Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Tese de Mestrado em Engenharia Informática
1º Semestre, 2008/2009

A Qualitative Assessment of Modularity in CaesarJ components based on Implementations of Design Patterns

Sérgio Alexandre Esteves Miranda Braz, aluno nº. 26316

Orientador
Prof. Doutor Miguel Pessoa Monteiro

20 de Fevereiro de 2009

Nº de aluno: 26316
Nome: Sérgio Alexandre Esteves Miranda Braz

Título da dissertação:
A Qualitative Assessment of Modularity in CaesarJ components based on Implementations of Design Patterns

Palavras-Chave:

- CaesarJ
- Padrões de Concepção
- Programação Orientada por Aspectos
- Polimorfismo de Família
- Modularidade

Keywords:

- CaesarJ
- Design Patterns
- Aspect-Oriented Programming
- Family Polymorphism
- Modularity

Acknowledgements

First of all, to my supervisor, Prof. Dr. Miguel Pessoa Monteiro, for an unbelievable presence, support and motivation. I have been truly fortunate to work with such an interested and participating supervisor who has driven me to expand my knowledge and encouraging me to learn about an array of concepts I was not familiar with and have greatly enjoyed discovering.

To my Guinness drinking "lads", André Sabino, Ricardo Cebola and Rui Nóbrega. Nothing better than a pint of Guinness to put things in perspective. Specially to André Sabino and Rui Nóbrega for always pushing me to do better and helping with my thesis. Were due for some pints.

To my "spam list" friends, Abel Campião, Arthur Teixeira, Bruno Miranda, Bruno Monteiro, Hugo "Janado" Almeida, Hugo "Guimbas" Teixeira, José Grilo, Miguel Campião, Pedro Barradas, Ricardo Nascimento and Rui Barros. For not letting me take myself to seriously, for making fun at me and themselves and for filling my mail box with the most incredible, nonsensical, borderline juvenile humor. Basically, for making me eagerly anticipate the next joke and allowing me to participate.

To my friends João Santos and Sérgio Matos. It feels funny to thank you, especially because you would make fun of me. Anyway… Thanks!

To my long lasting friend Pedro Palrão, a thank you as well as an apology. An apology for practically disappearing from the map. A thank you for not giving up on me.

To my parents, Carlos and Mila, and my little sister Sara. For understanding the fact that I couldn't be with them as much and reminding me that I always have a place at home, even at times when we shared the same roof but did not see each other due to my hectic schedules. For being family, at all times.

To Bruna, for always being present, for the encouragement, for telling me what I needed to hear instead of what I wanted to. For getting me through to the end. For being more than just my girlfriend, for being my best friend.

Resumo

O aparecimento do paradigma da Programação Orientada por Aspectos (AOP) trouxe novas funcionalidades e mecanismos para dar suporte à separação de conceitos transversais, de modo a desenvolver programas mais modulares e consequentemente, mais reutilizáveis. Com o amadurecimento deste paradigma, surgiram várias linguagens de programação para dar corpo aos conceitos por ele avançados. Entre essas linguagens encontra-se a linguagem CaesarJ.

Enquanto a grande maioria dos estudos práticos sobre AOP se têm focado na linguagem AspectJ, as características de outras linguagens como o CaesarJ continuam por explorar. A falta de investigação sobre a utilização do CaesarJ em casos concretos leva a que haja poucos casos de estudos a partir dos quais retirar elações sobre os seus pontos fortes e fraquezas.

No passado, implementações de padrões de concepção têm sido utilizadas para a demonstração das características de linguagens de programação. Esta dissertação adopta uma abordagem semelhante com o intuito de aferir a o suporte do CaesarJ a modularidade e reutilização por meio da implementação de padrões de concepção e subsequente análise quantitativa.

Esta dissertação apresenta implementações em CaesarJ de onze padrões do Gang-of-Four, que serviram de base a uma análise qualitativa sobre o grau de modularidade que o CaesarJ consegue atingir nestes padrões. É feita uma distinção entre quatro níveis de reutilização de módulos que as implementações suportam, de modo a diferenciar entre os diversos níveis de reutilização atingidos. É feita uma comparação com as implementações análogas de padrões em AspectJ. Finalmente, são descritas algumas direcções sobre a concepção de componentes em CaesarJ.

Abstract

The advent of the Aspect-Oriented Programming (AOP) paradigm brought new features and mechanisms to support the separation of crosscutting concerns, in order to develop programs with higher modularity and consequently, higher reuse. As the paradigm matures, various aspect-oriented programming languages appeared that propose varying ways to realize the paradigm's concepts. CaesarJ is one of those aspect-oriented languages.

While the majority of practical studies on AOP languages focused on the AspectJ language, the characteristics of other languages such as CaesarJ remain to be explored. The lack of research on the utilization of CaesarJ in concrete cases leads to the existence of few case studies from which to draw considerations about their strengths and shortcomings.

In the past, implementations of design patterns have been used for the demonstration of the characteristics of the programming languages used to implement them. This dissertation follows a similar approach to assess CaesarJ's support for modularity and reuse by producing CaesarJ design patterns implementations and subjecting those implementations to a qualitative analysis.

This dissertation presents CaesarJ implementations of eleven Gang-of-Four pattern that serve as the basis for a qualitative analysis of the modularity degree CaesarJ enables for each pattern. A distinction is made between four levels of module reuse that the implementations support, in order to differentiate between the several levels of reuse achieved. A comparison is drawn to analogue design pattern implementations in AspectJ. Finally, general guidelines for the implementation of CaesarJ components are described.

x

Index

Index of Figures

Index of Tables

Index of Listings

1. Introduction

Since its appearance in 1997, the paradigm of Aspect-Oriented Programming (AOP) is steadily growing a common subject of research. Much research has been done on the characteristics of (AOP) relatively to Object-Oriented Programming (OOP) and the modularity improvements it provides [14][22][26][27][28][45][48]. These efforts have contributed to a growing maturity of the paradigm. However, the vast majority of studies concerning AOP have mostly been focused in a single programming language, AspectJ[1]. Although AspectJ was the first AOP language to be developed, a great number of languages have been developed afterwards and offer alternatives to AspectJ [11]. Among those languages is CaesarJ[2]. This dissertation presents a study on CaesarJ based on the implementation of design patterns, as an effort to further increase the knowledge regarding this particular representative of AOP and the knowledge about the languages that have been created to embrace this paradigm.

The rest of this chapter is structured as follows: section 1.1 develops on the motivation behind this dissertation; section 1.2 describes the problem this dissertation aims to solve; section 1.3 describes the approach chosen to tackle this issue; section 1.4 lists the contributions of this thesis and section 1.5 ends by presenting the outline of the rest of this document.

1.1 Motivation

Aspect-Oriented Programming (AOP) [32] and aspect-oriented software development (AOSD) [3] have risen in software engineering with the purpose of assisting programmers in

[1] http://www.eclipse.org/aspectj/

[2] http://caesarj.org/

[3] http://aosd.net/

the separation of concerns, particularly crosscutting concerns. This new paradigm led to the creation of several aspect-oriented languages [11], namely CaesarJ and AspectJ. These new languages brought many advances in the modularization of programs, thereby enhancing reuse and other benefits [34] such as:

- *Cleaner responsibilities of individual modules* which is a consequence of code locality of crosscutting concerns.
- *Easier system evolution* due to the implementation of crosscutting concerns into specific modules that can be added to existing core modules without the need to change them.
- *Late binding of design decisions* because future requirements can be implemented in a separate module and later be flexibly introduced into a system.

However, not all languages have enjoyed equal development, research, support or general acceptance in the programming community. While AspectJ has been the dominant language in AOP, some studies show it still has some problems to be solved. These problems are related to limitations to code reuse [38] and poor aspect structure [41][45] which leads to integration issues. Meanwhile, CaesarJ's features and characteristics remain to be properly explored.

Relatively to AspectJ, currently AOP's most popular language, CaesarJ still lacks research and experimentation to assert its strengths and shortcomings. As a consequence, the properties of the CaesarJ language constructs and concepts have not been properly assessed so far. This dissertation aims to contribute to an assessment of CaesarJ, its concepts, mechanisms and capabilities for separation of crosscutting concerns, modularity and reuse.

For the definition of modularity in the context of AOP, this dissertation takes the definition provided by Kiczales et al. [33]. Kiczales el al. state that a code implementing a concern can be considered modular if:

- it is textually local;
- there is a well-defined interface that describes how it interacts with the rest of the system;
- the interface is an abstraction of the implementation, in that it is possible to make material changes to the implementation without violating the interface;

- an automatic mechanism enforces that every module satisfies its own interface and respects the interface of all other modules;
- the module can be automatically composed – by a compiler, loader, linker etc. – in various configurations with other modules to produce a complete system;

CaesarJ presents a new way of looking into modularization and offers new conceptual language modules. It also makes use of mechanisms like virtual classes [36] and family polymorphism [18] which are absent in AspectJ. These mechanisms are described in section 3.3.

