotation("wrappersForCollaboration.Bind");

        MemberValue memberValue = annotation.getMemberValue("value");

        String roleName = ((StringMemberValue)
                                        memberValue).getValue();

        String str=new String(aClass.getSimpleName());

        WrapperAPI wapi = new WrapperAPI();
            try {
                  if(roleName.equals("Subject")){
                        wapi.assign(roleName,str);
                        wf.insertMethod(str,"public String
                              getState(){return \"Point Color\" +
                                    wrappee.getColor();}");
                        wf.after("setColor","{changed();}",str);
                  }
                  else{
                        wf.assign(roleName,str);
                        wf.insertMethod(str,"public void
                              update(wrappersForCollaboration.Subject
                              s){wrappee.display(\"Color changed\" +
                              s.getState();}");
                  }

            } catch (NotFoundException e) {
                  e.printStackTrace();
            } catch (CannotCompileException e) {
                  e.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```

Code-5.10 Metaobject for binding of aspects and components

5.4 Chapter Summary

This chapter has given an insight into the integration process with Reflex. A sample Aspect Binding class has been designed, using Java annotations, which gives a feel of how the wrapper API can be used to build a language like Caesar based on the idea of wrappers. This binding class also shows how wrappers can be built implicitly for integrating aspects and components. Then a small example has been built which shows how the binding class can be used to implement collaborations (or a collaboration).

Chapter 6. CONCLUSION

Reflex is an open reflective extension of Java which was evolved into a versatile kernel for building AOP languages. The objective of this AOP kernel is to support and combine a large range of Aspect approaches. Reflex has mainly focused on “behavior-oriented” aspect languages. In order to explore the versatility of the approach, it is necessary to consider languages which are interested in the structure of aspects, like Caesar. Caesar is based on two ideas – wrappers which are a mid-level abstraction and collaboration, which come into play when the aspect has to be bound to the base program. Motivated by the Caesar model, I worked with the idea of wrappers and collaboration to build a language which integrates aspects and components.

The wrapper API was built using Javassist. This API supports generation of specialized wrapper classes for integrating aspects and components. Aspects and components are considered two separate entities, and wrappers integrate these two at load time, which facilitates the reuse of aspects for different components. The API contains methods for creating wrapper classes either from scratch, shown in chapter 4, or converting existing classes into wrapper classes. This facility of converting existing classes into wrapper classes helps deal with inner classes which can be converted into wrappers, which is shown in chapter 5. For binding aspects and components, two approaches are available – the inheritance approach and the forwarding approach. The API also contains methods for modifying / changing the behavior of the wrappee and for the insertion of a method or an instance variable. Three examples give an overview of the usage of the methods available in the wrapper API and also show how these wrappers can be put to various uses. Basically, wrappers can be used for changing the behavior of the wrappee, for implementing aspects, in particular, aspects structured as collaborations.

After the API was developed, it was integrated with Reflex. The Reflex API for structural modifications was used for the integration process, which is built on the lines of the Javassist API. The integration process highlighted some of the loop holes in Reflex. The structural API was not complete enough to generate wrapper classes. Hence at some places during the integration process, the Javassist API had to be used. Also there was a problem with implementing advice method calls (before, after and around). Implementing these calls was considered as a behavioral modification, not allowed in the S-link application phase. Reflex had to be patched in order to still be able to make these modifications. This points out that the definition of the S-link application phase has to be refined.

Once the integration was done, a sample aspect binding class was designed to show how the wrapper API can be used to build a Caesar-like aspect-oriented language. This sample aspect binding class uses Java annotations for implicit generation of wrappers and for binding aspects and components. The idea of inner classes was explored to design this class. An example is also shown which uses this sample aspect binding class to implement the Observer Pattern.

One more thing that surfaced while working with the Reflex API for structural modifications was that *rules* did not apply for structural links. Working with the concept of annotations and structural cuts highlighted the fact that rules were indeed useful in the context of structural links too.

Future Work

At the API level

Dynamicity - At present the wrapper API is very static in nature. It would be helpful if it can be made dynamic by introducing Dynamic Binding of Roles and Components at runtime**.** In the present scenario, when talking about collaboration, binding of roles and components is static. Once the role is bound to the component, it cannot be changed. Hence there has to be a way to deal with dynamic binding (or reconfiguration) of roles and components at runtime.

Extending the wrapper API – At present, the API supports inheritance and forwarding approaches for binding of aspects and components. It could be extended to incorporate Superimposition.

