An Exploratory Study of CaesarJ Based on Implementations of the Gang-of-Four patterns

TECHNICAL REPORT FCT-UNL-DI-SWE-2008-01

Draft, June 2008

Edgar Sousa
Miguel P. Monteiro

CONTENTS

FIGURES

CODE LISTINGS

An Exploratory Study of CaesarJ Based on Implementations of the Gang-of-Four patterns

Technical Report – draft, June 2008

Edgar Sousa
Departamento de Informática
Escola de Engenharia
Universidade do Minho
PORTUGAL
edgar@di.uminho.pt

Miguel Pessoa Monteiro
Departamento de Informática
Faculdade de Ciência e Tecnologia
Universidade Nova de Lisboa
PORTUGAL
mmonteiro@di.fct.unl.pt

Abstract

This report presents the results of an exploratory study of the aspect-oriented programming language CaesarJ. The study is based on implementations of seven Gang-of-Four design patterns. The report describes the implementations and then provides a short analysis. A preliminary assessment of CaesarJ is made in which AspectJ is used as a basis for comparison.

1. INTRODUCTION

This report documents the results of an effort to perform an initial assessment of the capabilities of the CaesarJ language, in which several examples in the CaesarJ programming language [4] were developed. The focus of the work was on the constructs and mechanisms that classify CaesarJ as aspect-oriented, as well as to its support separation of concerns and module reuse. To this effect, seven of the well-known Gang-of-Four design patterns [12] were developed in CaesarJ. This report describes the implementations developed and refers to AspectJ implementations of the same patterns for the purpose of a comparison [1] that uses as criteria flexibility of support for modularity and separation of concerns and direct support to a given effect.

The complete code is available at the following URL:

`http://ctp.di.fct.unl.pt/~mpm/CaesarJGoFv0.5.rar`

1.1. Motivation

Many languages for *aspect-oriented programming* (AOP) [20] were proposed in recent years [8]. For each language it is usual for one or several publications to be available, describing its distinguishing characteristics, usually with respect to AspectJ. However, reports describing the use of an AOP language in practice are less frequent. In recent years, the CaesarJ language attracted attention among researchers of Aspect-Oriented Software Development, due to its promising features and the claims stated of the advantages over AspectJ in relation to flexible support for modularity and separation of concerns [24]. Few studies are available on the use of CaesarJ in practice, the one by Schwaninger et al [33] being the only exception of which we are aware. This scarcity motivates the work described in this report.

To adequately assess the capabilities of CaesarJ, a selection of the Gang-of-Four patterns is used as a case study. Design patterns are a distillation of many real systems for the purpose of cataloguing and categorizing common programming and design practice. The most well-known patterns are the 23 Gang-of-four (GoF) patterns, which propose flexible solutions for many

design and structural issues [12]. A repository of implementations of the GoF patterns comprises an interesting case study for the practical assessment of a programming language. In the GoF book, the examples are coded in languages that were mainstream (mostly C++) at the time in which the book was published. It was later noticed that the patterns provide many insights on the strengths and weaknesses of languages, as well as providing hints as to what language features could overcome the limitations [6]. The GoF patterns were also used as case studies for research on AOP languages from its early days [21]. See for instance the page on Subject-oriented Programming (SOP) in which several GoF pattern are used as case studies to assess and illustrate the advantages of SOP over traditional techniques[1]. Use of the GoF includes showcase some AOP languages [16], illustrate the advantages of a given AOP language over some other language used as benchmark [26][32], and as a basis for tutorials [27]. Probably the most cited example is that of Hannemann and Kiczales [16], in which *all* 23 GoF patterns were implemented in both Java and AspectJ and the corresponding source code was made available online[2]. A substantial part of the Miles' book on AspectJ techniques [27] is based on those examples. In addition, public availability of the examples opened the way for its use in further research [13][30]. Use of the GoF patterns as a case study also has the advantage that each pattern can be tackled separately and each individual example can be kept simple, which eases the task of someone approaching the examples.

Despite the benefits brought by that example and the opportunities it provided, presently most AOP languages lack its own repository of GoF implementations. To our knowledge, just one complete repository was developed in addition to that of AspectJ – using the EOS language [32] – and it was not made publicly available. If we keep in mind that there are now many AOP languages available [8], this gap is more glaring. We believe that it would be beneficial if a wider range of AOP languages was used to develop implementations of patterns such as the GoF. If such repositories were available, they would provide case studies for various kinds of research as well as facilitating various kinds of comparative studies and assessments. This report presents example for seven patterns.

The rest of this report is structured as follows. Section 2 describes the main mechanisms and constructs of CaesarJ. Section 2.9.1 describes the approach taken for the study. Section 4 focuses on the particular case of the *Observer* pattern, which is used to illustrate how CaesarJ can be used to develop a reusable software component. Section 5 covers the remaining patterns covered by the study. Section 6 provides a discussion on the results obtained. Section 7 provides suggestions for future work. Finally, section 8 concludes this report.

2. THE CAESARJ LANGUAGE

In this section, an overview is provided of some of the mechanisms of CaesarJ. CaesarJ shares some important similarities with AspectJ, presently the most popular and widely used aspect-oriented programming language. However, CaesarJ also has some important differences from AspectJ. Since most potential readers of this report are likely to be familiar with AspectJ, this reports assumes such familiarity and outlines the main characteristics of CaesarJ by underlining the differences relative to AspectJ [1] [19]. For a comprehensive and up to date overview of CaesarJ, see Aracic et al [5].

Like AspectJ, CaesarJ is also a backwards compatible extension to Java. However, CaesarJ extends Java 2, whereas AspectJ is kept up to date with the latest versions of Java. AspectJ is already available as an extension to Java 6. CaesarJ does not support constructs introduced to

[1] http://www.research.ibm.com/sop/sopdpats.htm

[2] http://www.cs.ubc.ca/labs/spl/projects/aodps.html

Java in versions after Java 2, such as annotations, generic types and the generalized for loop. CaesarJ uses a joinpoint model similar to that of AspectJ, with the exception that AspectJ's **if** pointcut designator is not supported. Unlike AspectJ, however, CaesarJ supports *dynamic deployment* of advice, meaning that all advice of a CaesarJ aspect can be activated and deactivated. In addition, CaesarJ supports *deploy on object*, i.e. the possibility to constrain the captured joinpoints to those that originate from a given target object.

In addition to plain-Java classes, CaesarJ provides a second kind of class-like module, the *CaesarJ class* **that** is roughly equivalent to the aspect modules of AspectJ. CaesarJ classes are represented through the **cclass** keyword. An important feature of CaesarJ is the inner classes of **cclass** modules, which are also declared and defined through the **cclass** keyword.

2.1. Structure of a CaesarJ component

With CaesarJ, the emphasis is not so much on *aspects* (and classes) as on *components*. The features that CaesarJ shares with AspectJ are mainly regarded as a means to compose modules and components. CaesarJ was designed to support the development and integration of software components and to the flexible reuse of the modules that comprise the structure of a component. In this regard, CaesarJ goes further than mainstream object-oriented (OO) languages and older AO languages such as AspectJ. Figure 1 outlines the structure of a CaesarJ component. The names shown are abstract and not related to any particular case.

The design of CaesarJ takes into account that component integration typically comprises several parts which developers wish to keep separate and be able to evolve separately. In CaesarJ it is possible to represent the design-level structure of a component to some extent, by means of a CaesarJ class. One such class is shown at the top of Figure 1 and usually called *collaboration interface* (section 2.4).

CaesarJ takes into account that there are two primary parts in the structure of a component: (1) the internal implementation of the component itself and (2) the glue that integrates it to a specific application. Concrete implementations of components are placed in *CaesarJ implementations* (section 2.5), represented at the left side of Figure 1. CaesarJ supports polymorphism in such a way that it is possible to switch from one implementation to another without the need to perform invasive changes on the remaining parts of the system. In addition to collaboration interfaces and CaesarJ implementations, one often needs some glue that binds a component to a specific application. CaesarJ provides support to this kind of glue through *CaesarJ bindings* (section 2.6), represented at the right side of Figure 1.

As regards component integration, it is important to note that any component comprises a set of abstractions that must somehow be mapped to the structural elements of the application. The glue becomes particularly important in cases in which the application does not have first-class representations (typically classes) of the abstractions on which the component is based. One example of this problem, mentioned by Mezini and Ostermann [26], is that of a component that supports the notification of changes in lines and/or line segments from an application. If the application does not include a class representing lines but just individual points, some glue must be created to map pairs of points to lines. In the context of CaesarJ, this mapping is called *on demand remodularization* [25]. In CaesarJ, it is supported by a mechanism of *wrappers* (section 2.6) that is represented at the bottom right of Figure 1.

Finally, all implementations and bindings are composed together by declaring a concrete CaesarJ class called *weavelet* (section 2.7) that uses *mixin composition* to integrate all modules into a component (section 2.7.1). This module is shown at the bottom left of Figure 1.

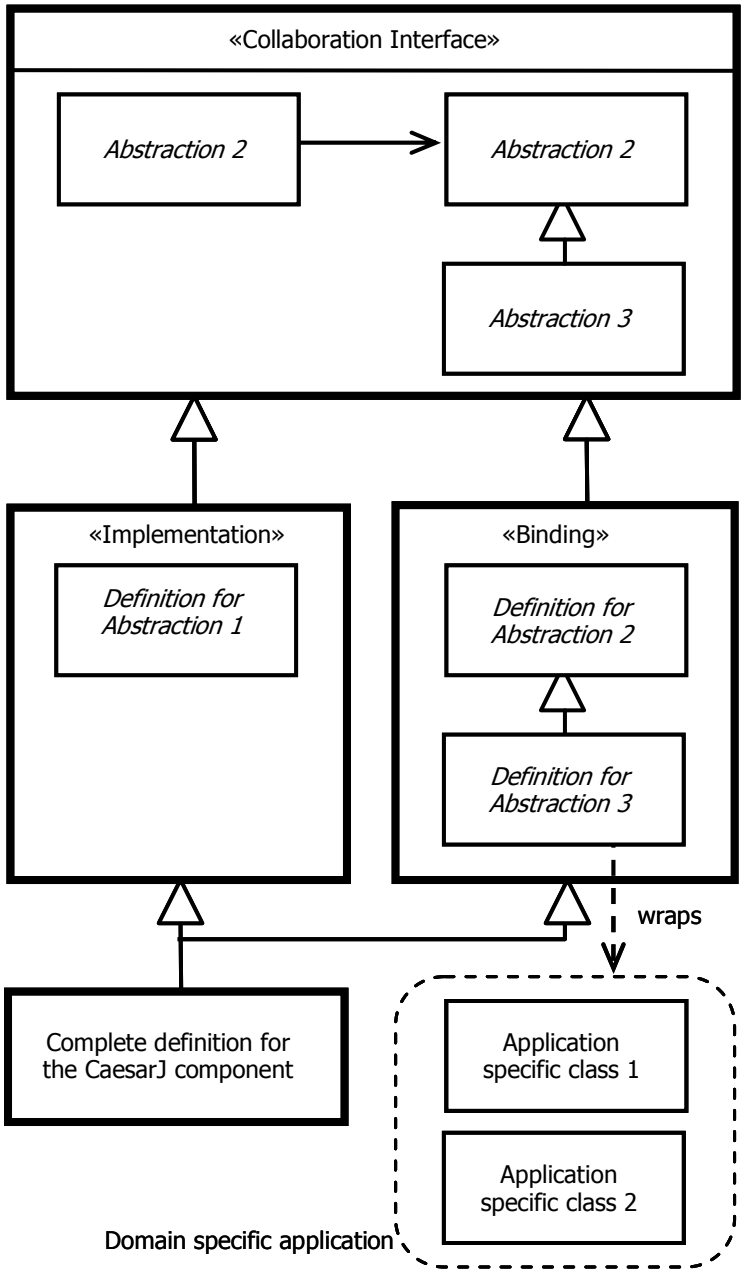

Figure 1. General structure of CaesarJ component.

The component structure supported by CaesarJ provides a clear guide as to where each piece of code is to be placed and suggests the following process:

1. First, analyze of the original example, with the help of a class diagram. In some cases, the abstract part of the structure maps directly into a CI.
2. Next, place all implementation code not dependent of case-specific types in the CaesarJ implementation.
3. Place of all code depending on case-specific types in the CaesarJ binding.
4. Create the complete component through an empty CaesarJ class (the weavelet) that extends the implementation and the binding.

2.2. Virtual classes

Virtual classes [23] comprise the capability to treat inner, nested classes, polymorphically. We give to the term *virtual* the same meaning as in the context of the C++ language, i.e. class members that can be overridden in subclasses and whose actual implementation is *dynamically bound*, or *late bound*, according to the actual type of the object at runtime. In plain Java, methods are virtual by default, unless marked by the **final** keyword. Plain Java inner classes, however, are not virtual. They cannot be overridden and are not subject to dynamic binding. If an inner class is declared in a subclass with the same name as the inner class of the super-class, this gives rise to a *shadowing* phenomenon. Shadowing occurs when a variable or member declared in a given scope hides a different variable or member with an identical name declared in a broader, enclosing scope. Shadowing occurs most often with fields and variables and is generally considered bad-style, as it results in obscure code. For this reason, programmers strive to avoid them.

CaesarJ supports virtual classes, with the consequence that the name of an inner class does not uniquely identify a specific class, as is the case of mainstream OO languages. Instead, any CaesarJ class nested within another CaesarJ class is *virtual*, i.e., its name can be dynamically bound to multiple, different classes, depending on the specific instance of the enclosing CaesarJ class. Thus, CaesarJ can be said to support *class name polymorphism* in addition to the *object reference polymorphism* of mainstream OO languages. This provides opportunities for reuse that are absent from traditional languages, as a concrete class name comprises an *extension point*.

Any class nested within a CaesarJ class is a virtual class and thus can be dynamically bound to different concrete classes in different circumstances, the same way method calls are late bound to the most specific implementation in plain Java.

Traditionally, instantiation of inner classes in Java is done the same way as top-level classes, using the **new** keyword followed by *TopLevelClass.InnerClass()*. However, for instantiating virtual classes, the **new** keyword is placed *after* the complete path to the instance of enclosing type (i.e., family object):

```
object = OuterCClass.new InnerCClass();
```

2.3. Family polymorphism

A top-level CaesarJ class is the construct through which CaesarJ supports a mechanism termed *family polimorphism* [11] Family polymorphism is the ability to group a set of distinct, collaborating classes into a larger class, such that the member classes, and hence their instances, are uniquely owned by the enclosing *instance*. The type system enables the set of inner classes to be refined but at the same time statically ensures that objects from different families are not confused. In CaesarJ, refinement of member classes is carried out by the virtual class mechanism. The refinement is also performed in such a way that it is hidden from external clients, enabling the whole group to be handled polymorphically.

Many problems are described in terms of a family of distinct, collaborating classes that can be realized through many different implementations. In modern software, cases often arise in which several such implementations co-exist in the same system. This gives rise to the problem of how to ensure consistency. Each of the collaborating classes can be refined along parallel inheritance chains, which results in multiple possible combinations of class implementations. Some combinations are consistent and therefore should be allowed by the type system, while other combinations are inconsistent and should not be permitted. In his seminal work on family

polymorphism [11], Ernst analyses how the type systems of various mainstream languages cope with this problem.