Nowadays, the AOP paradigm is characterized as a systematic approach to modularity [46]. Consequently, the study of an AOP language can be justified by an assessment of its support for modularity. The motivation behind this dissertation is to develop case studies which will serve as the basis for an analysis of CaesarJ's support for modularity characteristics, strengths and shortcomings. This analysis will focus on CaesarJ's abilities to produce reusable modules, distinguishing between 4 levels of reuse. A short assessment of the composition abilities of such modules is presented. A comparison between the CaesarJ and AspectJ pattern implementations [28] is established as far as their reuse capabilities. Finally, some guidelines for the development of CaesarJ components are provided based on the experience gained from the case studies.

It is important to note that, although this thesis only mentions languages that are AOP extensions to the Java programming language, there are other AOP languages that extend other object-oriented languages such as AspectC++ and AspectS for C++ and Smalltalk, respectively, or other paradigms like AspectC and AspectML, for C and ML, procedural and functional programming languages, respectively. However, this dissertation will focus mainly on CaesarJ, using AspectJ for comparison purposes and on Java since it is the basis for both languages.

1.2 Problem description

Currently, there are few case studies on the CaesarJ language. The same is not true for AspectJ, since it is the most popular language in AOP so far.

Much research on programming languages has been based on the implementation of the 23 Gang-of-Four (GoF) design patterns [21] and their consequent analysis. Design patterns

present common problems that can be found in large and complex systems and the corresponding solutions. Many implementations of the GoF patterns have been collected into various repositories. Table 1 presents some examples collected for the realization of this dissertation. These researches have led into case studies both in Object-Oriented Programming (OOP) languages like Java, and in AOP. As far as AOP is concerned, there are only four repositories available for studying, and only three of them are freely available [28][1][4]. Although some studies on CaesarJ have been based on this design patterns, there is currently no complete repository with CaesarJ implementations of all design patterns.

The lack of practical CaesarJ case studies, particularly CaesarJ implementations of the GoF design patterns, deters studies to be carried out. Such studies would enable a deeper understanding of its language features and possibilities as far as the support for separation of crosscutting concerns, the capacity for the creation of composable modules and the reusability of those modules.

1.3 Presented solution

Repositories of the well known design patterns have provided suitable case studies for subsequent research. Design patterns present the advantage that patterns can be approached one at a time concentrating in a single problem and its characteristics. Each solution to a given problem a pattern potentially solves provides insights on the language on which the implementation is made and about its features.

This dissertation will take some existing Java repositories of the design patterns and create new implementations of those scenarios on CaesarJ. These repositories are freely available online, and are implemented in Java 2, the Java version currently supported by CaesarJ. The choice to implement CaesarJ scenarios from independently developed Java repositories instead of creating completely new CaesarJ implementations brings the benefits of greater result independence and lesser bias probability. Table 1 lists the group of independent repositories chosen as examples for the implementation of the CaesarJ design patterns.

Each repository presents different styles of programming which are reflected in the implementation of the design patterns. Each design pattern implementation is called a *scenario*. A scenario is the application of a design pattern to a concrete situation. This set of

Repository reference name	Author(s)	Repository URL
Thinking in patterns	Bruce Eckel	http://www.mindviewinc.com/downloads/TIPatterns-0.9.zip
Design pattern Java companion	James Cooper	http://www.patterndepot.com/put/8/JavaPatterns.htm
Fluffy Cat	Larry Truett	http://www.fluffycat.com/Java-Design-Patterns/
Hannemann et al.	Jan Hannemann and Gregor Kiczales	http://hannemann.pbwiki.com/Design+Patterns
Huston	Vince Huston	http://www.vincehuston.org/dp/
Guidi Polanco	Franco Guidi Polanco	http://eii.ucv.cl/pers/guidi/documentos/Guidi-GoFDesignPatternsInJava.pdf

Table 1 Gang-of-Four Java repositories used for CaesarJ implementations

repositories presents scenarios that span across several language constructs to implement the design patterns. Some scenarios illustrate design pattern producing text based examples, others resort to Java API classes to implement the design patterns or the participant roles in the pattern while others illustrate the design pattern through graphical interfaces using Java API objects. This range of different scenarios is meant to expose CaesarJ to a diversity of situations.

For comparison purposes between CaesarJ and AspectJ, scenarios from the Hannemann et al. repository have been privileged but, at every time, a minimum of two CaesarJ scenarios per pattern was developed. The reason for producing several implementations for the same pattern is to correctly assert if a module can be reused in several situations.

By offering new implementations based on CaesarJ, this dissertation aims to help increasing the knowledge about CaesarJ and its features, as well as establishing a basis for comparisons with other programming languages.

This dissertation's goal is to make an analysis on CaesarJ's strengths based on 4 hierarchical criteria of reusability and modularization. Each implementation is evaluated to **(1)** whether CaesarJ provides direct language support to this specific design pattern problem; **(2)** if some reusable module can be produced; **(3)** if, although not reusable, this problem can be modularized into a module with composition flexibility or **(4)** if the facet cannot be modularized at all. Through this evaluation, it should be possible to distinguish between these four levels of reuse, where direct language support to a pattern constitutes the highest level since the mechanisms are embedded in the language itself.

In the cases when patterns with reusable modules are produced, several different situations are tested regarding these modules' capabilities. For evaluation purposes, the same module is **(1)** composed in multiple scenarios; **(2)** if it can be composed several times in the same system; **(3)** if different implementations of the same module can coexist within the same system compatibly.

If patterns with reusable modules or modules with composition flexibility are produced, these patterns are subjected to a composability assessment. They are tested whether the module has **(1)** the ability to be composed only to selected instances of a class; **(2)** if the composition order is observable; **(3)** if the modules can be easily deployed and undeployed into existing applications.

For every pattern that has given rise to a reusable module or a module with composition flexibility, a scenario was developed to further evaluate each pattern whose CaesarJ implementations demonstrated such properties.

1.4 Contributions

The contributions this dissertation brings are present next:

- **Thirty six implementations of GoF design patterns.** These implementations are the basis of the entire thesis. They provide the grounds for the theoretical considerations themselves and serve as case studies for future work on CaesarJ.
- **Class diagrams documenting the CaesarJ pattern implementations.** These diagrams provide illustrations for the CaesarJ implementations and their general structure. Also, these diagrams are useful in the comparison between the produced CaesarJ implementations and the original Java design patterns implementation.
- **An analysis of the CaesarJ pattern implementations.** This analysis comprises:
    - An analysis regarding the level of reuse attained in the implementation of each pattern, differentiating between four levels of reuse.
    - An analysis on the composition capabilities of each pattern that derived a module deemed as reusable or as a module with composition flexibility.
    - A general comparison of the implementation of design patterns in CaesarJ and in AspectJ.
- Guidelines for the design of a CaesarJ component.

1.5 Document outline

The rest of the dissertation is as follows: Chapter 2 describes the patterns approached for the pattern implementations; chapter 3 presents the CaesarJ programming language; chapter 4 discusses previous studies that are directly related to this dissertation; chapter 5 describes the CaesarJ pattern implementations; chapter 6 presents the analysis of the CaesarJ pattern implementations; chapter 7 mentions related work and chapter 8 ends by presenting this dissertation’s conclusions and future work.

2. Design Patterns

Patterns originated in Alexander's work in architecture [5]. Later, this concept was adopted to *design* decisions in object-oriented programming by Gamma et al. [21], thus giving birth to *design patterns*. This group of four authors has become known as the Gang-of-Four (GoF). A *design pattern* is the description of common and recurring problems in software engineering, the general outline of the possible solutions to these problems, the context within which these solutions work and the implications of those solutions.

The GoF design patterns are the most well known and popular design patterns. They are the result of the study of real frameworks, and are a catalog of common programming and design practice. This catalog serves as a knowledge repository that enables software developers to choose between proven solutions without the need to reinvent them. The focus of this catalog is to provide developers with solutions that are flexible and reusable, and to offer a set of choices that can be adopted to different contexts, making reusable object-oriented software design more productive. Patterns provide value because they capture design knowledge [25] and document it in a methodic way that makes this knowledge easily approachable to software designers. Although there is other literature concerning design patterns [16][44], only the GoF design patterns will be considered within the context of this thesis.

The rest of this chapteris structured as follows: sections **Erro! A origem da referência não foi encontrada.** to 2.3 depict the structure of the description of design patterns, their organization and classification, their relation to frameworks and idioms differentiating their level of granularity; section 2.4 states the benefits that come from the study of design pattern; sections 2.5 through 2.15 describe the patterns selected for the implementations of this thesis, beginning with a transcript of the pattern's intent as expressed by Gamma et al [21]. Section

2.16 concludes by presenting a short enumeration of the patterns and the aspects they should allow to vary, i.e. their variation points.

2.1 Format of the description of design patterns

There are 23 design patterns in the GoF catalog. Their description is comprised of several parts, namely:

- **Name;** naming a pattern is an obvious requirement. Nevertheless, a pattern name is the first description of the pattern. It should be representative of problem it describes and allows us to identify a concrete pattern unambiguously.
- **Intent and applicability;** the pattern intent is the problem it aims to solve. It briefly describes the problematic situation and how this pattern tries to solve it. The applicability of a pattern are the concrete situations where it can be applied. These situations reflect design decisions that can be implemented in multiple cases.
- **Structure, participants and collaborations;** the structure of a pattern is usually a diagram representation of the abstractions involved in the pattern. The participants are the classes and/or objects that participate in the design pattern and what role they play within the pattern. The collaborations between participants described in a pattern describe the way the abstractions interact to fulfill the responsibilities they carry out within the pattern.
- **Consequences and implementation;** the consequences are the trade-offs the use of the pattern reflects in the overall design of the system. The implementation of a pattern is related to the actual techniques involved when implementing a pattern. The implementation techniques are related to the programming language used to implement the design pattern.
- **Sample code;** a design pattern is illustrated with a sample code of a situation where the pattern can be efficiently used.