At the language level

Develop a full implementation of Caesar Language - At the moment, only a very small part of the Caesar language has been implemented. The idea of Aspect Collaboration Interfaces (ACIs) [3], which is the main feature of Caesar model, has not been introduced. The ACI is implemented using inner interfaces and then it is linked to aspect implementation class which uses the concept of inner classes. Working with inner interfaces and inner classes and then linking these inner interfaces to inner classes was quite a tricky job. Many technical issues had to be solved there. Therefore it was decided to first work only with the idea of inner classes. Hence, the idea of inner classes and annotations were used to design the aspect binding class which is responsible for binding of aspect implementation class and components and also for the implicit generation of wrappers. This idea of annotations can be extended to design the ACIs as shown in Fig-6.1.

The annotations `@provided` and `@expected` can be used to specify which methods are provided by the aspect and which methods are expected from the wrappee and accordingly they can be implemented in aspect implementation or aspect binding classes.

```
interface ObserverProtocol{
      interface Subject{
            @provided void addObserver(Observer o);
            @provided void removeObserver(Observer o);
            @provided void changed();
            @expected String getState();
      }
      interface Observer{
            @expected void notify(Subject s);
      }
}
```

Fig-6.1

Design a new Higher Level Language - Based on the implementation of Caesar language, a new higher level language can be developed which will fully integrate aspects and components and which will also incorporate all the ideas put forward by Mira Mezini et.al. At present, only a simple Java Component (Java interface and a Java class file) has been considered for developing the API. An actual component can be a collaboration of many sub components. It would be interesting to incorporate this idea of component into the language.

REFERENCES

1. Annotations (http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html), Graham O'Regan, 14 Jan 2004, Sun Java Software

2. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. *An Overview of AspectJ*. In Jørgen Lindskov Knudsen, editor, ECOOP 2001 - Object-Oriented Programming, 15th European Conference, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

3. Mira Mezini and Klaus Ostermann. *Conquering aspects with Caesar*. In Mehmet Aksit, editor, Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), pages 90–99, Boston, Massachusetts, USA, March 2003. ACM Press.

4. R. Cardone and C. Lin. *Comparing Frameworks and Layered Refinement*. In Proceedings of the 23rd International Conference on Software Engineering, pages 285-294, Toronto, Canada, May 2001. IEEE Computer Society Press.

5. A.M Reina and J. Torres. *Components + Aspects: A General Review*.

6. Renaud Pawlak, Laurence Duchein, Lionel Seinturier. *Dynamic Wrappers: Handling the Composition Issue with JAC*. IEEE 2001

7. Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. *An Adaptive Object Model with Dynamic Role Binding*. In Proceedings of the 27th International Conference on Software Engineering, pages 166-175, St. Louis, MO, USA, May 2005. ACM Press.

8. Geofrey A. Pascoe. *Encapsulators: A New Software Paradigm in Smalltalk-80*. In Conference Proceedings on Object-oriented programming systems, languages and applications , pages 341-346, Portland, OR, USA, 1986. ACM Press.

9. John Brant, Brian Foote, Ralph E. Johnson and Donald Roberts. *Wrappers to the Rescue*. In Proceedings of the 12th European Conference on Object-Oriented Programming (July 20 - 24, 1998). E. Jul, Ed. Lecture Notes In Computer Science, vol. 1445. Springer-Verlag, London, 396-417.

10. Urs Hölzle. *Integrating Independently-Developed Components in Object-Oriented Languages*. In Proceedings of the 7th European Conference on Object-Oriented Programming (July 26 - 30, 1993). O. Nierstrasz, Ed. Lecture Notes In Computer Science, vol. 707. Springer-Verlag, London, 36-56.

11. Tetsuo Tamai. *Evolvable Programming based on Collaboration-Field and Role Model*. In International Workshop on Principles of Software Evolution (IWPSE 02), pages 1-5, Orlando, Florida, 2002. ACM Press. Invited Paper.
12. Jan Hannemann and Gregor Kiczales. *Design Pattern Implementation in Java and AspectJ*. In OOPSLA 2002, Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 161-173.

13. Davy Suvée, Wim Vanderperren, and Viviane Jonckers. *JAsCo: an Aspect-Oriented approach tailored for Component-Based Software Development*. In Mehmet Aksit, editor, Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), pages 21-29, Boston, Massachusettes, USA, March 2003. ACM Press.

14. Mira Mezini and Klaus Ostermann. *Integrating Independent Components with On-Demand Remodularization*. In OOPSLA 2002, Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 52-67, Seattle, Washington, USA, October 2003. ACM Press.

15. Johan Ovlinger. *Modular Programming with Aspectual Collaborations*. OOPSLA 2002. ACM.

16. Roman Pichler, Klaus Ostermann, and Mira Mezini. *On Aspectualizing Component Models*. Software Practice and Experience, 33(10):957-974, 2003.