Mainstream type systems either allow too much or block too much. As a result, either safety or flexibility is lost. If it leans on the side of safety, it blocks many situations that would be safe and consistent, thus compromising reuse. On the other hand, if the type checker leans on the side of flexibility, it also allows for many unsound or erroneous combinations. Neither scenario is satisfactory. Programmers using mainstream languages do not enjoy the support of the type checker and therefore must deal with those cases manually, through techniques such as the Abstract Factory pattern [12].

Ernst notes that in traditional OO languages, an explicit representation of class families is lacking. Mainstream languages are not expressive enough to provide type systems with enough information to provide adequate support to multi-object structures [11]. His proposal to cope with these problems is a language mechanism comprising *family classes* and *family objects*. A family class specifies a family of collaborating classes in abstract terms. There are many different possible variant implementations of the family, each being an instance of the family class. These instances – the family objects – act as repositories of concrete classes that comprise one possible implementation. The type system relies on the identity of the family object, meaning that two different family objects are considered by the type checker as two different implementations, whose members are not allowed to mix. Thus, the families of implementation types are associated with the *instances* of the class family, not with class family itself. As a consequence of the type checker relying on instances, the family object must always be passed as an argument along with the instance types of the family. In order for the type checker to guarantee that the enclosing (family) object is the same in all situations, some situations associated to the handling of object references are prohibited. For instance, an object is not even "equal to itself" in a scenario in which it is passed twice to a method, in the form of two different arguments. This is due to the existence of multithreading in which the referenced object may be changed between two different accesses. For this reason, the reference to the family object must be declared **final** and propagated as such from its definition to any point in the program that uses parts of the family.

In CaesarJ, any top-level CaesarJ class is a family class. Any nested CaesarJ class is a virtual class that can be overridden and whose name is subject to polymorphism. This way, CaesarJ enables the developer to incrementally refine class collaborations.

2.4. Collaboration interfaces

In CaesarJ, a *collaboration interface* (CI) [25] is an approach proposed by the authors of CaesarJ, in which an abstract CaesarJ class is used to specify, in abstract terms, a collaboration between several abstractions. CIs represent design-level information akin to that represented by the abstract classes in many class diagrams of the GoF patterns. In CaesarJ, such design knowledge can directly be represented through abstract CaesarJ classes. CIs are a characteristic of CaesarJ, not of languages supporting family polymorphism in general.

A collaboration of abstractions usually involves multiple, different classes. In mainstream OO languages, this means that collaborations are a structural manifestation of crosscutting. In CaesarJ, participants in collaborations are declared, though not necessarily defined, in the CI as inner, nested, virtual classes. Only top-level CaesarJ classes can be CIs. Each different system of concrete classes that represents the collaboration comprises one of possibly many implementations of the CI. In CaesarJ, the elements of a system of concrete classes can be polymorphically bound to the names of the nested classes declared in the CI.

Listing 8 shows the CI for the Observer pattern. This CI represents the abstract roles of Observer (note that different operations names are used in each case). In addition to specifying the internal structure a software component, CIs can also be said to specify its interface. Keeping in mind that it is possible to approach a collaboration as an aspect, it is worth noting that the interface of a CaesarJ aspect is quite different from that of AspectJ aspects, as proposed by Gudmundson and Kiczales [15]. The CaesarJ approach focuses on structure, while the AspectJ approach focuses on dynamic events as represented by abstract pointcuts.

In software component engineering, the concepts of *provided* and *expected* interfaces, or contracts, are often used. The *provided* part represents what the component provides to clients, i.e., what a client of the component can expect from it when using it. The *expected* part represents the assumptions that a component can make from its environment and that can be used in its implementation. When they were initially proposed, CIs used **required** and **expected** keywords, to represent these concepts. These keywords were used to tag the operations declared in the CI. These were later dropped as it was deemed not general enough: each case could find a different variant of what is provided and what is expected [5]. However, this division of the parts of a component is still present at the conceptual level, though no longer explicitly represented in CaesarJ source code. This division also provides the motivation for distinguishing between the *bindings* and *implementations* parts of a CaesarJ component.

2.5. CaesarJ implementations

A CaesarJ *implementation* is a top-level CaesarJ class that encloses all elements of the concrete implementation of the component that are potentially reusable across multiple cases and scenarios. It inherits from a CI when one such exists. Though it usually specifies concrete state and behaviour, it is often abstract. Typically, a CaesarJ implementation provides implementations to some, though not necessarily all, the inner classes declared in the CI. Conceptually, these parts relate to the *provided* part of the component (section 2.4). A CaesarJ implementation carries out the same role as an abstract aspect in AspectJ. Listing 10 shows one example of a CaesarJ implementation. Note that there is no need for a CaesarJ implementation to define all members declared in the CI. Some of the members may be defined in other CaesarJ classes that also inherit from the CI. All is required is that a definition is provided along one of the inheritance paths that take the CI as root, so that all names are resolved in the weavelet.

2.6. CaesarJ bindings and wrappers

A *CaesarJ binding* is a top-level CaesarJ class that inherits from the CI (when one such exists) and encloses the logic that glues the component to a specific application. Thus, application-specific elements should be placed in a CaesarJ binding. A CaesarJ binding corresponds to the *expected* part of the component (section 2.4). It performs the same role as a concrete sub-aspect in AspectJ.

Code within a CaesarJ binding must map the nested, virtual classes declared in the CI to the concrete classes of the application. In CaesarJ, the primary mechanisms to perform an actual mapping are *wrappers*. Virtual classes can be declared to *wrap* one or several objects of arbitrary types, which become its *wrappees*. Through access to the public methods and fields of the wrappees, the virtual class can extend the behaviour of the wrappees in order to adapt them to the component. Wrappers hold any extra members that may be necessary for the purposes of the integration between the component and the application. Listing 13 shows a CaesarJ binding with wrappers.

The binding also handles the cases in which the application does not have classes that directly correspond to the abstract roles declared in the CI, i.e., when the application lacks some

abstraction that plays a role in the component. In such cases, the binding defines nested classes that hold together the various objects that comprise the missing abstraction (cf. *on-demand remodularization* [25]).

2.7. CaesarJ weavelets: synthesizing a complete component

Both implementations and bindings are subclasses of the CI. Thus, we need a mechanism to compose them to yield the complete, unified component. The mechanism provided by CaesarJ is a form of *mixin composition* [7].

2.7.1 Mixin composition

The main idea of mixins is to specify additional functionality to not just one, but to an open-ended set of existing modules in a transparent and non-invasive way. Mixins are also known as *abstract subclasses* [7]. In this context, mixins are approached as a mechanism to specify a subclass that can be used to extend more than one class. Some programming languages can emulate this effect, including C++ and CLOS. C++ can emulate mixins using parameterized inheritance, i.e., a template subclass that gets a given type as a parameter and also *inherits* from the parameter type. Listing 1 shows a C++ example of the technique. In the C++ community, the technique is also known as *traits classes* [31].

```
template <class Super> class Mixin : public Super {
  // mixin body
};
```

Listing 1. Emulating mixins in C++

Though Java 5 supports generic types that are akin to C++ templates, the Java variant is not expressive enough to emulate mixins. In Java 5, it is not possible to use a type parameter in the **extends** clause. In the example shown in Listing 2, if the comments in the first line are removed, the Java compiler issues an error.

```
public class Template<T> /* extends T */ {
  // class body
}
```

Listing 2. Hipothetical emulation of mixins in Java 5

In Java, each subclass presets the class that it extends. It is not possible to pick a Java class that extends some Java class with some additional functionality and reuse it to extend a different Java class with the same functionality. In such cases, programmers must create a different subclass for each class they want to extend, which often results in much duplication. Another option is to resort to the Decorator pattern [12].

Mixin composition is often approached as a variant of inheritance, and described in terms of inheritance, more specifically in the context of multiple inheritance. However, mixin composition can take other forms that do not correspond to inheritance. For instance, the inter-type declarations of AspectJ are a form of mixin composition.

2.7.2 Mixin composition in CaesarJ

In CaesarJ, a component can be created by composing several CaesarJ classes. Using mixin composition, different components can be composed to build more complex components without compromising the independence of each component. Mixin composition is the way by which CaesarJ creates complete components out of various parts. The CI provides the general

framework, the CaesarJ implementation provides the realization of specific implementation decisions and the binding provides the glue to the other components and/or applications.

Listing 11 shows a weavelet for one example of Observer.

2.8. Status of current support for CaesarJ

The version of CaesarJ used comprises eclipse 3.2.2 [2] with the CaesarJ Development Tools (CJDT) plug-in [4], version 0.8.7. Later versions of eclipse do not seem to be compatible with CJDT.

2.8.1 Compiler efficiency

The CaesarJ compiler was noticeably slow even with relatively small code bases. For this reason we found it necessary to disable the automatic build option in the eclipse environment. The program launcher was also very slow to launch a run.

2.8.2 Limitations

Package-level visibility of CaesarJ class members is not allowed. If the default package is used, the CaesarJ compiler refuses to compile the project. Programmers are advised to use separate packages for each abstract implementation of CIs, to avoid conflicts with the scope of pointcut definitions.

A limitation of CaesarJ that placed some constraints on some programming solutions is that CaesarJ class cannot extend regular Java classes. Presently, the best available option to compose additional functionality to plain Java classes seems to be by means of wrappers.

One other absent but missed feature is arrays of CaesarJ's objects. The workaround used is to use references of type java.lang.Object and perform the appropriate casts to the desired type.

2.8.3 Bugs found

None of the CJDT problems found is really critical. The most notable defect we found is attributable to the compiler. Around advice can't be placed on CaesarJ classes that implements interface PerThisDeployable (or sub-classes of such classes). The CaesarJ compiler fails by throwing ArrayIndexOutOfBoundsException. The following code illustrates the problem:

```
public cclass SomeClass implements PerThisDeployable{
  //...
  around() : somePointcut(){
    //do something
  }
}
```

After some time, we noticed that syntax highlighting behaves strangely, as well as the graphical representation of pointcuts.

2.9. Preliminary comparison between AspectJ and CaesarJ

2.9.1 Asymmetry of the programming models

Both AspectJ and CaesarJ are *asymmetric* languages, meaning that they support more than a single kind of module and each kind has different capabilities and is subject to differing rules. This is undesirable, because whenever exceptions to rules arise, this always increases complexity, as exceptional cases and corresponding rules must be considered. All rules and mechanisms in a language would ideally apply in the same way to all constructs. In practice, many limitations and exceptions must be tolerated, due to technological or conceptual considerations. Languages that

extend existing languages, such as AspectJ[3] and CaesarJ, are especially prone to the occurrence of exceptions and special cases. Such languages add new constructs and features that may not be applicable to the constructs and libraries from the core language. In the case of AspectJ, many asymmetries are well-known, for instance module instantiation (explicit in Java and implicit in AspectJ).

CaesarJ shares with AspectJ one particularly important form of asymmetry. AspectJ advices are *nameless* and therefore not first-class entities. For this reason, visibility rules and polymorphism are not applicable for advice. CaesarJ basically reuses the pointcut and advice mechanisms of AspectJ, and therefore also has this asymmetry. In addition, CaesarJ adds new asymmetries of its own, many of which seem motivated either because current support for the language is not complete (the latest version number is 0.87) or due to incompatibilities with existing Java constructs.

Another important asymmetry in CaesarJ is the existence of two kinds of module (as is the case with AspectJ): plain Java classes and CaesarJ classes. Though CaesarJ classes are different from AspectJ aspects, it is also the case in CaesarJ that different rules apply to different modules. Plain-Java classes cannot hold **cclass**-specific members apart from enabling/disabling the composition of CaesarJ classes. Only inner CaesarJ classes, though not Java classes, can be virtual. CaesarJ classes cannot inherit from classes from the Java standard APIs (though curiously CaesarJ classes possess all state and behaviour from java.lang.Object) and it is not possible to declare an array of CaesarJ class types.

2.9.2 Support for Modularity and Module Reuse

Figure 2 summarizes the differences between AspectJ and CaesarJ as regards support for reuse, abstracting from the specific constructs and mechanisms. Part **a** of Figure 2 outlines AspectJ's separation between a reusable abstract aspect, a case-specific concrete aspect and the specific case, the application to which aspect functionality is to be composed. The parts of the aspect component that are potentially reusable are placed in an abstract aspect, which can be used in all different systems to which we want to compose the aspect functionality.

The fact that the abstract aspect is reusable does not imply the absence of many possible alternative implementations. Different implementations may be desirable for different concrete cases (performance considerations are one reason). In many cases, the implementation of the generally applicable part is orthogonal to the implementation of the case-specific part. In such cases, one would like to vary the two independently, i.e. that the boundary between the generally applicable and the case-specific parts comprise a *variation point*. That is indeed the case of CaesarJ though not of Java or AspectJ.

For independent evolution of modules to be possible, the various parts would ideally be polymorphic relative to each other. That is not the case with AspectJ, as the mechanism to compose an abstract aspect to concrete aspects is a form of inheritance akin to that of Java (though devoid of reference polymorphism). Only the terminal nodes, or "leaves", of an aspect inheritance hierarchy can be concrete. As a consequence, no polymorphism is possible as regards aspects. Concrete aspects are statically bound to a single abstract aspect and that is why there is a single abstract aspect at the top of part **a** of Figure 2. It is not possible to replace the super-aspect without performing invasive changes on the sub-aspects. By contrast, in CaesarJ it is possible to polymorphically switch implementations of a given part of a component without impact on the remaining modules of the component. This is illustrated in part **b** of Figure 2.