2.2 Design pattern organization and classification

Patterns are often related with each other, and can sometimes be used together. There are many situations when patterns address the same problem or can be used together to complement each other with different functions.

Figure 1 is taken Gamma et al. [21] and illustrates the 23 GoF patterns and the relations between them.

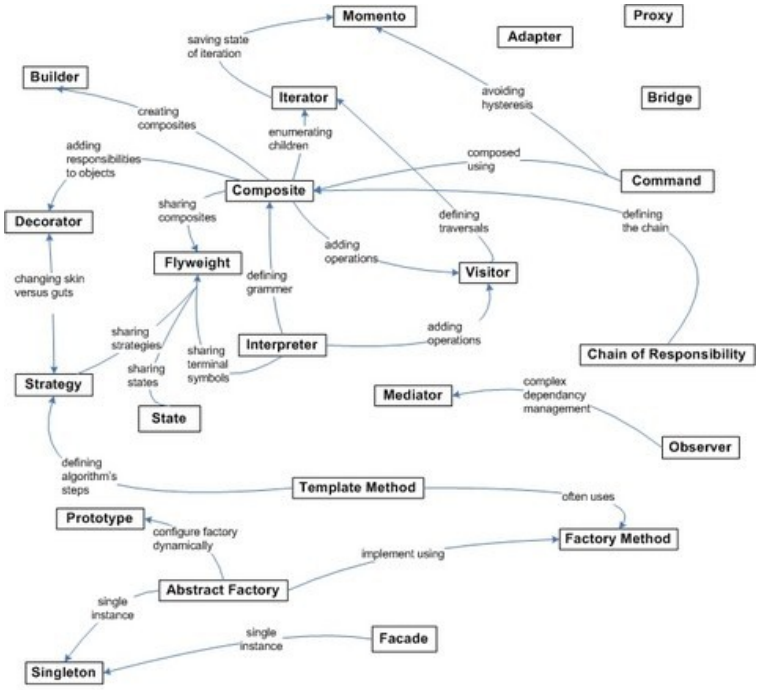

Figure 1 Design pattern relationships

Depending on the situations they address, the patterns can be organized according to two criteria: purpose and scope.

The purpose of a pattern is simply what problem it addresses. Based on the purpose criterion, patterns can be divided into creational, structural and behavioral. Creational patterns address the activity of object creation. Structural patterns tackle the composition of classes or objects. Behavioral patterns describe how classes or objects interact and distribute responsibility.

The scope criterion reflects whether the pattern is aimed for classes or objects. Class patterns deal with the relationships between classes and their subclasses, namely through

inheritance. Since inheritance relationships are determined at compile time, class patterns are typically static by nature. On the other hand, object patterns are concerned with object relationships which can be changed during run time and therefore, are more dynamic.

Table 2 is adapted from Gamma et al. [21] and reflects the categorization of the 23 patterns according to these criteria.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter
				Template Method
	Object	Abstract Factory	Adapter	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Table 2 Pattern classification table

2.3 Relation to idioms and frameworks

It is also important to discuss the abstraction level of design patterns by comparing them to *idioms* and *frameworks*. Design patterns can be placed in an intermediary level between frameworks (that represent a concrete architectural design) and idioms, which are language specific.

Buschmann et al. have defined idioms as low level patterns specific to a programming language [12]. Idioms tell us how to solve a particular implementation situation using the features of a concrete language. A programming language tutorial is an example of a collection of idioms. Tutorials tell inexperienced developers how a particular language implements a given situation, making the best use of its features.

Like design patterns, idioms have names which allows developers to identify them and refer to a well defined situation. However, unlike design patterns, idioms are not easily reproduced between languages. This reflects the fact that implementations depend on the

features of the language used, which has an impact on the idiom. Design patterns differ because they are not specific to a particular language. While a pattern implementation does depend on the language used to implement it, design patterns are not concerned with implementation issues but on general structure principles that do not depend on a particular language.

Frameworks are placed in the opposite level of idioms. They are the realization of architectural patterns. Architectural patterns specify the fundamental structure of an application [12]. Architectural patterns attempt to give an entire system a certain property like adaptability of the user interface. A framework has several subsystems that must communicate and collaborate among themselves within the architectural structure defined by the architectural pattern. Design patterns perform this function. They express how the components in a framework should interact in a reusable fashion so that the framework is itself reusable.

Applications have often originated from the use of frameworks, thus conforming to its design and collaboration model. This way, frameworks cause patterns in the applications that use them repeatedly [31]. This is what made possible the discovery and classification of the GoF design patterns. As far as granularity, design patterns are comparatively smaller than frameworks, and their application in a framework doesn't have an impact in the system as a whole [12]. Nonetheless, they might have an influence on the architecture of a given subsystem of the framework.

2.4 Benefits from the study of design patterns

Due to their importance and popularity, the GoF design patterns have led to research in many different areas of software engineering. Since the implementation of a pattern is influenced by the programming language used to implement it [21], they can be used to study language characteristics. Some studies have proven that patterns influence language mechanisms and suggested they should be part of their constructs [8][9].

Since the collection of 23 patterns isolates individual design problems and solutions, other studies have focused on the assessment of the features of existing programming languages. By tackling each pattern separately, pattern implementation constitutes a practical

way of drawing conclusions on the capability of those languages to solve particular design problems. As patterns focus on reusable design, an important question is how programming languages support modular design. Within the context of AOP languages, there has been some research on design patterns and aspects as far as their modularity [29][22], the advantages of aspect oriented pattern implementations over object oriented [28], of deficiencies patent in some aspect oriented implementations [41] and as illustrations for the comparison of different AOP languages [39][45].

Design Patterns often present crosscutting behavior because typically, they are composed of different roles. These roles communicate with each other, in a manner that should be as modular as possible, in order to enhance reuse. Hence, they are appropriate case studies for AOP languages.

Although there are 23 GoF patterns this thesis concentrates on 11 of those patterns. These patterns were selected because they were considered the most interesting regarding the characteristics of the studied language, the problem they try to solve and how these characteristics of the language might enhance the implementation of these patterns. Not all patterns present the same complexity and some are actually supported by some programming languages, like the *Iterator* pattern in the Java programming language, for example.

The selection criteria for the patterns reflects a preference towards the creational patterns and patterns that have been considered good examples of object oriented design that are better modularized using AOP techniques and have thus created reusable modules [28][22]. The preference for creational patterns is based on the interest of the study of CaesarJ's constructs for aspect structure, while the preference for design patterns that have shown improvements from AOP implementation is based by the interest in assessing CaesarJ's capability to produce reusable modules.

2.5 Abstract Factory

"Provide an interface for creating families of related or dependent objects without specifying their concrete classes".

The *Abstract Factory* is intimately related to the more recent concept of family polymorphism [18]. Both *Abstract Factory* and family polymorphism deal with the need to create families of related objects and aim to ensure that objects of different families do not mix.

The solution presented by the GoF is to defer the responsibility of creating objects of a particular family to a special object, the so called *factory object*. Different factory objects create objects with implementations specific to the factory that creates them, thus ensuring consistency between objects of the same family. If one needs to create objects with different implementations, this can be achieved by changing the factory object.

Figure 2 shows the structure of the pattern.

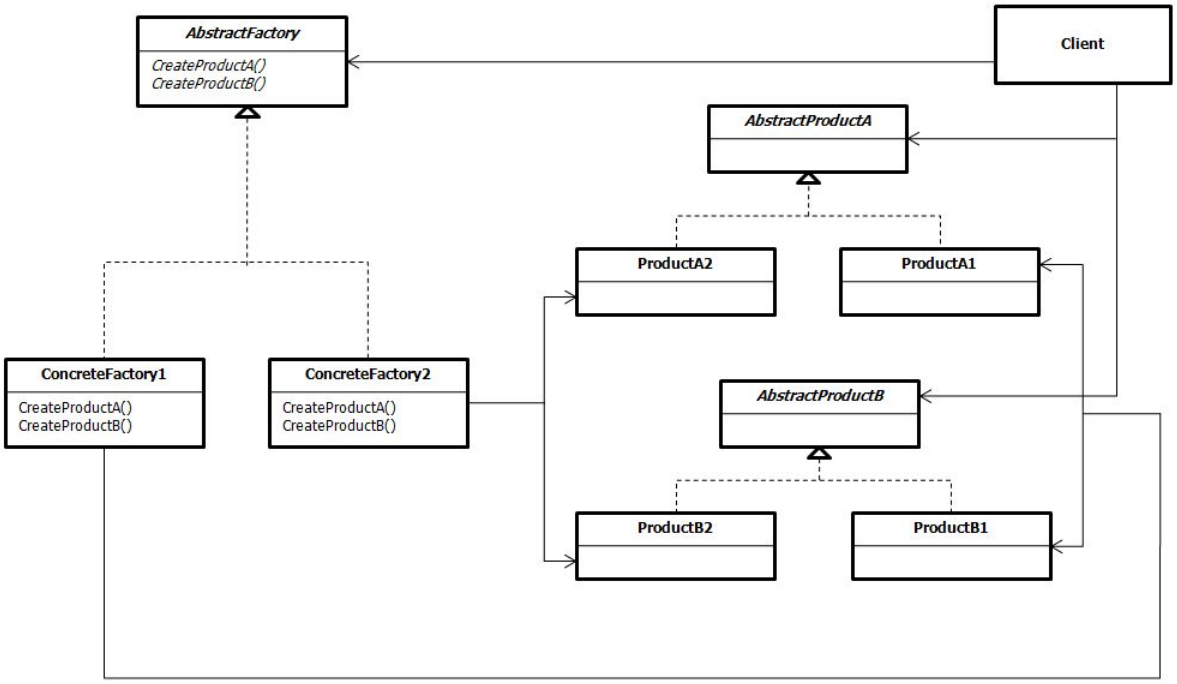

Figure 2 *Abstract Factory* pattern structure

In the *Abstract Factory* pattern there is an abstract class, *AbstractFactory*, which acts as the interface for the creation of objects of related families. The concrete classes that realize the creation of the actual objects of the family, such as *ConcreteFactory*, hide the implementation details for the creation of product objects in *CreateProduct* operations.