17. Matthew Flatt and Matthias Felleisen. *Units: Cool modules for HOT languages*. In Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, pages 236-248, Montreal, Canada, May 1998. ACM SIGPLAN Notices, 33(5).

18. Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. *JAC: A Flexible Solution for Aspect-Oriented Programming in Java*. In A. Yonezawa and S. Matsuoka, editors, Meta-Level Architectures and Separation of Crosscutting Concerns, Third International Conference, Reflection 2001, volume 2192 of Lecture Notes in Computer Science, pages 1-24, Kyoto, Japan, September 2001. Springer-Verlag.

19. S. McDirmid and W.C. Hsieh. *Aspect-Oriented Programming with Jiazzi*. In Mehmet Aksit, editor, Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), pages 70-79, Boston, Massachusettes, USA, March 2003. ACM Press.

20. E. Tanter, N.M.N. Bouraqadi-Saâdani, and J. Noyé. *Reflex -Towards an Open Reflective Extension of Java*. In A. Yonezawa and S. Matsuoka, editors, Meta-Level Architectures and Separation of Crosscutting Concerns, Third International Conference, Reflection 2001, volume 2192 of Lecture Notes in Computer Science, pages 25-43, Kyoto, Japan, September 2001. Springer-Verlag.

21. Éric Tanter and J. Noyé. *A Versatile Kernel for Multi-Language AOP*. In Robert Glück and Michael Lowry, editors, Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05), Lecture Notes in Computer Science, Tallin, Estonia, September/October 2005. Springer-Verlag. To appear.

22. Leonardo Rodriguez, Eric Tanter, and Jacques Noyé. *Supporting Dynamic Crosscutting with Partial Behavioral Reflection: a case study*. In Gonzalo Navarro, editor, 14th International Conference of the Chilean Computer Society, pages 48-58, Arica, Chile, November 2004.

23. Éric Tanter and Jacques Noye. *A Versatile Kernel for Language-Neutral AOP*.

24. Éric Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, École des Mines de Nantes, Université de Nantes, and University of Chile, November 2004.

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

26. Yannis Smaragdakis and Don Batory. *Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs*. ACM Transactions on Software Engineering and Methodology, 11(2):215 255, April 2002.

27. Stephen H. Edwards, Gulam Shakir, Murali Sitaram, Bruce W. Weide and Joseph Hollingsworth. *A Framework for Detecting Interface Violations in Component-Based Software*. IEEE Computer Society Proceedings 5th International Conference on Software Reuse Victoria, Canada (Jun. 1998).

28. P. Popov, Stringini S. Riddle and A. Romanovsky. *Protective Wrapping of OTS Components*. Presented at 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction, Toronto, 2001.

29. S. Chiba. *Load-time Structural Reflection in Java*. In E. Bertino, editor, Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2000), number 1850 in Lecture Notes in Computer Science, pages 313-336, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.

30. Jeniffer M. Haddox, Gregory M. Kapfhammer, Christoph C. Michael. *An Approach for Understanding and Testing Third Party Software Components*. 2002RM-037.

31. Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. *Jiazzi: New-Age Components for Old Fashioned Java*. In Proceedings of the 16th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications (OOPSLA 01).

32. Sergei Kojarski, Karl Lieberherr, David H. Lorenz, and Robert Hirschfeld. *Aspectual Reflection*. In AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT 03), Boston, USA, 2003.

33. Nigamanth Sridhar and Jason O. Hallstrom. *Generating Configurable Containers for Component-Based Software.* Proceedings of the 6th Component-Based Software Engineering workshop (CBSE6) at ICSE 2003.

34. Javassist Home Page (http://www.csg.is.titech.ac.jp/~chiba/javassist/)

35. Annotations in Tiger, Part 1: Add Metadata to Java Code (http://www-128.ibm.com/developerworks/java/library/j-annotate1/index.html#N100D7). Author/Editor, O'Reilly Media, Inc. 02 Sep 2004

37. AOP@Work: Introducing AspectJ5 (http://www-128.ibm.com/developerworks/java/library/j-aopwork8/) AspectJ project leader and IBM senior technical staff member, IBM. 12 July 2005

38 Johannes Maria Zaha, Alexander Keiblinger, Klaus Turowski. *Component Market Specification Demand and Standardized Specification Of Business Components*. 1st International workshop Component Based Business Information Systems Engineering September 2nd, 2003 - Geneva, Switzerland.

39. Rémi Douence, Pascal Fradet, and Mario Südholt. *Trace-Based Aspects*. In Filman et al. [2005], pages 201–217.