[3] In the case of AspectJ some asymmetries seem to exist by design. See, for instance, the post on named advice in: http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg03730.html

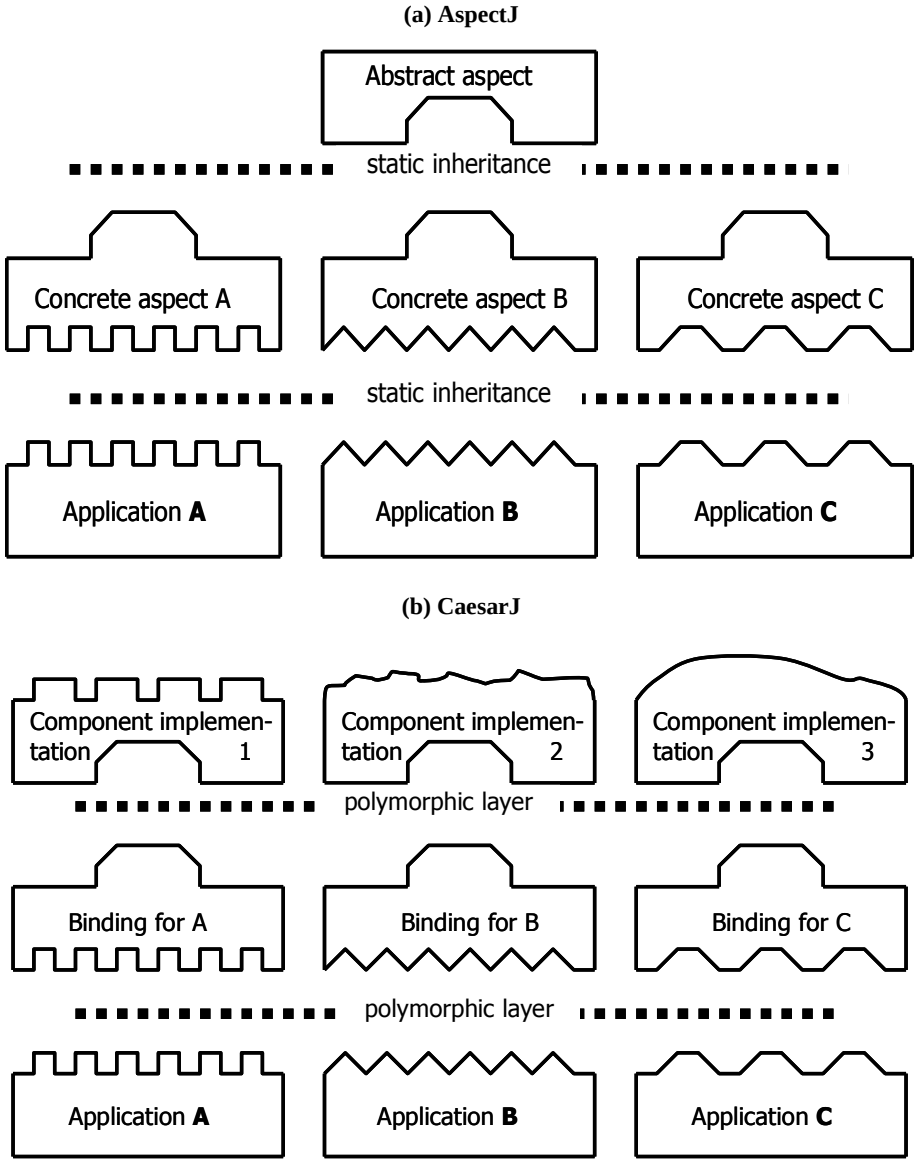

Figure 2. Mechanisms for reuse for AspectJ and CaesarJ.

2.9.3 Support for component integration

The differences between the AspectJ and CaesarJ mechanisms to support module and component integration are represented in Figure 3. Part a) of Figure 3 shows the mechanisms of AspectJ, which are based on ITDs and marker interfaces. AspectJ supports ITDs for two kinds of members: fields and methods. ITDs are used to compose such members to target classes from the specific application. Often, the mapping is not done directly, but rather in "two phases". The usual approach is to declare inner, often empty, interfaces within the abstract aspect to represent the concepts from the internal implementation of the aspect. These interfaces are often called *marker interfaces*. *Declare implements* clauses are then used to carry out the actual mapping. In AspectJ, when a **declare implements** clause makes a class implement a marker interface, the target class also acquires all members composed to the marker interface, through ITDs. Thus, **declare parents** clauses bind the marker interfaces, as well as the members it acquired through ITDs, to the classes from the application. ITDs and **declare implements** amount to a form of mixin composition [7]. ITDs are potentially applicable to multiple cases and placed in the abstract aspect for this reason. The **declare implements** clauses are specific to a particular combination of aspect and application, and therefore placed in the concrete sub-aspect.

It also usual for the abstract aspect to include concrete advice that refers to the marker interfaces. In the concrete sub-aspect, inherited code that refers to the marker interfaces also refers to the classes that are the target of the **declare implements** clauses. This way, no code from the abstract aspect needs to refer to case-specific classes. Note that in AspectJ (and CaesarJ), advice are not first-class entities. Advice do not have name to which they can be referred and therefore the concept of member visibility does not apply to them. For the same reason, it would be hard to enable advice to be overridden and be subject to some form of polymorphism.

A consequence of the AspectJ mechanism is that it does not distinguish between the implementation and binding parts of a component. Though pointcuts are primarily a mechanism to compose an aspect or component to an application, they are usually present as abstract pointcuts in the (supposedly reusable) abstract aspect. ITDs are usually top-level members within the abstract aspect, lacking a proper "home" to encapsulate them. There is no additional internal structure to organize them or to relate to the pointcuts and advice also found in the aspect hierarchy. As a result, AspectJ aspects tend to have a flat internal structure [24]. The reusable abstract aspects for Observer and Visitor patterns are typical examples of this trait.

Another consequence is that the target objects (instances of the original, target classes) and additional members share the *same identity*. Mechanisms to distinguish what is originally defined in the target class from what aspects add as ITDs are poor or non existent.

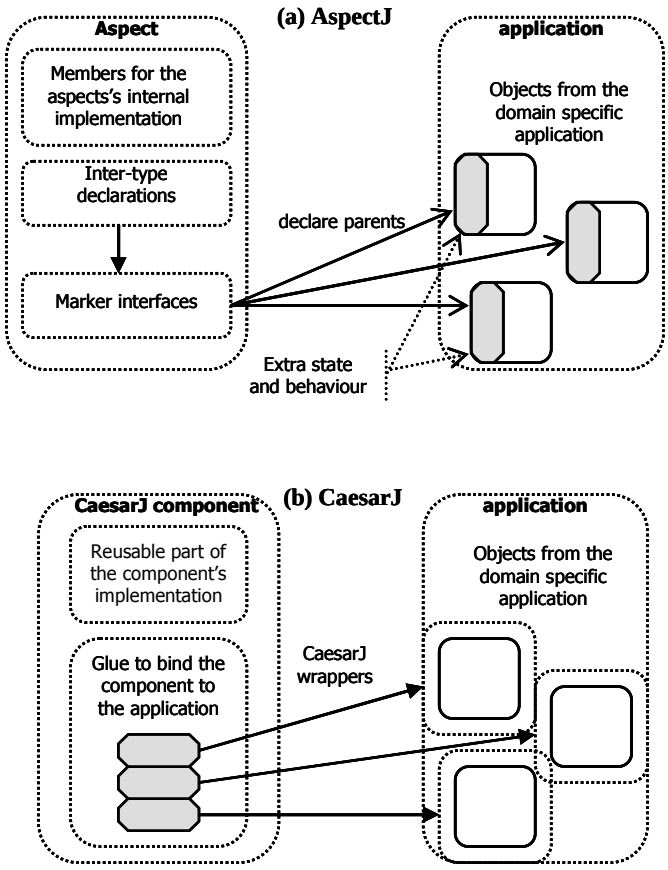

Figure 3. Mechanisms of AspectJ and CaesarJ for integrating with an application.

Unlike AspectJ, CaesarJ does not provide **declare warning** and **declare error** clauses, neither does CaesarJ provide **declare parents** clauses that enable aspects to augment the inheritance hierarchy of existing, target types (e.g., to make a given class to implement an additional interface). Relative to AspectJ, CaesarJ modules have a richer internal structure, as each add-on

to a target class is enclosed in a separate module that has its proper place within the component's internal structure. A potential drawback is that a single identity for target objects and additional state and behaviour is lost. An open question is whether this proves to be inconvenient in some cases.

3. APPROACH TAKEN

3.1. Scenarios by independent authors

Since the start of this project, the aim was to base the CaesarJ examples on the scenarios found in existing examples, rather than invent new scenarios for the purposes of this study. We give the name *scenario* to a specific example implementation of a given pattern. Different authors chose different scenarios for the various patterns. For instance, to illustrate Abstract Factory, James Cooper uses scenarios based on various kinds of plants in a garden and multiple kinds of garden. On the other hand, Larry Truet (*fluffycat*) illustrates Abstract Factory with a scenario based on various kinds of soups and soup factories.

Use of independently developed scenarios and implementations yields results that are less vulnerable to bias. The examples are also more likely to expose hurdles and situations that test the limits and capabilities of the subject language. For this reason, all scenarios used to illustrate the GoF patterns were derived from existing repositories of examples freely available on the Internet. These are shown in Table 1.

Adopted name of the repository	Author	Language	URL of the page from which the repository was extracted
Thinking in patterns	Bruce Eckel	Java 2	http://www.mindview.net/Books/TIPatterns/
Design pattern Java companion	James Cooper	Java 2	http://www.patterndepot.com/put/8/JavaPatterns.htm
Fluffy cat	Larry Truett	Java 2	http://www.fluffycat.com/Java-Design-Patterns/
Hannemann et al	Hannemann and Kiczales	Java 2/ AspectJ	http://hannemann.pbwiki.com/Design+Patterns
Huston	Vince Huston	Java 2	http://www.vincehuston.org/dp/
Guidi Polanco	Franco Guidi Polanco	Java 2	http://eii.ucv.cl/pers/guidi/documentos/ Guidi-GoFDesignPatternsInJava.pdf

Table 1 – Repositories of implementations of GoF patterns found in the Net

The scenarios, implementation approaches and style found in the various repositories vary widely across the examples. For instance, some authors heavily rely on graphics objects from the Java standard APIs while other repositories are mostly text based. In some cases, the participants in the pattern are instances of classes from the Java standard API, though more often these are represented by specific classes for a simple scenario. As would be expected, data structures used in implementations also vary and in some cases reuse functionality from the standard Java API. For instance, Eckel resorts to Java's Observer/Observable API to implement Observer, while Hannemann and Kiczales use weak hash maps whose elements are array lists. There is a wide range in style and implementations.

In order to explore the most promising features, a pre-selection work had to be done. From the classical patterns from the Gang-of-Four (GoF) [12] the following were chosen:

- **Observer** is one of the few patterns for which previous CaesarJ implementations are known [24]. Thus, Observer represents an ideal entry point to someone learning CaesarJ. **Chain of Responsibility** is structurally similar to Observer, though simpler (just one participant instead of two) and thus looked a direct follow up during training.
- **Abstract Factory** seemed the ideal candidate to test family polymorphism and virtual

classes, as Abstract Factory is really about emulating these features.

- **Singleton** is simple and generally one's favourite entry point to the GoF, being selected partly for these reason, and also for training purposes.
- **Decorator** looked the most suitable vehicle to test the hypothesis whether the wrapper mechanism can emulate mixin composition.
- **Visitor** and **Bridge** were selected because the patterns represent interesting structural problems suitable for testing CaesarJ's composition capabilities.

3.2. Multiple Examples per scenario

Though both AspectJ and CaesarJ provide feature to separate the parts of a component so as to enable some to be reusable, CaesarJ goes significantly further than AspectJ in some respects. The internal roles within a component are more clearly delineated in Caesar and benefit from a dimension of polymorphism absent in AspectJ. As the possibilities opened by these features became clear, we concluded that presenting a *single* CaesarJ implementation of each pattern wouldn't illustrate the full extent of the features. Thus, we decided to develop *multiple* examples of each selected pattern, to assess whether it is possible to re-use all potentially applicable parts.

	Thinking in patterns	**DP Java companion**	**Fluffy cat**	**Hannemann et al**	**Huston**	**Guidi Polanco**
Observer	X X[4]	X	X			
Abstract factory		X	X			
Decorator				X		
Visitor			X			
Bridge			X		X	
Chain of responsibility				X		
Singleton				X		

Table 2 – Implementations presented in this report

Ideally, the CaesarJ implementations would cover each and every scenario from the available repositories. Due to the time constraints of the undergraduate project, just the variations shown in Table 2 were developed.

4. ILLUSTRATIVE EXAMPLE: OBSERVER

This section illustrates the use of CaesarJ in implementing Observer. The use of Observer as a showcase for AOP is widespread [26][30], even in CaesarJ. Observer also has the advantage that it is relatively independent of implementations of it and can be described independent of the programming language used. For these reasons, it is used in this report as an illustrative example of CaesarJ. Structurally, all CaesarJ examples are more or less complex variants of the framework outlined in Figure 1. Thanks to the enhanced polymorphism, we managed to hold multiple implementations and bindings in the same system [21].

[4] Two scenarios of Observer by Eckel were implemented.

4.1. The Observer pattern

The intent of Observer – also known as *Publish-Subscribe* – is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [12]. The pattern is an example of a crosscutting concern connecting sets of otherwise unrelated classes, implemented as a simple framework. Observer defines two roles that are superimposed on existing objects of a given application: the role of *subject* for objects generating events of interest to other objects, which play the role of *observer*. Figure 4 shows the original class diagram for Observer shown in the GoF book [12].

Many OO implementations of Observer provide subjects with an extra field holding the list of their registered observers. Observers are registered (i.e., added to the list) by an *attach* operation and are removed from the list by a *detach* operation. When a subject gives rise to an interesting event – usually a change in its state – it calls a *notify* operation, which in turn calls the *update* operation of each registered observer.

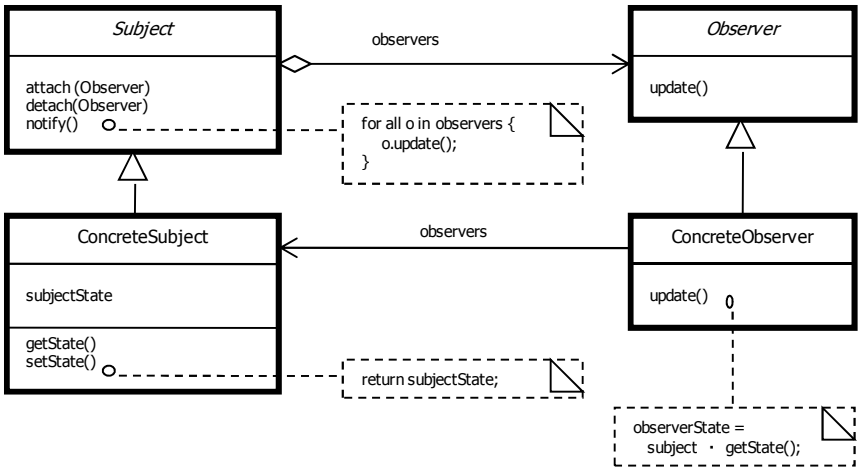

Figure 4. Structure for the Observer pattern

4.2. A concrete Observer scenario

To illustrate the CaesarJ mechanisms, a concrete example is described next, taken from Eckel's (incomplete) online book "Thinking in Patterns" [10]. The scenario by Eckel uses a flower as subject, shown in Listing 6. Its interesting events are the operations to open and close the petals, whose observers are one instance each of two unrelated types: a bee, shown in Listing 7, and a humming bird. Unlike in most toy examples of Observer from the repositories, *two* observing relationships are supported and the observers have different reactions to each, represented by messages sent to the console. In both listings, code related to the pattern role is shaded. Also unlike most available examples, Eckel's example of Observer uses the java.util standard API comprising interface Observer for observers and class Observable holding the logic of subjects. Since Java supports only single inheritance, this gives rise to a problem: what if subject classes need to inherit some other class other Observable? Eckel overcomes it by emulating multiple inheritance through *closures*. We use the definition of *lexical closure* proposed by Baumgartner et al [6]: a "mechanism for creating behaviour on the fly that can be invoked at a later time but has access to the lexical environment current when this behaviour was created." Since version 2, Java supports closures in the form of inner classes. Such classes have access to all members of the enclosing class, including private members.

Eckel's example resorts to inner classes that either extend Observable or implement Observer. See in Figure 5 the class diagram for Eckel's example. Each participant contains one inner class for each of the observing relationships that partly isolate the code related to the role played in the pattern. Participant classes remain free to inherit from some other class, though this particular example does not take advantage of this.