On the other side, different products are declared in product interfaces such as *AbstractProduct* and implemented in concrete classes *Product*.

Abstract Factory ensures consistency because concrete products are referenced by the concrete factories that define the families. It also enhances flexibility because clients only use the interfaces declared by *AbstractFactory* and *AbstractProduct* and concrete factory implementations can be swapped easily.

2.6 Bridge

"Decouple an abstraction from its implementation so that the two can vary independently".

The concern behind *Bridge* is to allow the development of the implementation of an abstraction in a more flexible way than inheritance so different implementations may be switched at run-time. Inheritance hierarchies bind an implementation to an abstraction permanently at compile-time which results in a static, inflexible option.

Gamma et al. [21] suggest that abstractions and implementations should belong to separate class hierarchies with the abstraction forwarding client requests to the implementation object through an implementation reference.

Figure 3 illustrates the *Bridge* pattern.

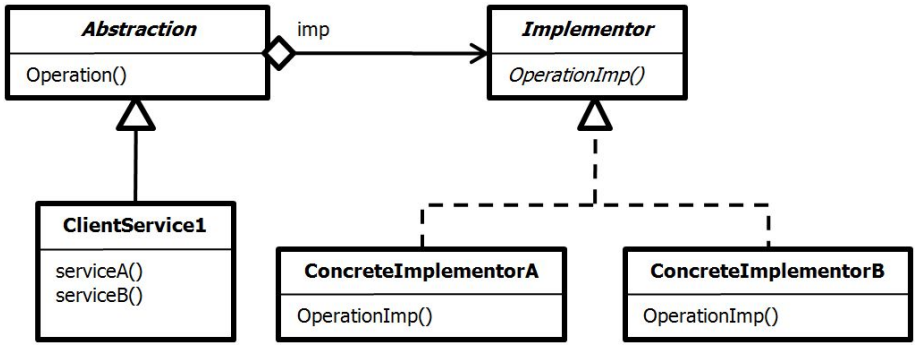

Figure 3 *Bridge* pattern structure

Class *Abstraction* defines the interface for the abstractions in the pattern. *Abstraction* keeps a reference to an *Implementor* object which is responsible for the execution of

operation *Operation*. When *Operation* is called, it forwards its execution to the *Implementor* object referenced the *Abstraction* and executes operation *OperationImp*.

Subclasses *ConcreteImplementor* are responsible for the implementation of operation *OperationImp*. If the implementation of operation *Operation* needs to be change, this can be achieved by changing the referenced *ConcreteImplementor* object.

ClientService classes simply extend the interface defined by *Abstraction*.

This pattern illustrates the object oriented tendency to favoring object composition over class inheritance.

2.7 Builder

"Separate the construction of a complex object from its representation so that the same construction process can create different representations".

Builder is different from other creational patterns because it deals with the process of creating objects step by step instead of all at once. The separation of representation and construction process gives additional control over the creation of complex objects to the developer. This way, different final products can be created simply by changing the parts of the object that are created, the order by which the parts of the object are created, or their implementations.

The *Builder* pattern structure is described in Figure 4.

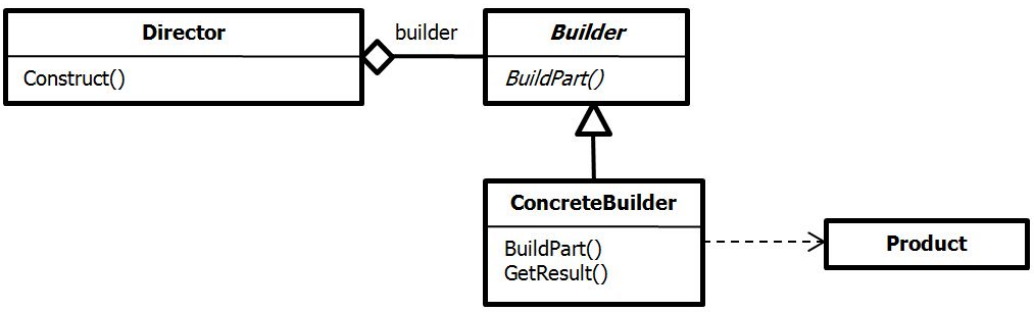

Figure 4 *Builder* pattern structure

The abstract class *Builder* acts as an interface for building different parts of the object through the *BuildPart* operation. This operation is implemented in the *ConcreteBuilder* subclasses. Each *ConcreteBuilder* subclass is responsible for the creation of specific different parts of the complete product object. The *Product* class represents the parts each *ConcreteBuilder* creates and holds their representation. Later, *ConcreteBuilder* classes are also responsible for the retrieval of the product representation using operation *GetResult*.

The *Director* class keeps a reference to a *Builder* class which is the responsible for the creation of the parts of the final product object. When a different *Product* must be created, this can be achieved by changing the *ConcreteBuilder* referenced by the builder reference. It also keeps a structure where the different parts of the object are stored.

Builder increases modularity because the code responsible for creating new products is encapsulated in the *ConcreteBuilder* classes. This allows different representations to be added easily simply by adding another *ConcreteBuilder* class responsible for creating a new product.

2.8 Chain of Responsibility

"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it".

If a request is tightly couple to a receiver, no other receivers are given the chance to handle that request. Decoupling senders from receivers of requests allows for a flexible definition of which receiver should handle a particular request. This approach allows multiple receiver objects to handle a request.

The request passes along a sequence of objects until it reaches the appropriate handler. This sequence is called the chain of responsibility. The request is passed along this chain, from the most specific handler to the most general, until it is handled or reaches the end of the chain.

Figure 5 further describes the pattern.

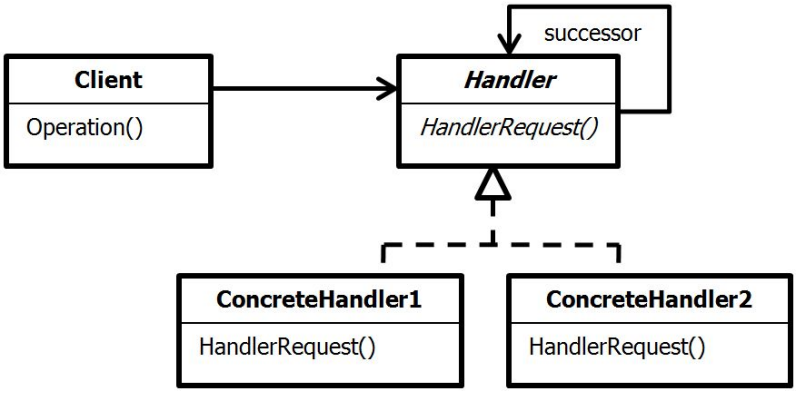

Figure 5 *Chain of Responsibility* pattern structure

Objects of class *Client* initiate the request to handlers in the chain of responsibility. The chain of responsibility is composed by objects with a common interface class, class *Handler*. This class defines an operation for dealing with the request, operation *HandlerRequest*, and has a reference to the successor in the chain. Different *ConcreteHandler* classes assume the responsibility of handling specific requests. If the request is this *ConcreteHandler*'s responsibility, the *ConcreteHandler* handles the reuqest. If not, it passes the request to the following *ConcreteHandler* in the chain of responsibility.

With the *Chain of Responsibility* pattern, handlers are concerned with the way how they handle the requests they are responsible for, without the concern of the chain's structure. Also, because sender objects only keep references to their successor, this makes interactions between sender and receiver objects simpler. Finally, the *Chain of Responsibility* pattern also enhances flexibility because the structure of the chain can be dynamically changed without impact on the system.

2.9 Composite

"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly".

The *Composite* pattern uses recursive composition to treat primitive and container objects in the same way. This makes dealing with tree structures a simpler task. To achieve this effect, the *Composite* pattern defines an abstract class that represents both atomic parts and

their containers. This abstract class declares operations common to both primitive and container classes, as well as operations for accessing and managing composite object's children.

Figure 6 exemplifies the pattern's structure.