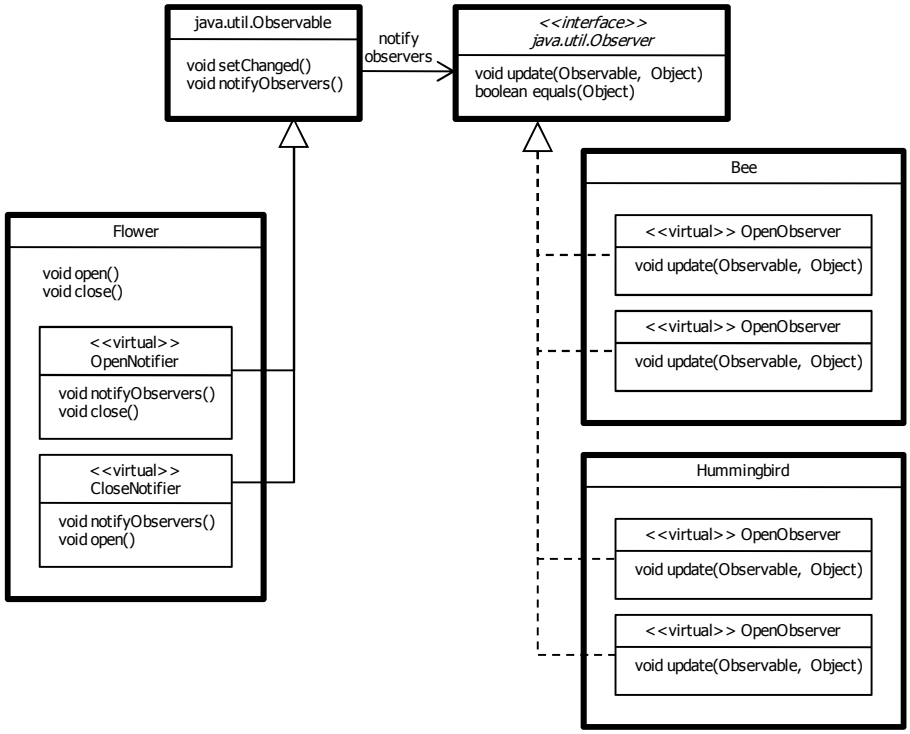

Figure 5. Class diagram for the scenario for Observer by Bruce Eckel, in Java

Despite the clever design, a tight structural relationship between participants and the roles they play in the pattern remains in place. Participant classes betray the *Double Personality* smell [30]. Any method of the subject performing an interesting operation must still include code related to its role in the pattern. Due to the requirement that observers only react to the first of multiple consecutive occurrences of the same operation, each observing relationship must monitor both operations. For this reason, observers of *open* need to be notified of *close*, to determine whether a call to *open* belongs to a sequence of calls to *open*, without calls to *close* in between. Duplication is particularly noticeable in the use *four* inner classes between the two observers. Each class duplicates the code related to the two observing relationships and each observing relationship requires a duplication of essentially the same logic. Listing 7 shows how classes Flower and Bee look like when devoid of the secondary concern, as is the case of both AspectJ and CaesarJ examples described in this section.

4.3. Observer in AspectJ

The implementation of Observer proposed by Hannemann and Kiczales is well-known. Since it illustrates the typical approach for structuring AspectJ aspects, code listings are included in this section. The structure of the AspectJ implementation exactly conforms to the critique presented in sections 2.9.2 and 2.9.3.

Listing 3 shows abstract reusable ObserverProtocol. It is identical to the aspect proposed by Hannemann and Kiczales [16], with the exception that a clearObservers operation. That operation was added by Monteiro and Fernandes so as to fill a gap in the aspect's functionality that was

needed to apply it to Eckel's scenario [28]. Listing 4 and Listing 5 show two concrete aspects that extends ObserverProtocol. The aspect shown in Listing 4 is used for Eckel's scenario for Observer, and which is also used to illustrate the use of CaesarJ in section 4.4. Listing 5 is a concrete aspect for a simpler scenario created by Hannemann and Kiczales.

```
public abstract aspect ObserverProtocol {
  /** This interface is used by extending aspects to say what types
    *  can be subjects. It models the subject role.    */
  protected interface Subject  { }

  /** This interface is used by extending aspects to say what types
    *  can be observers. It models the observer role.        */
  protected interface Observer { }

  /** Stores the mapping between <code>Subject</code>s and <code>
    *  Observer</code>s. For each subject, a <code>LinkedList</code>
    *  is of its observers is stored.      */
  private WeakHashMap perSubjectObservers;

  /** Returns a <code>Collection</code> of the observers of
    *  a particular subject. Used internally.
   protected List getObservers(Subject s) {
    if (perSubjectObservers == null) {
         perSubjectObservers = new WeakHashMap();
    }
    List observers = (List)perSubjectObservers.get(s);
    if ( observers == null ) {
       observers = new LinkedList();
       perSubjectObservers.put(s, observers);
    }
    return observers;
  }

  /** Adds an observer to a subject. This is the equivalent of
    * attach(), but is a method on the pattern aspect, not the subject.
  public void addObserver(Subject s, Observer o) {
    getObservers(s).add(o);
  }

  /** Removes an observer from a subject. This is the equivalent of
    *  detach(), but is a method on the pattern aspect, not the subject.
  public void removeObserver(Subject s, Observer o) {
      getObservers(s).remove(o);
  }
  public void clearObservers(Subject s) {
    getObservers(s).clear();
  }

  /** The join points after which to do the update.
    *  It replaces the normally scattered calls to <i>notify(). To be
    *  concretized by sub-aspects.   */
  protected abstract pointcut subjectChange(Subject s);

  /** Defines how each <code>Observer</code> is to be updated when a change
    * to a Subject occurs. To be concretized by sub-aspects.
    * @param s the subject on which a change of interest occurred
    * @param o the observer to be notifed of the change      */
  protected abstract void updateObserver(Subject s, Observer o);
}
```

Listing 3. Reusable AspectJ aspect for Observer

```
public aspect ObservingOpen extends ObserverProtocol {
  declare parents: Flower implements Subject;
  declare parents: (Bee || Hummingbird) implements Observer;

  public abstract void Observer.breakfastTime();
  private boolean Subject.alreadyOpen = false;

  protected pointcut subjectChange(Subject subject):
    execution(void Subject+.open()) && this(subject);
  after(Subject s): subjectChange(s) {
    Flower f = (Flower)s;
    if(f.isOpen() && !f.alreadyOpen) {
      Iterator iter = getObservers(s).iterator();
      while ( iter.hasNext() ) {
        updateObserver(s, ((Observer)iter.next()));
      }
    }
  }
  pointcut flowerClose(Subject flower):
    execution(void close()) && this(flower);
  after(Subject subject): flowerClose(subject) {
    if (subject instanceof Flower) {
      ((Flower)subject).alreadyOpen = false;
    }
  }
  protected void updateObserver(Subject s, Observer o) {
    o.breakfastTime();
    ((Flower)s).alreadyOpen = true;
  }
}
```

Listing 4. Concrete aspect in AspectJ for Eckel's flower scenario for Observer

```
public aspect ColorObserver extends ObserverProtocol{
  /**
   * Assings the Subject role to the Point class.
   * Roles are modeled as (empty) interfaces.   */
  declare parents: Point  implements Subject;

  /**
   * Assings the Observer role to the Screen class.
   * Roles are modeled as (empty) interfaces.   */
  declare parents: Screen implements Observer;

  /**
   * Specifies the join points that represent a change to the
   * Subject. Captures calls to Point.setColor(Color)
   * @param subject the Point acting as Subject       */
  protected pointcut subjectChange(Subject subject):
    call(void Point.setColor(Color)) && target(subject);

  /**
   * Defines how Observers are to be updated when a change
   * to a Subject occurs.
  protected void updateObserver(Subject subject, Observer o) {
    ((Screen)o).display("Screen updated " +
      "(point subject changed color).");
  }
}
```

Listing 5. Case-specific aspect for an Observer scenario by Hannemann and Kiczales

```
public class Flower {
   private boolean isOpen;
   private OpenNotifier oNotify = new OpenNotifier();
   private CloseNotifier cNotify = new CloseNotifier();
   public Flower() {
      isOpen = false;
   }
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      isOpen = true;
      oNotify.notifyObservers();
      cNotify.open();
   }
   public void close() { // Closes its petals
      System.out.println("Flower close.");
      isOpen = false;
      cNotify.notifyObservers();
      oNotify.close();
   }
   public Observable opening() {
      return oNotify;
   }
   public Observable closing() {
      return cNotify;
   }
   private class OpenNotifier extends Observable{
      private boolean alreadyOpen = false;
      public void notifyObservers() {
         if(isOpen && !alreadyOpen) {
            setChanged();
            super.notifyObservers();
            alreadyOpen = true;
         }
      }
      public void close() {
         alreadyOpen = false;
      }
   }
   private class CloseNotifier extends Observable{
      // similar to OpenNotifier, but focusing on operation close.
   }
}
```

Listing 6. Class Flower in Java – subject participant in Eckel's Observer.

```
public class Bee {
  private String name;
  private OpenObserver openObsrv = new OpenObserver();
  private CloseObserver closeObsrv = new CloseObserver();
  public Bee(String nm) { name = nm; }
  // An inner class for observing openings:
  private class OpenObserver implements Observer {
    public void update(Observable ob, Object a) {
      System.out.println("Bee " + name + "'s breakfast time!");
    }
  }
  // Another inner class for closings:
  private class CloseObserver      implements Observer{
    public void update(Observable ob, Object a) {
      System.out.println("Bee " + name + "'s bed time!");
    }
  }
  public Observer openObserver() { return openObsrv; }
  public Observer closeObserver() {
    return closeObsrv;
  }
}
```

Listing 7. Class Bee in Java – observer participant in Eckel's example.

4.4. Observer Implemented in CaesarJ

The CaesarJ component implementing Observer has all constituent modules referred in section 2 (Figure 1). The overall design is represented by a CI, whose code is shown in Listing 8 devoid of comments. The names used are different from those used in the GoF book but the semantics are the same.

```
public abstract cclass ObserverProtocol {
  public abstract cclass Subject {
    public abstract void addObserver(Observer obs);
    public abstract void removeObserver(Observer obs);
    public abstract void removeObserver();
    public abstract void notifyObservers();
    public abstract Object getState();

  public abstract cclass Observer {
    public abstract void refresh(Subject s);
  }
}
```

Listing 8. Collaboration interface for the Observer pattern.

Many different implementations of the component can be created, by extending the CI. Three CaesarJ implementations were developed for the work presented in this report. Two of them differ in just the type of the field that maps the subject to its observers. Listing 10 shows one of these, based on an ArrayList object. Note that the virtual class enclosed by CaesarJ implementation only extends virtual class declared in CI ObserverProtocol. From Listing 10, we can see that no explicit use of the **extends** keyword is necessary. Listing 9 shows a different CaesarJ implementation directly based on Eckel's idea of using the standard Java API Observer/ Observable. In this particular implementation, additional logic is necessary to adapt the Observable and Observer types to those declared by the CI. Since a CaesarJ class cannot extend a regular Java class, adapter classes are used as a stopgap. Using AspectJ, such an adaptation would be done through **declare parents** clauses.

```
class Notifier extends Observable {
   private ObserverProtocol.Subject sub;

   public Notifier(ObserverProtocol.Subject sub){
      super();
      this.sub = sub;
   }
   public void observer_notify() {
      super.setChanged();
      super.notifyObservers();
   }
   public ObserverProtocol.Subject getSubject(){
      return sub;
   }
}

class ObserverBox implements java.util.Observer {
   private ObserverProtocol.Observer o;

   public ObserverBox(ObserverProtocol.Observer obs){
      this.o=obs;
   }
   public void update(Observable obs, Object arg) {
      this.o.refresh(((Notifier)obs).getSubject());
   }

   public boolean equals(Object other){
      return (other instanceof ObserverBox) &&
         this.o.equals(((ObserverBox)other).o);
   }
}

public abstract cclass ObsImpl3 extends ObserverProtocol {
   public cclass Subject {
      private Notifier notifier;

      public Subject() {
         this.notifier=new Notifier(this);
      }
      public void addObserver(Observer obs){
         notifier.addObserver(new ObserverBox(obs));
      }
      public void removeObserver(Observer obs){
         notifier.deleteObserver(new ObserverBox(obs));
      }
      public void removeObserver(){
         notifier.deleteObservers();
      }
      public void notifyObservers(){
         notifier.observer_notify();
      }
      public Object getState() {
         return null;
      }
   }

}
```

Listing 9. A CaesarJ implementation based on the standard Java API Observer/Observable

```
public abstract cclass ObsImpl1 extends ObserverProtocol {
  public cclass Subject {
    private ArrayList observers = new ArrayList();

    public void addObserver(Observer obs){
      this.observers.add(obs);
    }
    public void removeObserver(Observer obs){
      this.observers.remove(obs);
    }
    public void removeObserver(){
      this.observers.clear();
    }
    public void notifyObservers(){
      Iterator it = this.observers.iterator();
      while(it.hasNext())
        ((Observer)it.next()).refresh(this);
    }
    public Object getState(){
      return null;
    }
  }
}
```

Listing 10. A simple CaesarJ implementation for Observer.

```
public cclass FlowerObserverDeploy
  extends ObsImpl3 & ObserverFlowerBinding {
}
```

Listing 11. Weavelet for the CaesarJ component for Eckel´s flower scenario for Observer

```
public class Flower {
  private boolean isOpen;

  public boolean isOpen() {
    return this.isOpen;
  }
  public Flower() {
    this.isOpen=false;
  }
  public void open() {
    this.isOpen=true;
  }
  public void close() {
    this.isOpen=false;
  }
}
```

```
public class Bee {
  private String name;

  public Bee(String name){
    this.name = name;
  }
  public void dinner(){
    System.out.println("Bee " + name +
"'s breakfast time!");
  }
  public void rest(){
    System.out.println("Bee " + name +
"'s bed time!");
  }
}
```

Listing 12. Flower and Bee participants devoid of secondary concerns

```
public abstract cclass ObserverFlowerBinding extends ObserverProtocol{
  public cclass FlowerOpening extends Subject wraps Flower {}
  public cclass FlowerClosing extends Subject wraps Flower {}

  public cclass BeeIsOpenObserver extends Observer wraps Bee {
    public void refresh(Subject s) {
      wrappee.dinner();
    }
  }

  public cclass BeeIsCloseObserver extends Observer wraps Bee {
    public void refresh(Subject s) {
       wrappee.rest();
    }
  }

  public cclass HummingbirdIsOpenObserver
       extends Observer wraps Hummingbird {
    public void refresh(Subject s) {
      wrappee.dinner();
    }
  }

  public HummingbirdIsCloseObserver
       extends Observer wraps Hummingbird {
    public void refresh(Subject s) {
      wrappee.rest();
    }
  }

  pointcut openCloseEvents(Flower f):
    (set(* Flower.isOpen)) && this(f);

  void around(Flower f, boolean new_val) :
    openCloseEvents(f) && args(new_val) {
    boolean old_val = f.isOpen();
    proceed(f,new_val);
    if(old_val != new_val)
      if(new_val)
        FlowerOpening(f).notifyObservers();
      else FlowerClosing(f).notifyObservers();
  }
}
```

Listing 13. A CaesarJ binding for the flower example.

5. OTHER PATTERNS

5.1. Abstract factory

The original intent of *Abstract Factory* is to "provide an interface for creating families of related or dependent objects without specifying their concrete classes" [12]. The solution originally proposed for the pattern is to use objects whose responsibility is to create all the objects of a given family, called *products*. Each different implementation of the family comprises a different set of *concrete products*, which is created by a different factory object. For each scenario, there is a single abstract factory representation, to which all concrete families are related. The abstract representation of the factory is called an *abstract factory* and the concrete objects that conform to that representation are called *concrete factories*. The abstract factory also declares the methods that yield the objects from the family of products – the *factory methods*. Different scenarios lead to different sets of products and therefore to different definitions of the abstract factory. The point of the pattern is that client objects depend on the abstract factory but not the concrete factories.

The pattern derives its motivation from the exactly the same problem that family polymorphism (section 2.3) is meant to tackle: to ensure that consistent combinations, or families, of collaborating objects are created while guaranteeing that objects from different families are not mixed. Thus, Abstract Factory can be said to be directly supported by mechanisms of CaesarJ. However, this also means that no reusable code was obtained. Each scenario maps to different, case-specific code. The main advantage over Java (and AspectJ) is the type-checking protection against mixing objects from different factory instances, which enables one to write simpler and more flexible code.

The approach taken was to use an abstract CaesarJ class to represent the factory in abstract terms, not relating it to a specific scenario for products. The CaesarJ class declares a virtual class that represents *any* product from *any* scenario: it just represents the general concept of product. Note that the virtual class in the topmost CaesarJ class corresponds to concrete products.

Figure 6 and Figure 8 show class diagrams of the original Java examples for Abstract Factory, from the fluffycat and Cooper repositories respectively. Consider the CaesarJ implementation of the fluffycat scenario, shown in Figure 7. The topmost CaesarJ class is AbstractSoupFactory, which declares an abstract virtual class Soup that represents any product from this scenario. This topmost CaesarJ class occupies the place of a CI but is specific to this particular scenario and thus not reusable. The two concrete CaesarJ classes (BostonSoupFactory and HonoluluSoupFactory) that inherit from the topmost CaesarJ class correspond to two different concrete factories. Each CaesarJ class declares the actual products as well as its corresponding factory methods, represented as top-level methods. In terms of the parts of a CaesarJ component, these CaesarJ classes are CaesarJ bindings. There are no CaesarJ implementations either, so no code is reusable.

The CaesarJ example for the James Cooper scenario is shown in Figure 9. The topmost CaesarJ class is AbstractGardenFactory, again not reusable. This scenario differs from that fluffycat in that just one class is used to represent the concrete products. For this reason it possible for virtual class Plant in AbstractGardenFactory to be concrete, unlike the corresponding from the fluffycat scenario – virtual class Soup. Again, there are no CaesarJ implementations.

As regards the AspectJ approach to Abstract Factory, Hannemann and Kiczales group this pattern with Bridge (see also section 5.2), Factory Method, Template Method and Builder. The authors consider these patterns to be structurally similar relative to the solutions in AspectJ they propose. Hannemann and Kiczales say that these patterns are about using inheritance to distinguish between different but related implementations. They also say that this is already nicely realized in OO, something to which we do not subscribe, in relation to at least Abstract Factory and Factory

Method. The AspectJ approach proposed by Hannemann and Kiczales is about replacing the abstract class (from their Java example) with an interface and resorting to inter-type declarations (ITDs – also known as *introductions* or *open class mechanism*) to compose concrete members to the interface. The aspect that does this is case-specific and there is no reusable code. Hannemann and Kiczales claim that this approach has the advantage of freeing concrete factories to inherit from some other class. However, this strikes us as a bit beside the point in the case of Abstract Factory. Moreover, the scenario of Observer by Eckel demonstrates that Java provides the means to overcome this "multiple inheritance" problem to some extent.

5.2. Bridge

The purpose of *Bridge* is to "decouple an abstraction from its implementation so that the two can vary independently" [12]. The pattern defines participants *abstraction*, which defines the interface of the abstraction, and *implementor*, which defines the interface for implementing classes. Naturally, client classes should depend only on the abstraction.

5.3. Chain of Responsibility

The purpose of *Chain of Responsibility* (CoR) is to "avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle a request; chain the receiving objects and pass the request along the chain until an object handles it" [12]. The pattern defines the role of *handler*, which declares an interface common to all objects that are potentially capable of handling the request. Concrete handler objects play the role of *concrete handler*. As in many patterns, there is also a *client* object, which naturally should depend on handler but not on any specific concrete handler.

The role of handler is superimposed and therefore would ideally be separated from case-specific classes that play the role. In the Java examples, the case-specific classes betray the usual symptoms of *Double Personality*, code tangling and scattering. After taking into account the differences between AspectJ and CaesarJ, the examples for CoR from both languages are very similar. Both solutions successfully modularize the CoR concern, yield reusable modules and resort to pointcuts and advice to capture request events.

The differences between the AspectJ and CaesarJ examples are what one would expect. The AspectJ examples are structured as a reusable abstract aspect and one case-specific concrete sub-aspect for each example. The CaesarJ example is structured as a CI, a reusable CaesarJ implementation, and one case-specific CaesarJ binding for each example.

The AspectJ example is typical. The abstract aspect uses inner marker interfaces to represent the pattern roles and ITDs to compose state and behaviour to the marker interfaces. The concrete sub-aspects use **declare parents** clauses to that state and behaviour to the case-specific classes. The pointcut is declared in the abstract aspect and defined in each concrete aspect. The advice that acts upon the captured joinpoints is placed in the abstract aspect.

The CaesarJ example is also typical of our approach of using CaesarJ and conforms to the description of section 2. Figure 14 shows one scenario for CoR and Figure 15 shows the CaesarJ example for the same scenario. The CI specifies the abstract roles (handler and request, though only handler is important), which are used through inheritance by the remaining top-level CaesarJ classes. The CaesarJ bindings are the only modules to use pointcuts and advice, which in CaesarJ are exclusively used as glue code and never part of the interface of component. Extra state and behaviour to be composed to case-specific objects is defined in the virtual classes of the CaesarJ bindings, which are in turn wrapping the case-specific classes. The case-specific objects are passed to the wrappers upon their creation, as constructor arguments.

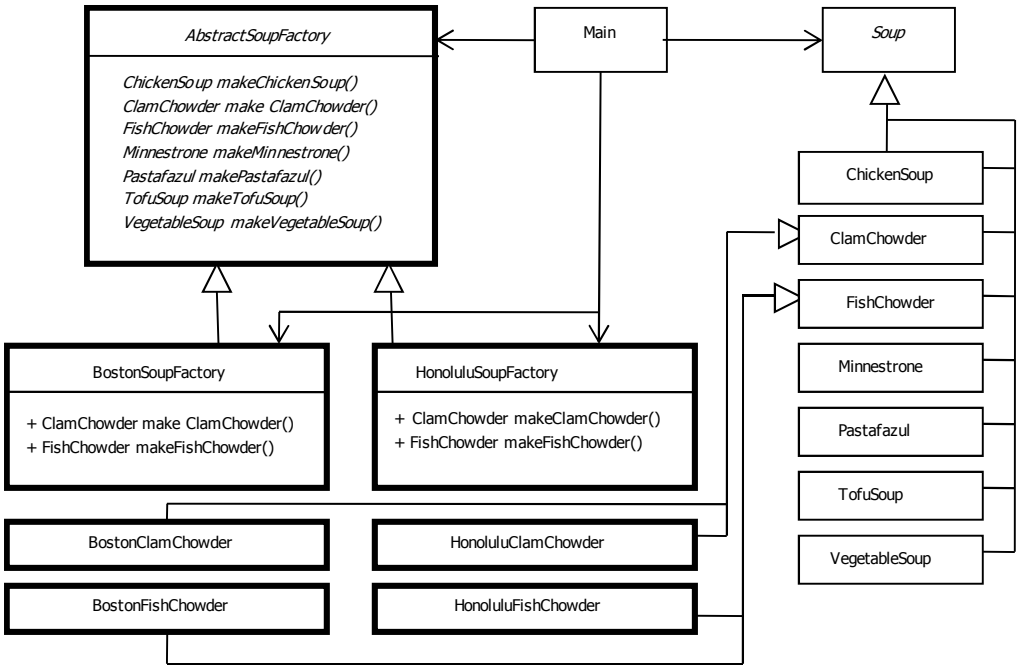

Figure 6. Abstract Factory: class diagram for the fluffycat scenario in Java.

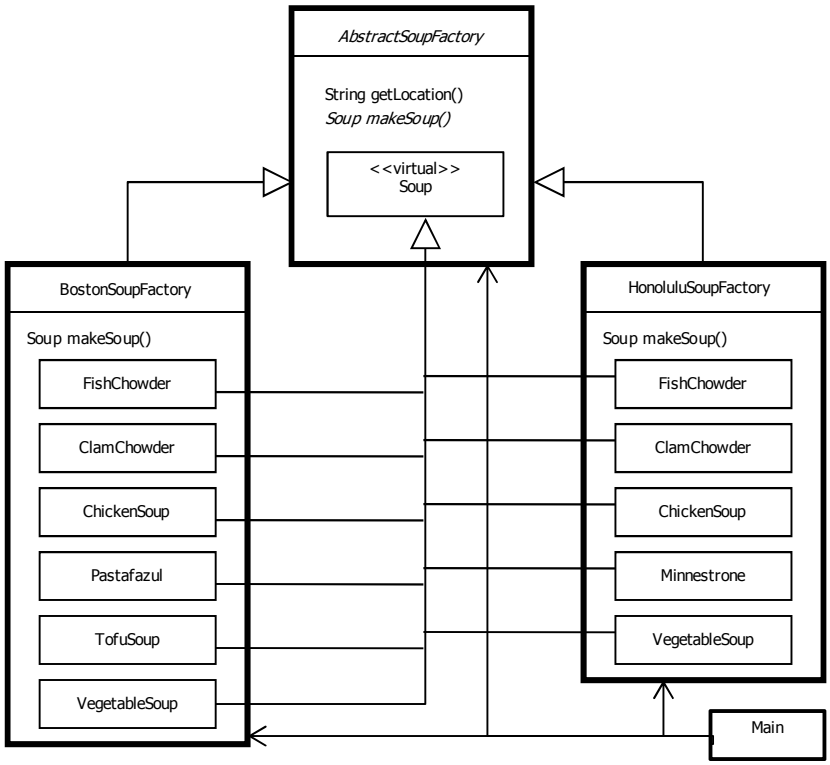

Figure 7. Abstract Factory: class diagram for the fluffycat scenario in CaesarJ.

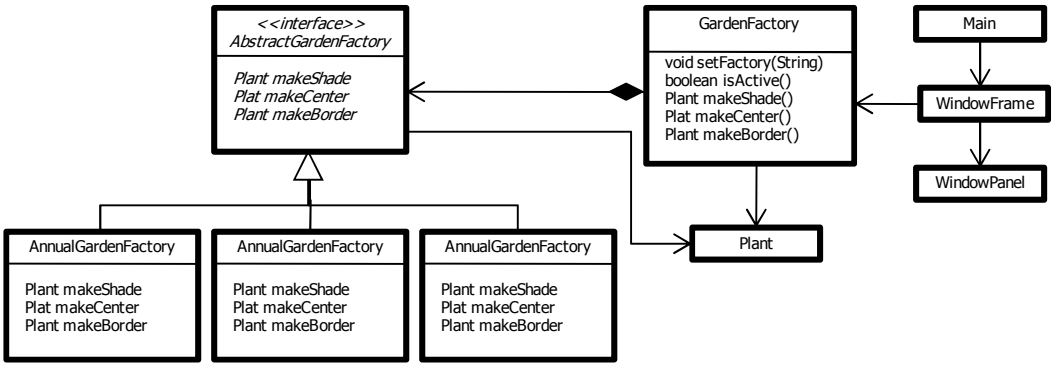

Figure 8.Abstract Factory: class diagram for the James Cooper scenario in Java.

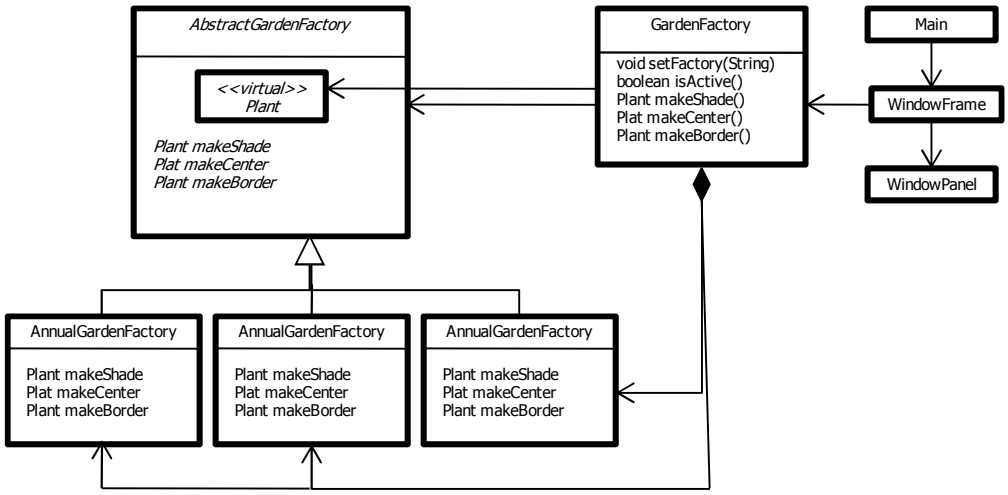

Figure 9. Abstract Factory: class diagram for the James Cooper scenario in CaesarJ.

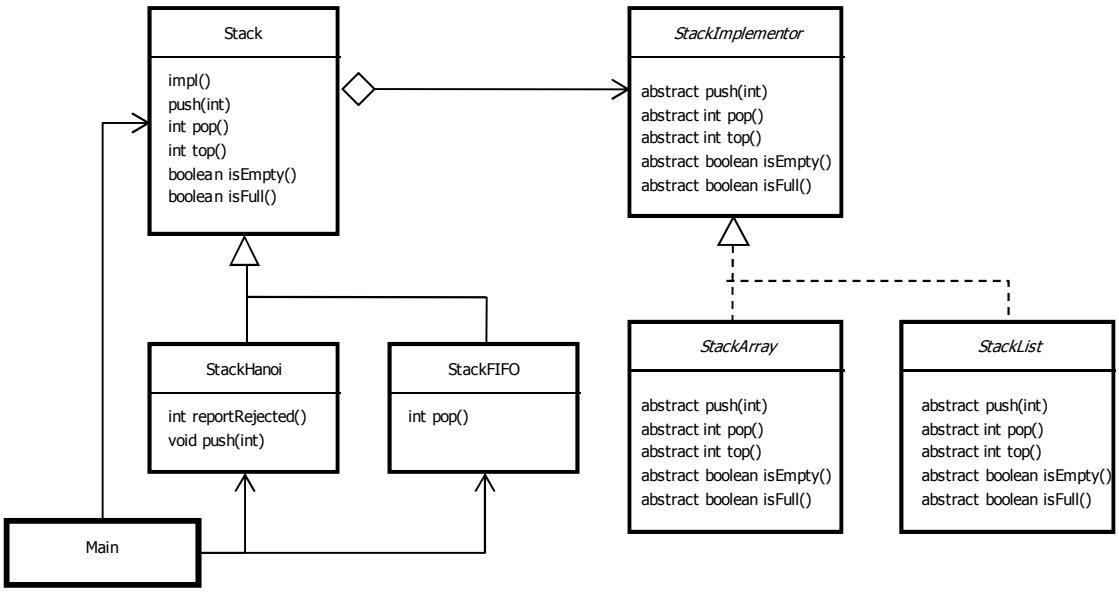

Figure 10. Bridge: class diagram for the Vince Huston scenario in Java.