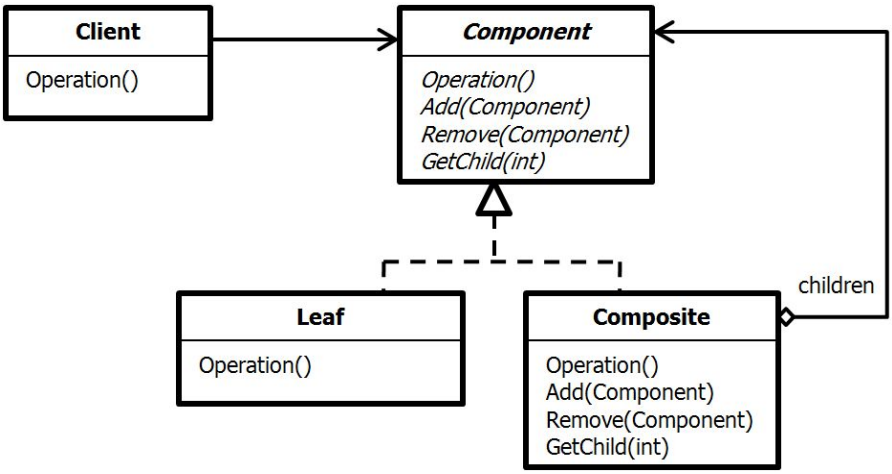

Figure 6 *Composite* pattern structure

The *Component* abstract class is the key to the pattern, since it declares the interface for the objects in the composition of the tree structure, whether they are *Leaf* or *Composite* objects. *Leaf* and *Composite* objects differ because the former has no children while the latter can have multiple children.

The *Component* class declares an operation *Operation*, with the behavior common to all classes and operations *Add*, *Remove* and *GetChild* to deal with the children of *Composite* objects.

Leaf objects simply define the behavior for primitive objects in the tree structure with the *Operation* operation.

Composite objects store several objects that can be both of *Leaf* or *Composite* classes, thanks to a data structure that references an undetermined number of *Component* objects. Operations *Add*, *Remove* and *GetChild* manage the storage of *Component* child objects and the *Operation* operation traverses the structure that stores *Component* objects. If they are *Leaf* objects, it calls the *Operation* operation of *Leaf* objects. If they are *Composite* objects, it calls *Operation* on that *Composite*'s children.

The *Client* objects access the composite structure through the *Component* class.

The *Composite* pattern makes it easier to add new kinds of components to applications, by making them conform to the interface defined by the *Component* class. Since clients access the structure through the *Component* class, this makes clients simpler because they don't need to know if they are dealing with a *Composite* or a *Leaf* object.

2.10 Decorator

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality".

Inheritance is typically used to add features to existing classes. This way, every instance of those subclasses exhibits the existing features of the super-class and the new features it defines. But since inheritance hierarchies are defined statically, this solution is inflexible. Additionally, it might be desirable that only some objects of a particular class have these added functionalities, not all instances.

The *Decorator* pattern addresses this problem by enclosing an object in another object that provides additional functionality. The enclosing object is called a decorator.

Figure 7 presents the structure for the *Decorator* pattern.

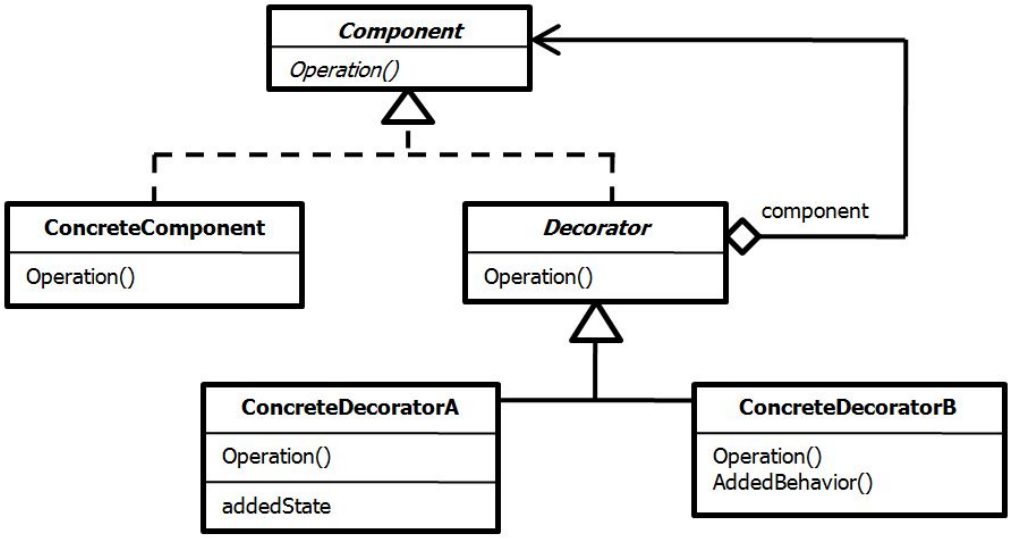

Figure 7 *Decorator* pattern structure

The *Component* class defines the interface of the objects that might be added with new responsibilities. These responsibilities might be additional members, methods or both and also additional behavior to existing operations. Instances of *ConcreteComponent* classes define objects that can be handled and later added with new responsibilities.

The *Decorator* abstract class defines an interface for classes that add new responsibilities that conforms to the interface defined by the *Component* class. It also keeps a reference to a *Component* object to which it forwards requests, namely through the *Operation* operation.

The *ConcreteDecorator* classes add the operations and state for the desired specific functionality dynamically.

An important aspect of the *Decorator* pattern is that it might be desirable to add several different functionalities to an instance of a *Component* object, regardless of the composition order, or that the same functionality might be added multiple times, in cases where this is. The *Decorator* pattern makes this process easier than inheritance.

2.11 Factory Method

"Define an interface for creating an object, but let subclasses decide which class to instantiated. Factory Method lets a class defer instantiation to subclasses".

The *Factory Method* is closely related to *Abstract Factory*. While *Abstract Factory* provides an interface for creating families of related objects, the *Factory Method* provides the interface for the instantiation of the appropriate objects that will ultimately compose those families. To this effect, the *Factory Method* provides a superclass with an abstract operation for creating individual objects and delegates the responsibility of creating the correct objects to the subclasses.

Figure 8 shows the structure of the *Factory Method* pattern.

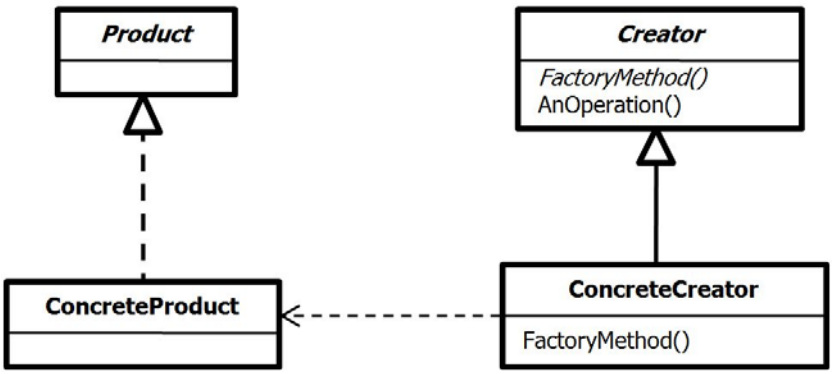

Figure 8 *Factory Method* pattern structure

The *Creator* abstract class declares the *FactoryMethod* operation. It returns an object of type *Product*. However, the *Creator* class doesn't know which *ConcreteProduct* object the *FactoryMethod* will return, unless it provides a default method implementation. The *ConcreteCreator* classes provide implementations that return instances of the appropriate related *ConcreteProduct*.

As described, the *Factory Method* makes the instantiation of new objects more flexible and independent of specific classes.

2.12 Mediator

"Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently".

Object oriented practices advise the encapsulation of individual concepts into objects that module these concepts and their responsibilities. When objects need functionalities from other objects, they should form connections among themselves. However, when too many connections are composed, the interactions between objects and the overall system become difficult to manage. The *Mediator* pattern bypasses this problem by creating an object that centralizes interactions between other related objects, controlling and coordinating them.

Participant objects only know of their intermediary object and interactions between them must pass through this intermediary object. This also leads to the reduction of the number of interconnections between objects, which makes the system easier to manage.

Figure 9 shows a representation of the *Mediator* pattern.

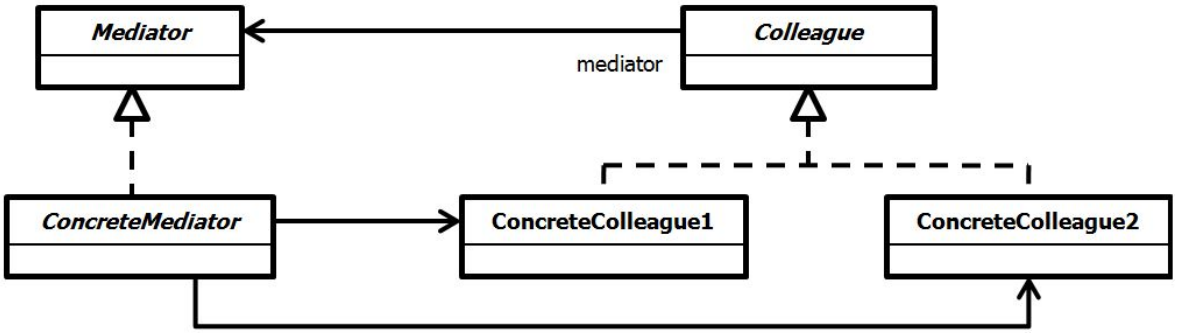

Figure 9 *Mediator* pattern structure

The *Mediator* class defines how *Colleague* objects communicate with each other. Each *Colleague* has a reference to its *Mediator* object. *Colleague* objects use this reference to communicate with the *Mediator* object that acts as the intermediary for its interactions with other *Colleague* objects.