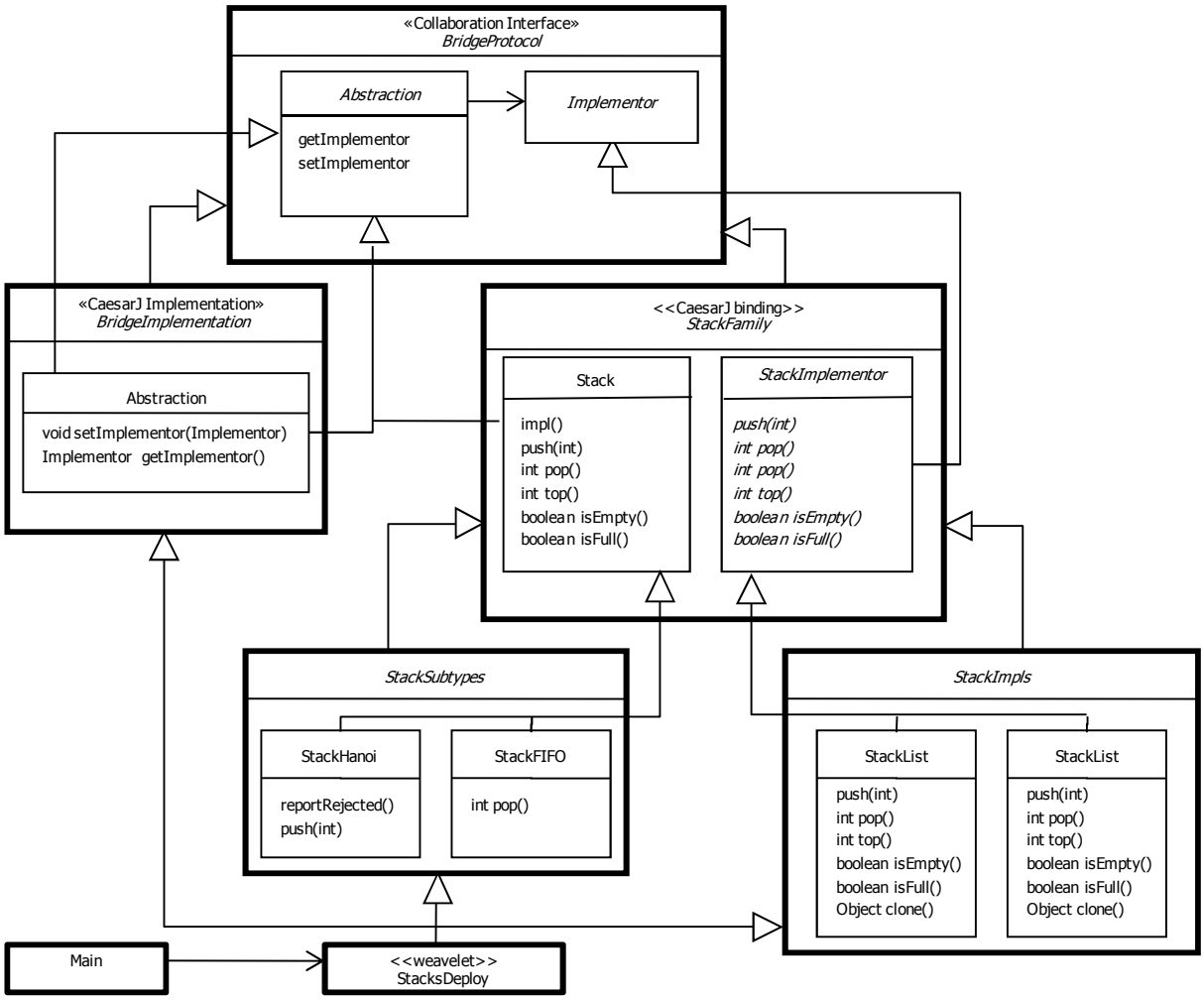

Figure 11. Bridge: class diagram for the Vince Huston scenario in CaesarJ.

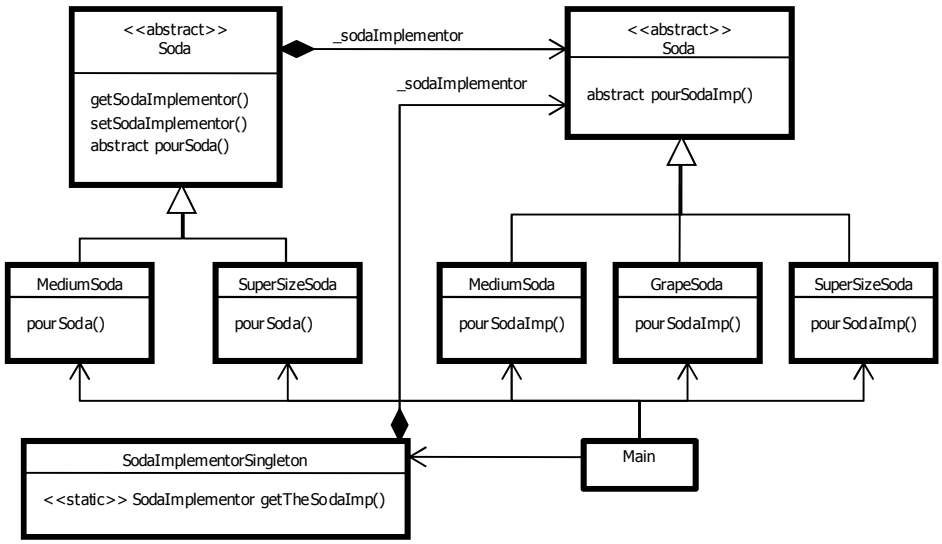

Figure 12. Bridge: class diagram for the fluffycat scenario in Java.

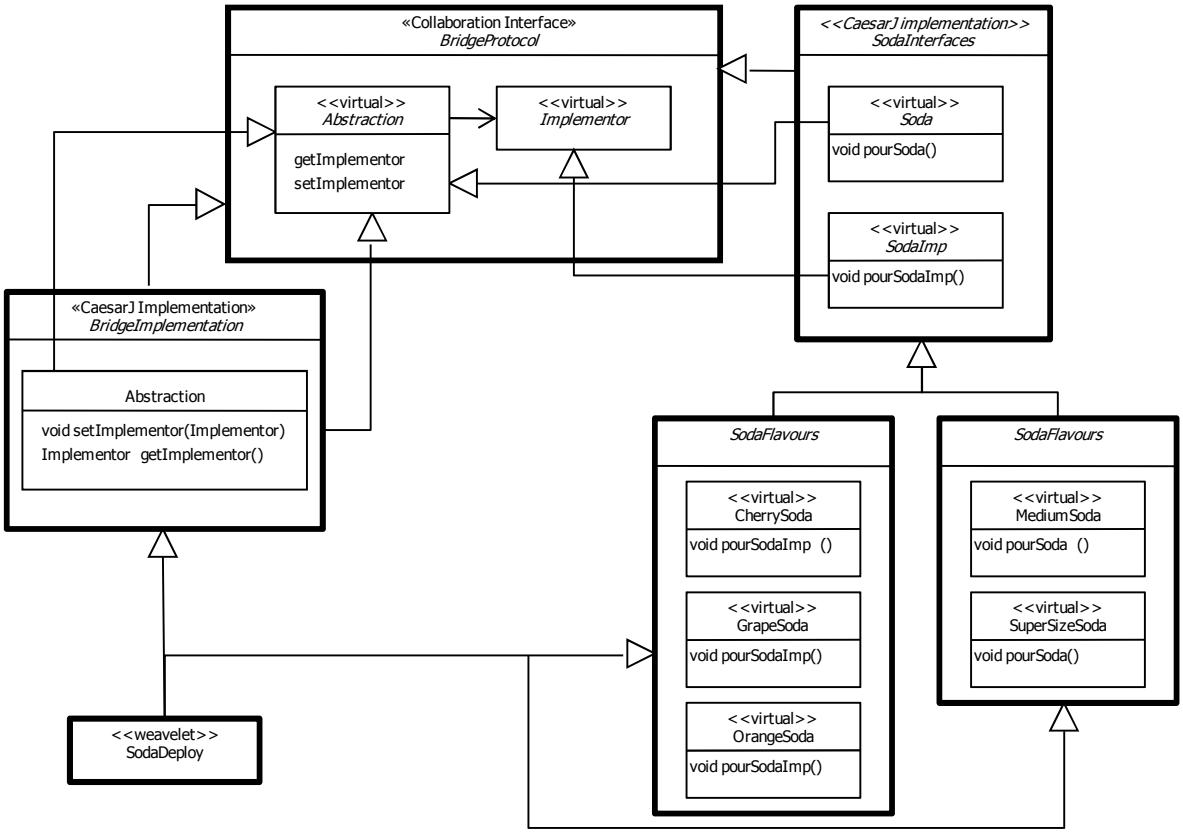

Figure 13. Bridge: class diagram for the fluffycat scenario in CaesarJ.

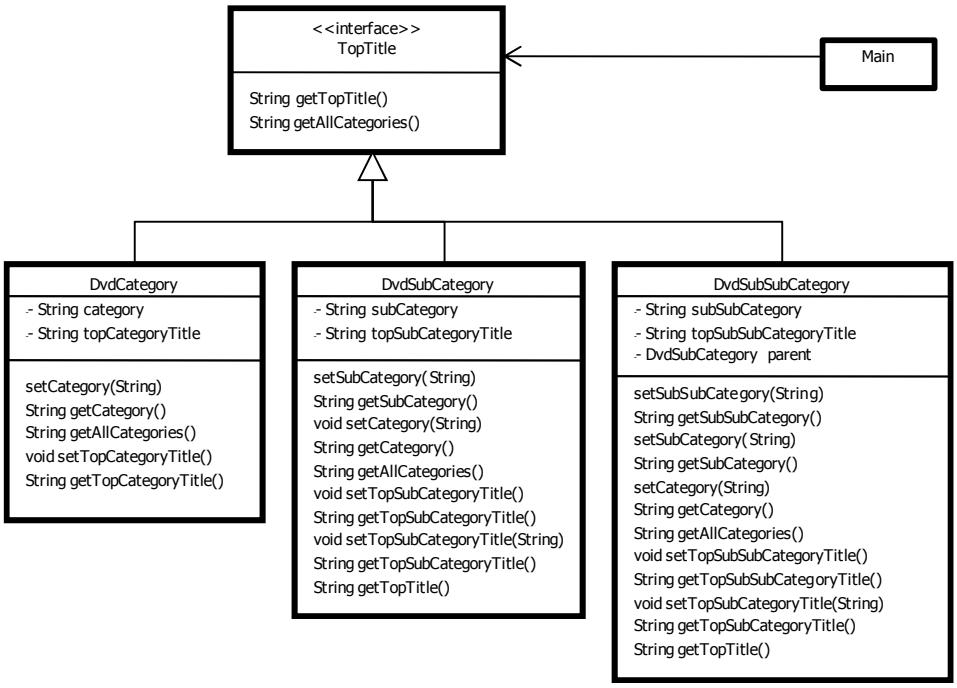

Figure 14. Chain of Responsibility: class diagram for the Huston scenario in Java.

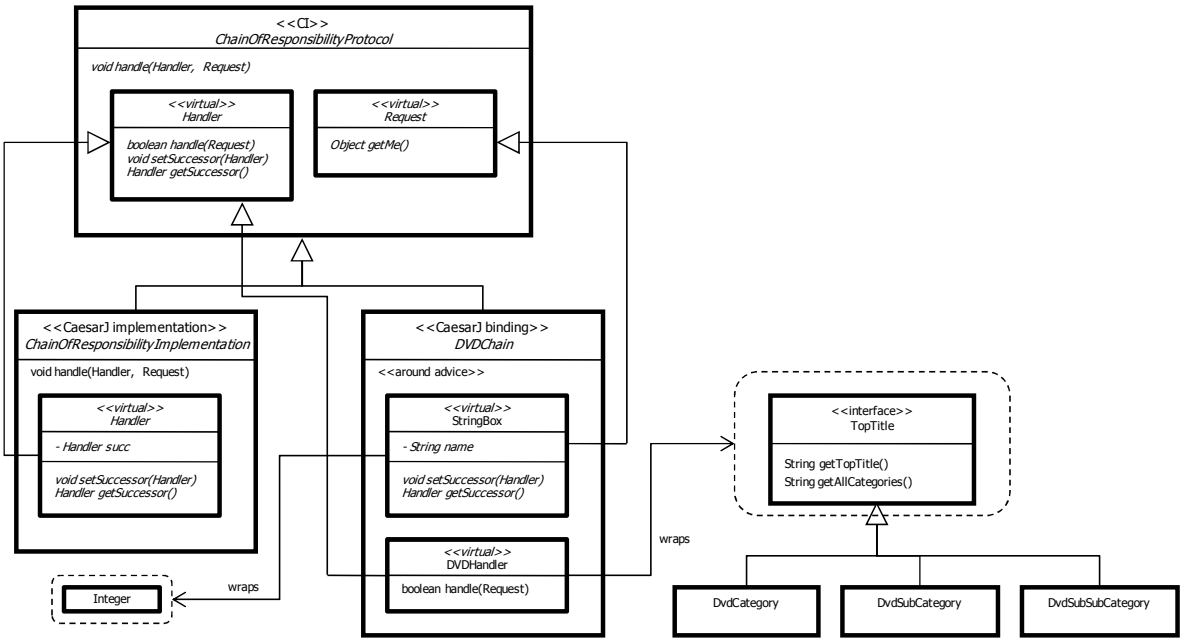

Figure 15. Chain of Responsibility: class diagram for the Huston scenario in CaesarJ.

5.4. Visitor

The purpose of *Visitor* is to "represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates" [12]. Visitor is about a tree structure to whose nodes one may want to add various different additional operations. It quickly becomes cumbersome to place the logic for many different operations in the nodes of the structure. Visitor embodies the problem of *double dispatch*, i.e., the ability to select a given block of code based on two different types that can evolve independently. This is an instance of the more general case of *multiple dispatch* [9].

It should be noted that CaesarJ supports only single dispatch based on the type of the family object associated with the concrete family at hand. One could conjecture that if CaesarJ supported multiple dispatch relative to a family of types, it would directly support the effect that motivates the use of Visitor, which is not currently the case. This limitation motivated the proposal by Gasiunas et al [14] of multiple dispatch on virtual types. Presently, however, this remains an issue left to future work.

The OO implementation originally proposed by Visitor is to define the role *element*, or *node*, which represents a node of the tree and declares an *accept* operation that takes a as argument an object playing the role of *visitor*. The interface of Visitor declares a different *visit* operation for each concrete subclass of element. The visit operations take an element as argument and implement the additional operation to be executed on element. Often, the visit operations differ in the argument type only. Thus, there is some noticeable duplication.

Figure 16 and Figure 18 show two Java examples of Visitor, by Vince Huston and Bruce Eckel respectively. Figure 17 and Figure 19 show the corresponding examples in CaesarJ. Neither Java, nor AspectJ nor CaesarJ provide completely satisfactory solutions for this pattern. In all cases, the duplication of the visit operation is not eliminated. The Java implementations have the additional drawback that concrete elements must define the accept operation – a instance of *Double Personality* [30]. The AspectJ and CaesarJ implementations manage to separate the *visit* operations from the element objects, through their respective means for composing extra members to domain-specific objects: ITDs in the case of AspectJ and wrappers in the case of CaesarJ. The CaesarJ implementation has the advantage over AspectJ in that wrapper composition is performed on an object by object basis, while the AspectJ intertype declarations apply statically, to *all* instances of the target class.

5.5. Singleton

The purpose of *Singleton* is to "ensure a class only has one instance, and provide a global point of access to it" [12]. The usual OO approach is to make all constructors private, provide the class with a static field referring to the unique instance and provide the class with a creation method the always returns that instance. It is also usual for the creation method to instantiate the instance when first called.

The CaesarJ Singleton closely resembles that in AspectJ, since it is based on pointcut and advice, just like the AspectJ example. However, even in this simple example CaesarJ provides an opportunity to separate a reusable advice into an implementation module. Since the example does not include a CI, the CaesarJ implementation is at the top of the inheritance hierarchy.

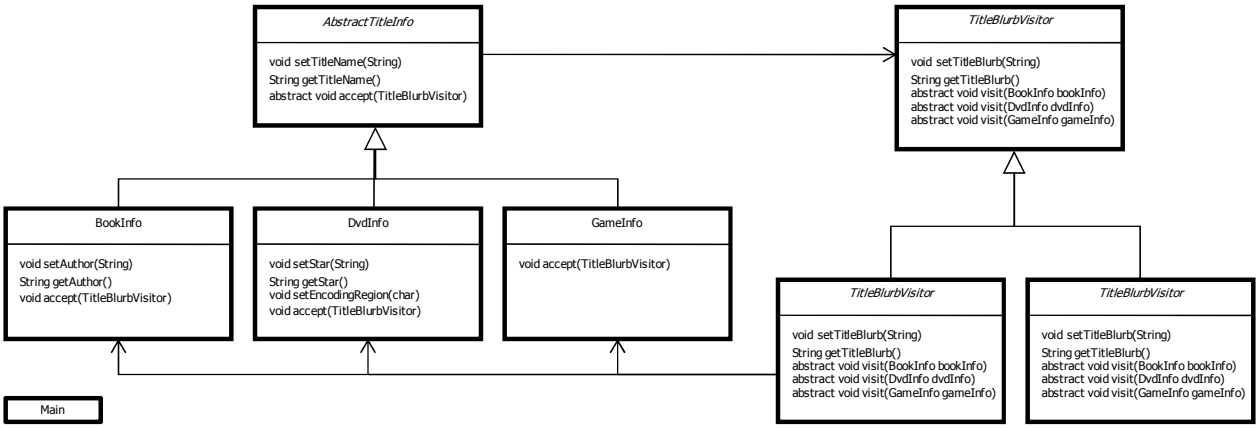

Figure 16. Visitor: class diagram for the Huston scenario in Java.

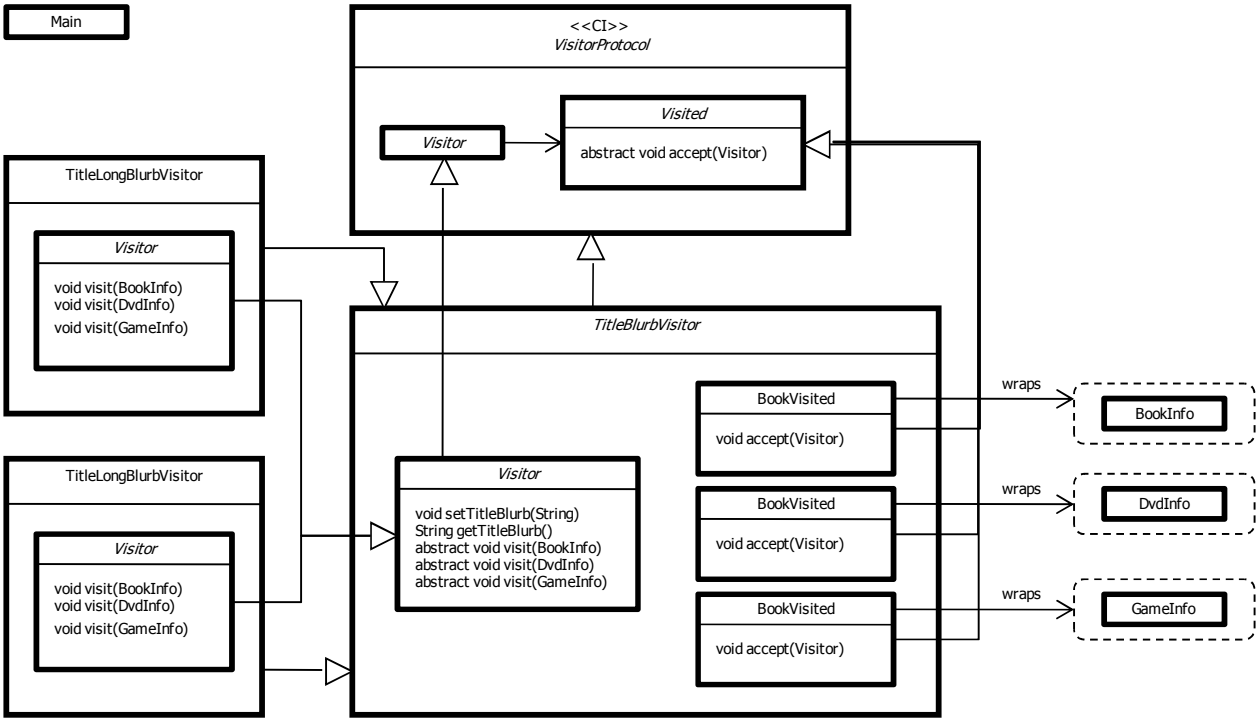

Figure 17. Visitor: class diagram for the Huston scenario in CaesarJ.

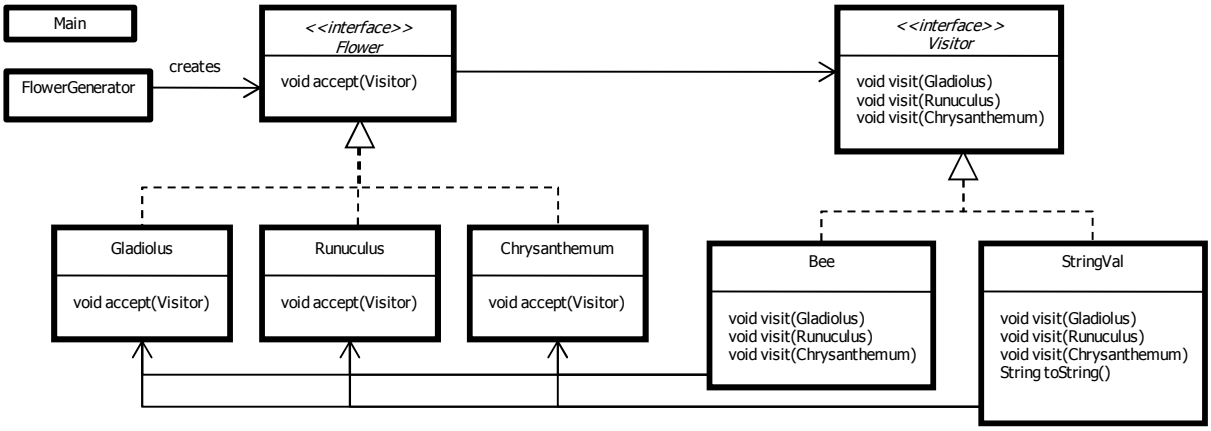

Figure 18. Visitor: class diagram for the Eckel scenario in Java.

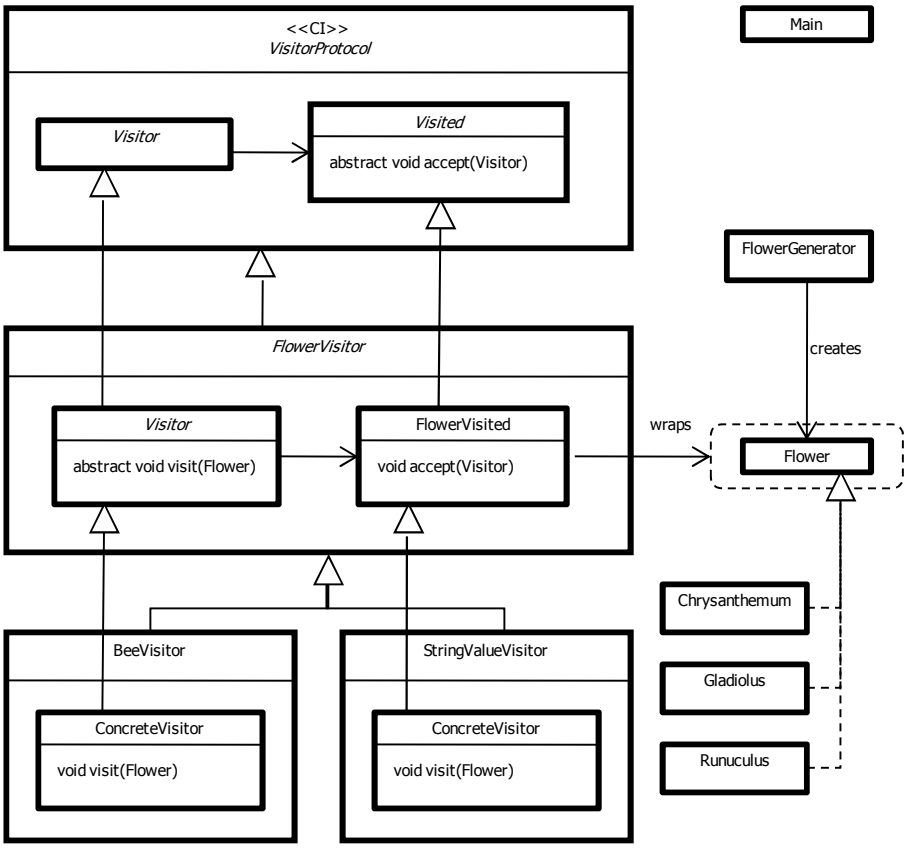

Figure 19. Visitor: class diagram for the Eckel scenario in CaesarJ.

5.6. Decorator

The purpose of *Decorator* is to "attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality" [12]. Decorator is about wrapping an object with another object that exposes the same interface and adds functionality to operations declared in the common interface. Decorator defines the roles of *component*, which defines the common interface; and of *decorator*, which is the role played by wrapper objects. Each wrapper is a *concrete decorator*, i.e. an object that maintains a reference to the component, conforms to the common interface and adds responsibilities to the component.

One issue related to this pattern is whether the original identity of component is preserved. This is not the case of traditional OO languages, in which the use of decorators entails creating additional identities. This has the consequence that certain fragments of code cannot be reused as the meaning of the self variable (i.e., **this** in the case of Java) corresponds to different objects. This problem is well-known and is usually referred as the "self problem" and was thoroughly discussed in the context of delegation-based OO languages, which are known to be free of this problem [22].

In addition to the self problem, relevant considerations in this pattern relate to the various ways in which modules can be composed. These include: (1) the possibility to apply a decorator more than once; (2) apply multiple decorators to a component, in which case some decorators are really decorating other decorators rather the original component; and (3) the order with which decorators are composed: differences in the order of composition should impact on the resulting behaviour.

The AspectJ approach proposed by Hanneman and Kiczales is to use pointcuts and advice to mimic the effect of decorators. This approach has the advantages that composition is totally transparent to the component and preserves its identity. The disadvantages are the loss of dynamic flexibility: the composition applies to all instances of the class of the component rather than to single objects. It is also possible to control the order of composition, through the mechanism of aspect precedence. However, this is inflexible, as it cannot be done dynamically.

A straightforward use of pointcuts and advice, such as found in the example by Hannemann and Kiczales, does not deal with the issue of decorating only individual instances of the component class. However, it is relatively simple to solve this problem through a slightly more complex implementation, as proposed by Monteiro and Fernandes [29]. Their proposal is for the aspect to register the component and for its advice to check that the target object corresponds to the component.

As regards composing the same decorator more than once, the AspectJ approach fails to some extent. AspectJ does not provide mechanisms to achieve this effect in a simple way, because one different aspect is required for each decorator instance. At the very least, an instantiation mode other than the default (singleton) must be specifically programmed for the aspect.

Two different implementations of Decorator were developed in CaesarJ. The first replicates the approach based on a pointcut and an around advice, but taking advantage of the dynamic deployment features of CaesarJ, which bring a number of advantages. CaesarJ aspects can be instantiated using **new** and therefore it is straightforward to compose a given decorator as many times as wanted. All is needed is to create as many instances of the decorator as wanted. Dynamic deployment is supported and thus it is also straightforward to control the moments in which a decorator is active. Multiple activations and deactivations of the same aspect are also straightforward. CaesarJ also provides direct language support to effect of decorating just a specific instance of the component class, by means of the *deploy on object* feature. However, the

order with which aspects compose their functionality is not under control of the programmer and no way was found to control the order of composition of decorators. Neither the dynamic deployment nor the order with which the CaesarJ components are instantiated provided a solution. In this respect, AspectJ is more flexible than CaesarJ.

The second CaesarJ implementation is based on the wrapper mechanism and without pointcuts and advice. The use of wrappers conforms more closely to the original intent of the pattern, namely in the dynamic nature of the compositions [18] and in the possibility of varying the order with which decorators are composed [29]. However, it does not bring any advantage over traditional OO implementations. CaesarJ wrappers have a different identity than the wrappees, with the consequence that the self problem is also not solved.

6. DISCUSSION

This section presents the conclusions derived from the effort of developing the examples described in the previous section. It is structured as follows. Section 6.1 presents a short analysis of what reusable modules were derived from the effort. Section 6.2 analyses differences between CaesarJ and AspectJ with respect to use pointcuts. Section 6.3 analyses the impact that CIs can have on the reasoning at the design level.

6.1. Reusable modules obtained from the examples

Table 3 shows what direct language support is provided by AspectJ and CaesarJ to the patterns covered in the study, as well as an overview of what reusable modules were derived. In the case of AspectJ the reusable modules are abstract aspects. In the case of CaesarJ, CIs and CaesarJ implementations are considered. Depending on the nature of the pattern, different levels of reuse are obtained. In some cases, only the high level design (i.e., the CI) is reusable and in others only the implementation is reusable.