In order to control interactions between *ConcreteCollegue* objects, *ConcreteMediator* keep a reference to each *ConcreteColleague* it serves as an intermediary.

The *Mediator* pattern brings advantages such as loose coupling and simpler communication between *Colleague* objects. The former characteristic makes reusing *Colleague* classes easier because objects are only concerned with their own behavior and not with the cooperation. Also, it makes the system easier to understand. However, the latter characteristic of the pattern has a downside to it, since it makes the *Mediator* object more complex because it centralizes all interaction protocols a single object.

2.13 Observer

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically".

The *Observer* pattern shares with *Mediator* the fact that both deal with managing interactions between groups of related objects in a consistent manner. However, as in the case of *Mediator*, this consistency must be obtained while still avoiding tight coupling between classes.

Observer solves the problem of decoupling objects that produces events of interest from objects that should be notified when those events happen. *Observer* offers a flexible solution in that it makes the connection between both kinds of objects without them making assumptions about each other.

Figure 10 illustrates the *Observer* pattern.

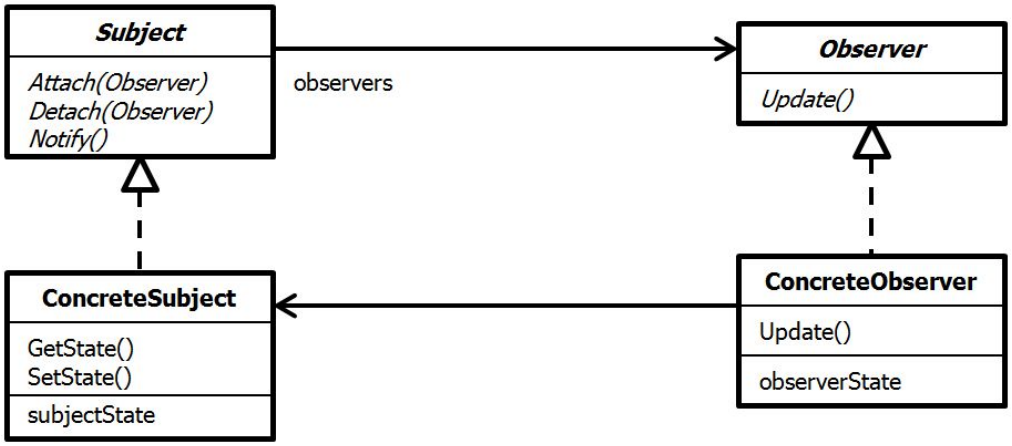

Figure 10 *Observer* pattern structure

The *Observer* pattern defines two roles, one for classes that create events of interest, other for classes that are notified of these events. These are the *Subject* classes, the classes that produce events of interest, and the *Observer* classes, the classes that must be notified of these events.

The *Subject* abstract class keeps references to *Observers* interested in changes in their own state. These references are stored in a data structure that keeps references to observers. *Subject* also has operations for adding and removing *Observers* objects to that data structure, operations *Attach* and *Detach*, respectively. *Subject* also has the *Notify* operation. This operation is responsible for the notification of the event to all interested *Observers*. It traverses the data structure and calls the *Update* operation of the *Observer* abstract class. This operation matches the *Observer*'s state to the *Subject*'s state.

The *ConcreteSubject* classes hold the state *Observers* are interested in and are responsible for notifying the *Observers* through the *Notify* operation when they change state. Finally, *ConcreteObserver* classes implement the *Update* operation, responsible for ensuring the consistency between the *Subject* and *Observer* state.

The advantages the *Observer* pattern brings are similar to the ones *Mediator* does, namely the loose coupling between *Subject* and *Observer*.

2.14 Prototype

"Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype".

The *Prototype* pattern offers a different approach to the creation of objects. It offers an alternative to inheritance sharing of common behavior. Instead of creating several subclasses to define a new behavior that share a common superclass, *Prototype* lets behavior be shared by creating new instances simply by copying a default instance of a subclass and then modifying it at will by saying how it differs from the default instance. This default instance is called the prototype instance.

This strategy is called delegation and has been previously discussed by Liebermann [35].

Figure 11 exemplifies the Prototype pattern's structure.

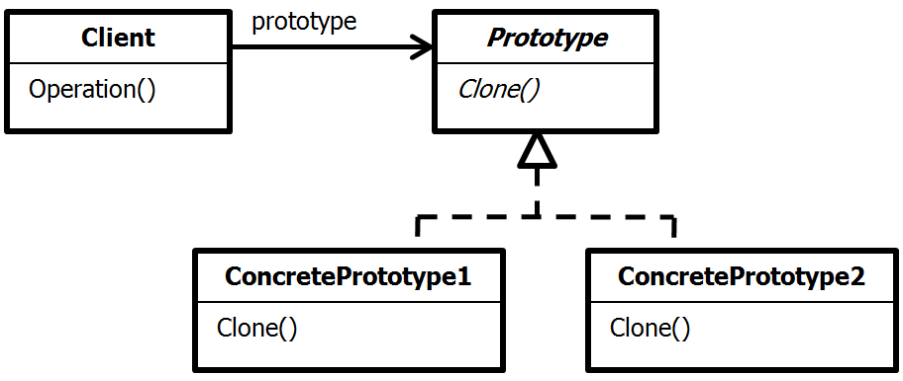

Figure 11 *Prototype* pattern structure

The *Prototype* pattern structure is relatively simple. The *Prototype* interface declares the *Clone* operation that lets objects produce copies of themselves. This operation is implemented in the *ConcretePrototype* classes. After the *Clone* operation is implemented, clients can create new *Prototype* instances by calling the *Clone* operation through their prototype reference.

The advantages of the *Prototype* pattern and its underlying delegation mechanism are associated with the advantages of object composition over inheritance, since delegation is a form of composition, in which the object responsible for the delegation passes itself to the other object.

2.15 Visitor

"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".

Instead of adding new functions to already existing classes in a given inheritance tree, the *Visitor* pattern adds them in a particular manner. It encapsulates related operations in specialized classes that add these operations. These classes can themselves create another inheritance tree for the implementation of operations to be added to classes of the original inheritance tree. A typical application of the *Visitor* pattern is when operations must be added to objects from the original inheritance tree that are stored in an object data structure. These operations are added dynamically to objects as objects from the specialized classes traverse the data structure. This is useful if the structure to be traversed has a considerable number of instances of a small number of classes and you want to perform some operation that involves all or most of them. This way, objects of the original inheritance tree must accommodate a new operation for accepting the new operations defined by the visiting object.

Figure 12 presents the *Visitor* pattern structure.

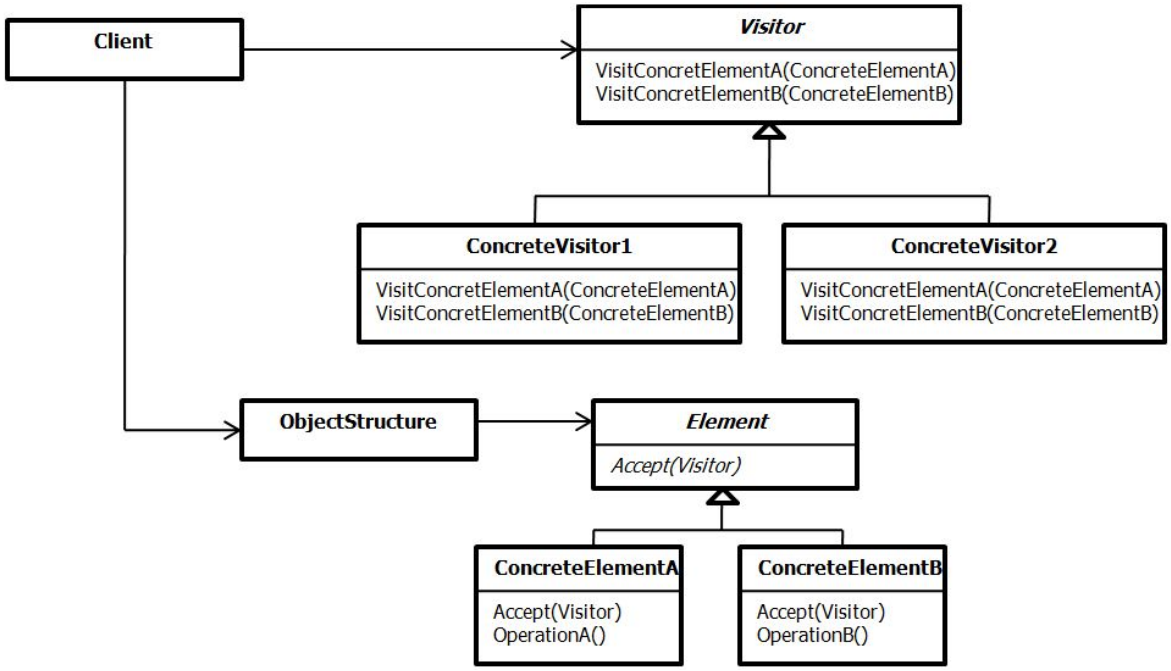

Figure 12 *Visitor* pattern structure

The *Element* and *ConcreteElement* classes define the base class hierarchy to which we want to add operations. These operations are added through the *Accept* operation. This operation takes as an argument the *Visitor* object that will add the operation to each object.

The *Visitor* class declares operations to visit each *ConcreteElement* class. The *ConcreteElement* class to be visited is determined by the operation's name and signature. *ConcreteElements* being visited use the appropriate operation to add operations to themselves in the definition of the *Accept* operation.

The *ConcreteVisitor* classes implement each operation declared by *Visitor*. Since the object structure has elements of different types, each *Visit* operation in a *ConcreteVisitor* define a part of the algorithm this particular *ConcreteVisitor* adds to the classes in the object structure. *ConcreteVisitor* also provides the context for the algorithm and stores its local state which can accumulate results during the traversal of the structure.

A language feature related to this pattern is double dispatching [15]. Most object oriented languages, like Java, support only single dispatch. This means that the exact implementation of method that gets executed depends only of the name of the method itself and a single additional factor, the type of the object that receives the method call.

This contrasts with multiple dispatching, as it is supported, for example, by the CLOS programming language. With multiple dispatching, the method that gets executed depends not just on the method's name and of the type of the object that carries out the method call. It also depends on the types of the various arguments of that method. Double dispatching is a special case of multiple dispatching because it deals with only one method argument.

The *Accept* operation is concerned with double dispatching because the exact visiting operation depends on the *Visitor's* and the *Element* it visits types.

The *Visitor* pattern's advantage is that it makes adding new operations easy. Furthermore, related operations can be gathered in a specific *ConcreteVisitor* classes. However, this pattern makes adding new *ConcreteElement* classes harder because every new *ConcreteElement* classes gives rise to an abstract method that must be implemented by every *ConcreteVisitor* class.

2.16 Summary

This chapter introduces the set of 11 GoF design patterns that are used for the CaesarJ design pattern implementations. Table 3 is an adaptation from Gamma et al. [21] and summarizes the design patterns that are approached in this study and the aspects that can vary within these design patterns.

Design pattern	Aspect(s) that can vary
Abstract Factory	families of related product objects
Bridge	implementation of an object
Builder	how a composite object gets created
Chain of Responsibility	object that can fulfill a request
Composite	structure and composition of an object
Decorator	responsibilities of an object without sub-classing
Factory Method	subclass of object that can be instantiated
Mediator	how and which objects interact with each other
Observer	number of objects that depend on another object; how the dependent object stays up to date
Prototype	class of object that is instantiated
Visitor	operations that can be applied to object(s) without changing their class(es)

Table 3 Design aspects that design patterns let you vary

3. CaesarJ

CaesarJ is one of several aspect-oriented programming languages. Like such, it strives to improve software engineering goals like modularity and reuse through constructs that support advanced separation of concerns. One of the main reasons why AOP solutions improve on OOP solutions in many cases is because OOP lacks proper language support to the separation of crosscutting concerns. This lack of support originates both code tangling and code scattering [32], which consequentially hinders modularity and reuse.

The rest of this chapter is organized as follows: section 3.1 introduces CaesarJ, makes a short comparison between CaesarJ and AspectJ and section 3.2 introduces its concepts of aspect implementation, its conceptual modules and language features. The following sections describe these concepts, kinds of modules and their related mechanisms in further detail. Section 3.3 presents the virtual class and family polymorphism mechanism; section 3.4 discusses the lack of support for the separation of concerns in traditional OOP languages by presenting an example of the *Observer* pattern through which some CaesarJ features are illustrated; sections 3.5 through 3.8 describe the four conceptual modules by which CaesarJ specifies an aspect component; section 3.9 ends the chapter by presenting an example of the impact that programming with CaesarJ has on client code, making a comparison with a Java implementation of the same example.

3.1 Introduction to CaesarJ

This chapter starts by drawing a comparison with AspectJ. Similarly to AspectJ, CaesarJ is also an AOP extension to the object-oriented programming (OOP) Java programming language. This way, any Java program (up to Java 2) can potentially benefit from CaesarJ functionalities. Being a more recent language than AspectJ, CaesarJ benefits from the

knowledge and experience gained from the first years of AspectJ and both have some common points, such as AspectJ's joinpoint, pointcut and advice model. Some exceptions to this case follow:

- The `if(…)` pointcut is not supported;
- Abstract pointcuts are not supported;
- A piece of advice can refer to the pointcuts in the declaring class or any of its superclasses;

Besides these exceptions, there are some more significant differences between them, as far as technical support, language constructs and conceptual models.

For years, AspectJ has been having more developed support and maintenance than CaesarJ. Not only does AspectJ seem to have a larger team providing technical support and maintenance, its support seems to be more complete, robust and sophisticated than CaesarJ. Also, there are by far many more scientific articles and documentation focusing on AspectJ than on CaesarJ.

These differences in technical support translate into the stability of the language conception. CaesarJ's constructs have evolved through different stages until the current version as can be testified from an overview of key publications describing the language [6][37][38][39][48]. Other than that, CaesarJ plug-ins for IDEs, namely Eclipse, are not as sophisticated and robust as AspectJ's. For instance, in the present plug-in version for the Eclipse IDE, the build automatically option is still unreliable since it may result in incomplete project builds.

As of January 2008, AspectJ became backwards compatible with Java 6, while CaesarJ remains backwards compatible with Java 2. Therefore, CaesarJ lacks the support for a number of features such as annotations and generic types. Furthermore, although CaesarJ is a newer language than AspectJ, it currently has less technical support [3] and its most recent version dates from April 2008 and is actually older than AspectJ's more recent version [2], as AspectJ's latest version was released in December 2008. Currently, there is no indication that CaesarJ will be compatible with newer versions of Java since there is no CaesarJ version scheduled to be released.

These differences are reflected in the widespread acceptance of AspectJ and its larger number of users. AspectJ has long since have a mailing list with heavy traffic by its users, whereas CaesarJ's mailing list has less traffic and is more recent.

3.2 Structure of a CaesarJ component

CaesarJ does not use the AspectJ construct *aspect*. Instead, CaesarJ presents the language construct *cclass***,** which defines a CaesarJ class. A **cclass** enhances a plain Java class by providing additional Caesar features. Among these features are the pointcut and advice mechanisms akin to AspectJ, but also virtual classes and family polymorphism (section 3.3) and mixin composition (section 3.8). Although plain Java classes can be composed with **cclasses**, these classes also present some limitations. **Cclasses** can implement Java interfaces but they cannot extend regular Java classes. Consequentially, Java classes cannot be casted into **cclasses**, and vice-versa. In addition, **cclass** arrays are not allowed, although traditional data structures from the Java API can be used.

Both languages differ in their reuse mechanisms. The primary technique for reuse in AspectJ is obtained through abstract aspects and concrete aspects which bind the abstract aspect to case-specific systems. Reuse comes from the fact that different concrete aspects can be made to inherit from a common abstract aspect.

CaesarJ has a different approach for dealing with aspects and crosscutting concerns. It recognizes that the joinpoint interception mechanism, although a cornerstone of AOP, is not sufficient to reflect the structural nature of aspects into modules with a rich inner structure [38]. Since crosscutting concerns, by definition, span over different concerns and involve different abstractions, the aspect structure should reflect this nature.

Conceptually, CaesarJ sees an aspect a component. To achieve better modularity, CaesarJ structures a component with different kinds of modules. These modules are called Collaboration Interfaces (CI) (section 3.5), CaesarJ Implementations (CJImpls) (section 3.6), CaesarJ Bindings (CJBindings) (section 3.7) and Weavelets (section 3.8).

The general structure of the component, with all the participant roles of the module and the actions they perform, is described in the CI. The CI is an abstract top level class, where the participant roles of the CI are declared, though not implemented, as inner, nested CaesarJ

classes. The implementation of these inner classes is made in the CJImpls and CJBindings. In CaesarJ, these inner classes are also virtual classes and have different properties from Java's inner classes. These virtual classes are used in CaesarJ to implement a mechanism called family polymorphism. Since all CaesarJ inner classes are virtual classes, the latter denomination will be used, to emphasize this important characteristic.

The CJImpl defines the context independent parts of a CI. There can be many different CJImpls to a single CI. In cases where different CJImpl modules exist, it is possible to switch a CI's implementation by a module with an alternative implementation without any impact or change on the code of the remaining modules.

The CJBinding defines the context specific facet of the CI. They are the "glue" that binds the CI to the concrete application to which the component is bound. CJBindings map the roles declared in the CI to the existing abstractions of the system where the aspect is to be inserted. CJBindings can use AspectJ-like pointcuts and advices but also define wrapper classes. These wrapper classes take a particular class performing a core-concern function in the system and enhance it with additional state and behavior related to the cross-cutting concern defined in the aspect component.