Table 3. Reusable modules in AspectJ and CaesarJ

Direct support and reusable parts:	**Direct language support for the pattern**		**AspectJ** (abstract aspects)	**CaesarJ** (CIs)	**CaesarJ** (CaesarJ implementations)
	AspectJ	**CaesarJ**			
Abstract Factory	No	Yes	No	No	No
Bridge	No	No	No	Yes	Yes
Chain of Responsibility	No	No	Yes	Yes	Yes
Decorator	With limitations	With limitations	No	No	No
Observer	No	No	Yes	Yes	Yes
Singleton	No	No	Yes	No	Yes
Visitor	No	No	Yes	Yes	No

6.2. Use of pointcuts

The fact that CaesarJ shares with AspectJ the mechanism of pointcuts and advice invites some comparisons. One interesting question is whether pointcuts are used the same way as in AspectJ, or the CaesarJ-specific mechanisms have an influence on the use patterns for pointcuts and advice. It turns out that in CaesarJ, pointcuts and advice feature less prominently than with AspectJ. Thanks to CaesarJ's more sophisticated mechanisms to deal with structure, it is possible to separate the various parts of a component in different modules to a greater extent than with AspectJ. One is thus more sparing in the use of pointcuts and advice, whose use is relegated to

that of glue code (i.e., in the CaesarJ bindings). The experience gained so far suggests that in CaesarJ, pointcuts and advice should be left to those situations in which the behaviour to be composed does not follow an identifiable pattern in the static structure of the system, e.g., scattered calls to a method or constructor. That is the case in all uses of pointcuts and advice in the examples presented in Table 4.

Table 4 presents a summary of the use of CaesarJ pointcuts, CIs, implementations and bindings, in the implementations referred in Table 2. As it would be expected, not all examples include all features. The bindings are the exception, which is also what we expected. Note that some of the constructs available in CaesarJ are not covered in this paper (e.g., deploy on object) as the implementations from Table 2 do not comprise sufficient material to perform an assessment.

Table 4. Use of mechanisms in the CaesarJ examples.

Use of the mechanism:	pointcut/advice	Collaboration interface	CaesarJ implementation	CaesarJ binding
Abstract Factory	No	No	No	Yes
Bridge	No	Yes	Yes	Yes
Chain of Responsibility	Yes	Yes	Yes	Yes
Decorator	No(*)	No	No	Yes
Observer	Yes	Yes	Yes	Yes
Singleton	Yes	No	Yes	Yes
Visitor	No	Yes	No	Yes

(*) *One scenario does use pointcuts/ advice but in this case we do not consider it good practice and do not count it for this reason.*

Since the ITDs of AspectJ can be regarded as an instance of mixin composition, we initially hypothesised that mixins could be used as a more structured alternative to AspectJ ITDs, as well as providing direct language support to decorators. However, mixin composition in CaesarJ cannot be used "on the fly": a specific declaration of a CaesarJ class extending the desired modules must be created for each different combination. In our view, this defeats the aim by Decorator of preventing a combinatorial explosion of class declarations. In addition, mixin composition is restricted to top-level CaesarJ classes. For these reasons, the primary use of mixins is to yield the weavelets that integrate the various modules of the component.

6.3. Reasoning with Collaboration Interfaces

One advantage of CaesarJ clearly felt relative to both Java and AspectJ was when reasoning about the examples through class diagrams. CIs provide a design-level view of a component or sub-system found in class diagrams but generally absent in mainstream languages. For this reason we initially expected that CIs would be of help to reason with the overall structure of the component when looking at the code. It turned out that class diagrams of CaesarJ designs provided significant benefits as regards comprehensibility. The diagrams representing the CaesarJ designs are conceptually closer to the original design intentions than traditional class diagrams, exposing the relationships between classes and individual operations more faithfully than traditional class diagrams (e.g., those representing the Java implementations shown in Table 1). The distinction between top-level classes and nested classes facilitated the reasoning with the overall design, as well facilitating the task of mapping the original Java designs to CaesarJ designs. The enhanced clarity also applies to top-level methods. For instance, the factory method from the CaesarJ design for Abstract Factory (see for example Figure 7) is a top level method placed on the same level as the virtual classes. This exposes a design decision that is absent from traditional designs because nested classes are absent from traditional class diagrams and top-level

methods do not carry the same meaning.

7. FUTURE WORK

The preliminary work documented in this report can be further developed in many directions. For the immediate future, we identify the following fronts:

- **Complete the GoF repository.** Completing the GoF repository seems the obvious next step, though not necessarily the most fruitful. Some patterns cover situations in which AOP is not expected to improve on traditional solutions. CaesarJ implementations of some patterns are likely to be identical to those in Java (e.g., Façade) or perhaps worse (e.g., Iterator in Java 5). On the other hand, a complete GoF repository would provide more material for comparisons with other languages.
- **Develop comparisons with AspectJ.** The focus of this report is on the description of the implementations of the seven patterns selected as case studies. Though some comparisons between CaesarJ and AspectJ are made, these are somewhat general in nature. More thorough and comprehensive comparative analysis of the AspectJ and CaesarJ implementations could to be carried out.
- **Further explore CaesarJ features.** Some features of CaesarJ are not thoroughly explored in this study. One example is dynamic aspect deployment. Another interesting issue is to assess the comparative advantages of the wrapper mechanism with respect to AspectJ-style ITDs. The targets of ITDs and additional members share the same identity. Though ITDs are structurally poorer than wrappers, it remains an open question whether a single identity brings benefits in some cases.
- **Further explore CaesarJ designs.** Even in the scope of the seven patterns covered here, there is room to further explore the possibilities offered by CaesarJ to derive new, better designs. One example is the dichotomy between the design approach based on the systematic use of CaesarJ classes to represent all abstractions (i.e., without wrappers) and the more asymmetric approach of designing an isolated CaesarJ component that must integrate with plain Java classes (through wrappers).
- **Derive Refactorings for CaesarJ.** In the past, availability of a GoF repository in a given language was used as a basis for pinpointing refactorings [30] for that language. This work provides similar opportunities for CaesarJ.
- **Extend Work to Other AOP Languages.** We aim to cover more aspect-oriented languages. The planned first stage is to focus on AOP languages that extend Java, as is the case with Object Teams [17]. Object Teams would form an interesting triangle of AOP languages. AspectJ has a rich pointcut language and nameless advice but no constructs for adequately structure the internals of aspects. CaesarJ has such constructs in the form of virtual classes, family polymorphism and wrappers, as well as a pointcut protocol and nameless advice just like AspectJ. Object Teams has a facility that mimics the quantification capabilities of AspectJ pointcuts, though with a much more coarse-grained joinpoint model (just method calls) and uses methods rather than nameless advice. In addition, Object Teams also supports virtual classes and family polymorphism. To our knowledge, there are no studies assessing the relative merits of these language designs.
- **Quantitative studies.** Availability of examples simultaneously coded in CaesarJ and in other languages opens the way to perform quantitative studies that assess CaesarJ characteristics with respect to those other languages. Thus, the collection of examples described in this report could be used for quantitative studies such as those performed by

Garcia et al [13].

7.1. Preparation for systematic studies

On the basis of the experience gained, the decision to base all scenarios on examples from independent authors proved advantageous. The existing available repositories already provide a significant variety of styles and approaches comprising much richer material than could be obtained by developing new examples in-house. Adoption of independently developed examples also provides a stronger guarantee that scenarios are not biased.

However, the experience gained in dealing with such disparate repositories called into attention the need to carry out various kinds of adaptations as prior work to more thorough and systematic studies. Such preparatory work will be essential for many of the fronts identified at the start of this section. The more rigorous the work to be performed, the greater will be the need for some prior adaptation and standardization. We identified the following desirable procedures:

- **Standardize the code style**. Ensure that all versions are using the same coding styles. This can be done with the assistance of the automatic formatter of the eclipse/JDT plug-in, after some suitable configuration. Consistency of code style is essential for deriving conclusions based of certain kinds of metrics, such as the number of lines of code.
- **Extract the client participant from the driver of the pattern**. Many patterns define a *client* participant that represents the objects that use functionality provided the primary participants in the pattern. This participant is important to assess whether certain independencies and variation points are achieved. That is why the client often appears in class diagrams from the GoF book [12]. However, *client* is often overlooked by the authors of the repositories. Many examples simply include a class containing the main method that acts as the driver for the example. This is different from *client* and naturally betrays dependencies on the remaining participants that would not be found in a genuine *client*. Thus, that participant must be extracted in many cases.
- **Increase the number of participants for each role**. For each repository, develop a version of the examples with a larger number of participants in each role. The version with fewer participants should be preserved, as availability of smaller and larger versions provides an opportunity to perform scalability assessments based on the comparison of metrics derived from both versions [13].
- **Develop unit tests**. In some cases, unit tests may provide a useful complement to a driver class for the example. For instance, availability of unit tests may be useful in refactoring experiments based on the examples [30].
- **Ensure consistency of the language version**. In cases of AOP extensions to Java, there is the issue of which version of Java the language extends. For instance, CaesarJ is available as an extension to Java 2 only, while Object Teams is available as an extension to Java 5 and there are multiple AspectJ versions that extend all the main versions of Java from Java 2 to and including Java 6. Since Java 2 is the common denominator, the need may occasionally arise to translate examples to a coding style compatible with Java 2 to enable comparisons with the various languages.

8. CONCLUSION

This report presents the first results of an ongoing effort to develop a repository of implementations of the GoF patterns in CaesarJ. Examples for seven patterns are presented and a short comparative analysis is made with an existing AspectJ repository. The experience gained

suggests that CaesarJ' features to deal with static structure, namely family polymorphism and a clear separation of implementations and bindings, lead to a more flexible management of the constituent parts of a component and avoids overusing pointcuts and advice, which are used primarily for glue code.

From the experience gained in developing the examples described in this report, as well as carrying out the preliminary analysis, we conclude that the language model CaesarJ proposed by CaesarJ provides better support for modularization and flexible module reuse that mainstream OO languages and AspectJ. However, more systematic studies are required to obtain clear evidence of these claims. This report provides a few suggestions on how that could be done.

9. ACKNOWLEDGEMENTS

Work described in this report was carried out in the context of a scholarship *Bolsa de Iniciação Científica* (BIC) to initiate undergraduates in research projects. The scholarship was financed by project SOFTAS (POSC/EIA/60189/2004), workpackage PLAP (Programming Languages for Aspect-Programming).

10.REFERENCES

[1] Aspectj project home page, http://www.eclipse.org/aspectj/.

[2] eclipse archive download page, http://archive.eclipse.org/eclipse/downloads/index.php.

[3] AspectJ project, http://www.eclipse.org/aspectj/.

[4] CAESARJ homepage, http://caesarj.org.

[5] Aracic I., Gasiunas V., Mezini M., Ostermann K. An Overview of CaesarJ. Transactions of Aspect-Oriented Software Development, vol.I, Springer, 2006.

[6] Baumgartner G., Läufer K., Russo V. F. On the interaction of Object-oriented Design Patterns and Programming Languages. Technical report CSD-TR-96-020, Purdue University, Feburary 1996.

[7] Bracha G., Cook W. *Mixin-Based Inheritance.* Proceedings of ECOOP/OOPSLA 1990, pp.303-311, 1990.

[8] Brichau J., Haupt M. Report describing survey of aspect languages and models. AOSD-Europe Deliverable D12, AOSD-Europe-VUB-01, May 2005.

[9] Chambers C. Object-oriented multi-methods in Cecil. ECOOP '92, Utrecht, The Netherlands, 1992.

[10] Eckel B. Thinking in Patterns, revision 0.9. Book in progress, May 20, 2003. Available at 64.78.49.204/TIPatterns-0.9.zip

[11] Ernst E. *Family polymorphism*, Proceedings of ECOOP 2001 (Heidelberg, Germany) (Jørgen Lindskov Knudsen, ed.), LNCS 2072, Springer-Verlag, June 2001, pp. 303–326.

[12] Gamma E., Helm R., Johnson R., Vlissides J. *Design patterns: Elements of reusable object-oriented sofware*, Addison Wesley, 1995.

[13] Garcia A., Sant'Anna C., Figueiredo E., Kulesza U., Lucena C., Staa A. Modularizing Design Patterns with Aspects: A Quantitative Study. LNCS TAOSD I, Springer vol. 3880, 2006.

[14] Gasiunas V., Mezini M., Ostermann K. Dependent types. ooPSLA 2007, Montréal, Canada 2007.

[15] Gudmundson S., Kiczales G., Addressing practical software development issues in AspectJ with a pointcut interface. Workshop on Advanced Separation of Concerns at ECOOP 2001, June 2001.

[16] Hannemann J., Kiczales G. *Design pattern implementation in Java and AspectJ*, Proceedings of OOPSLA 2002, pp. 161–173, 2002.

[17] Herrmann S. Object teams: Improving modularity for crosscutting collaborations. Net. Object Days, Erfurt, Germany, 2002.

[18] Hirschfeld R., Lämmel R., Wagner M. Design Patterns and Aspects – Modular Designs with Seamless Run-Time Integration. 3rd German GI Workshop on AOSD, 2003.

[19] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G. *An Overview of AspectJ*. Proceedings of ECOOP 2001, pp. 327-353, Budapest, Hungary, June 2001.

[20] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J. *Aspect-Oriented Programming*. Proceedings of ECOOP'97, Jyväskylä, Finland, Springer-Verlag Lecture Notes in Computer Science, vol. 1241, pp. 220-242, June 1997.

[21] Lieberherr K., Lorenz D.H., Mezini M. *Programming with Aspectual Components*. Technical report NU-CCS-99-01, Northeastern University, Boston, USA, March 1999.

[22] Lieberman H., Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. Proceedings of OOPSLA'86, Portland, USA, ACM press, pp. 214-223, September 29-October 2 1986.

[23] Madsen O. L., Moller-Pedersen B., *Virtual classes: a powerful mechanism in object-oriented programming*. Proceedings of OOPSLA'89, pp. 397–406, New Orleans, Louisiana, United States, 1989.

[24] Mezini M., Ostermann K. *Conquering aspects with Caesar*. Proceedings of AOSD 2003, pp. 90–99, Boston, USA, March 2003.

[25] Mezini M., Ostermann M. *Integrating independent components with on-demand remodularization*, Proceedings of OOPSLA 2002, pp. 52–67, 2002.

[26] Mezini M., Ostermann K. *Untangling Crosscutting Concerns with Caesar*. Chapter 8 of *Aspect-Oriented Software Development* (Filman R. E., Elrad T., Clarke S., Aksit M., editors), Addison Wesley 2005.

[27] Miles R. AspectJ Cookbook. O'Reilly 2004.

[28] Monteiro M. P., Fernandes J. M., An illustrative example of refactoring object-oriented source code with aspect-oriented mechanisms. Software: Practice and Experience 38(4): pp.361-396, 2008.

[29] Monteiro M. P., Fernandes J. M., Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns. DSOA'2004 workshop, Málaga, Spain, 2004.

[30] Monteiro M.P., Fernandes J.M. Towards a Catalogue of Refactorings and Code Smells for AspectJ. LNCS TAOSD I, Springer vol. 3880, 2006.

[31] Myers N. C. *Traits: a new and useful template technique*. In C++ Report, June 1995. http://www.cantrip.org/traits.html

[32] Rajan H., Design Patterns in Eos, PLoP '07, Monticello, Illinois USA, September 2007.

[33] Schwaninger C., Groher I., Meunier R., Hohenstein U. Empirical Study for Evaluating Evolvability Requirements. LATEr workshop at AOSD 2006, Bonn, Germany, 20 March 2006.

[34] Suvée D., Vanderperren W., Jonckers V. *JAsCo: An Aspect-Oriented Approach Tailored for Component-Based Software Development*. Proceedings of AOSD'03, pp. 21–29, 2003.