Since CJImpls and CJBindings describe different, non-overlapping facets of the CI, it is necessary to combine them in a single module that pairs both modules and realizes the whole aspect component. The module where this operation takes place is called a Weavelet. Weavelets are described in section 3.8.

Not all of these modules are necessarily a part of all possible CaesarJ components, except for the CJBindings, since they are essential to hold together the more concrete part of the application and the more abstract which is the CI.

CaesarJ achieves reuse by isolating different facets into separate modules that are allowed to evolve independently and easily combined to produce a complete aspect component. More concretely, the CI supports loose coupling between the code holding the abstract component implementation in the CJImpl, and the code in the CJBinding that attaches the component implementation to the application where the component is to be deployed.

Figure 13 illustrates the general structure of a CaesarJ component, describing all CaesarJ modules, their mutual relations and the relations to classes in an application.

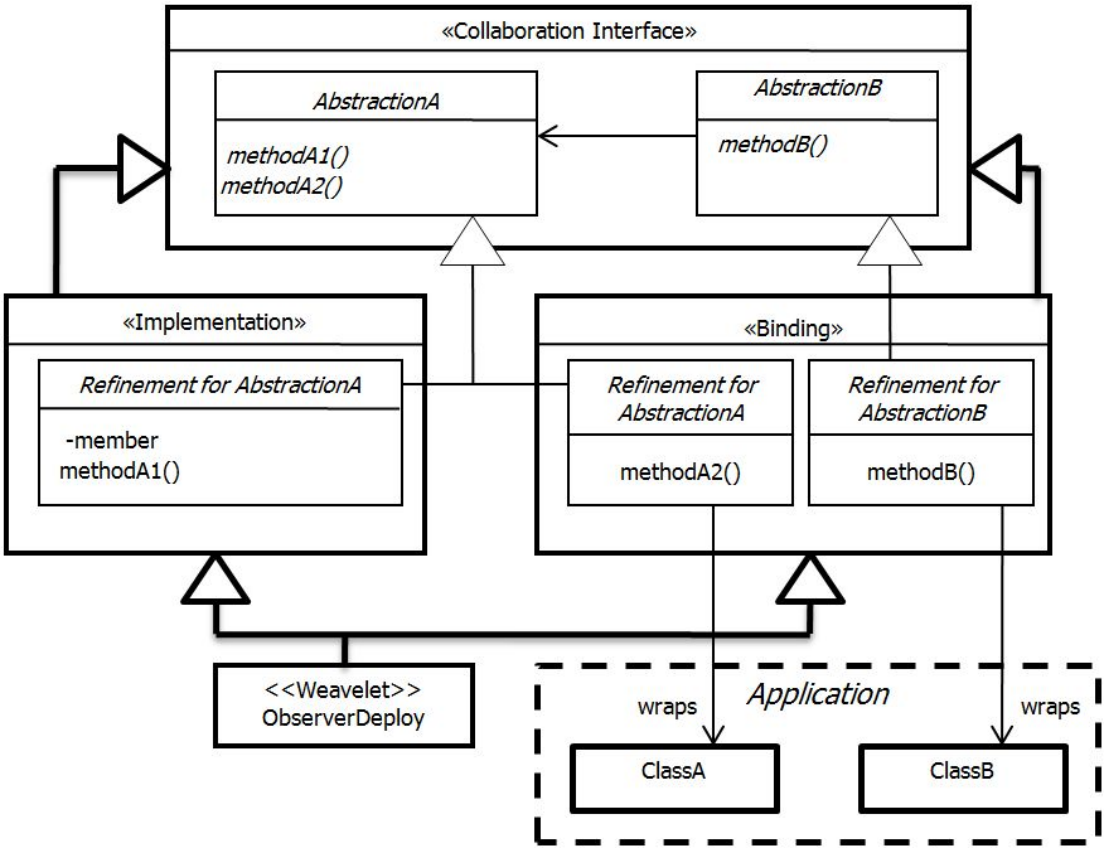

Figure 13 General structure of a CaesarJ component

3.3 Virtual classes

Virtual classes provide classes the ability to treat inner classes as class attributes, the same way as methods and fields [36]. The term *virtual classes* reflects the parallelism with *virtual methods* present in traditional object-oriented languages, since they follow similar rules as far as definition, overriding and reference [19]. This ability allows these inner classes to be polymorphically redefined by its subclasses. CaesarJ defines virtual classes as inner classes of an enclosing class, the *family class*. An instance of a family class is called a *family object*. Virtual classes must always be accessed through an instance of the enclosing class (the family object) where they are defined (the family class). As a consequence, the implementation of a virtual class is late bound, as it is dependent of the object used to access it. Therefore, the name of an inner class does not uniquely identify a specific virtual class [18]. Additionally, because the late binding of a class operates at the level of the class name, the name of a virtual class is not related to a single class.

3.3.1 Implicit inheritance

Family classes hold sets of collaborating inner (virtual) classes that must be accessed through their family object. Thanks to the virtual class mechanism, CaesarJ is able to define variations of abstractions represented as virtual classes by refining these virtual classes in family subclasses. The difference between CaesarJ's virtual class refinement and conventional sub-class refinement is that the references to a virtual class from other virtual classes are dynamically bound by the family object. Figure 14 illustrates this difference in CaesarJ's virtual class mechanism.

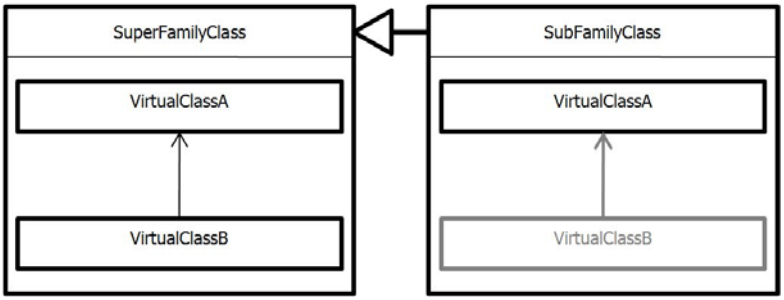

Figure 14 Virtual classes in CaesarJ

Figure 14 presents two family classes, `SuperFamilyClass` and its sub-class `SubFamilyClass`. Within `SuperFamilyClass` there are two virtual classes, `VirtualClassA` and `VirtualClassB`. `VirtualClassB` references `VirtualClassA`. Within `SubFamilyClass` there is only one virtual class, the `VirtualClassA` virtual class in `SubFamilyClass` which refines the `VirtualClassA` defined in `SuperFamilyClass`. The difference to conventional sub-class inheritance mechanisms is that all `VirtualClassB` references to `VirtualClassA` from an object of type `SubFamilyClass` are bound to the refined `VirtualClassA`. This effect is indicated by the gray shadows in `VirtualClassB`.

The refined `VirtualClassA` also reflects another mechanism provided by virtual classes, *implicit inheritance*. Because both `SuperFamilyClass` and `SubFamilyClass` define a virtual class `VirtualClassA`, and `SubFamilyClass` extends `SuperFamilyClass`, an implicit inheritance relation is created between the two family classes. The `VirtualClassA` in `SubFamilyClass` overrides the `VirtualClassA` in `SuperFamilyClass`. This makes the

instantiation of `VirtualClassA` late-bound, depending on the family object through which it was accessed.

3.3.2 Family polymorphism

CaesarJ uses the virtual class mechanism to implement family polymorphism [18]. Family polymorphism addresses the problem of expressing families of related classes and managing their relations polymorphically. Family polymorphism enables developers to flexibility redefine these classes while the type system guarantees that classes from different families are not mixed. Through family polymorphism it is possible to statically declare and manage relations between several classes polymorphically, in a way that a given set of classes is known to constitute a family and the relations between its members, without specifying statically exactly what those classes are.

Traditional object-oriented languages lack mechanisms to represent families of related classes explicitly. This limitation poses consistency issues, since objects from unrelated class families are allowed to mix. The issue of multi-object consistency is noticeable in parallel inheritance chains of collaborating classes with multiple possible combinations of class implementations [47]. However, not all combinations are consistent and the type system should be able to distinguish between the consistent and inconsistent variations. Since no explicit definition of related classes is present, mainstream object-oriented languages force developers to make a decision between flexibility and safety. As a consequence, either too many inconsistent combinations are allowed or correct combinations are blocked. To cope with this limitation, the *Abstract Factory* design pattern [21] is sometimes used.

The fact that the type system relies on the identity of the family object means that it must always be passed as an argument along with the instances of the family. In order for the type checker to guarantee that the family object is the same in all situations, some situations associated to the handling of the object references are not allowed. 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 part of the family [6]. CaesarJ's virtual class mechanism and its implementation of family polymorphism have been proven sound by Ernst el al. in [36].

3.4 Illustrating Example: *Observer*

The concept behind CaesarJ can be further illustrated by a concrete example of the situations it addresses and the mechanisms used in CaesarJ's aproach. To that effect, a Java scenario of the *Observer* pattern [21] is used. This example presents the problems related to the lack of separation of concerns in traditional OOP and to serve as comparison to CaesarJ's method to modularizing crosscutting concerns. This example has been taken from Bruce Eckel's book "Thinking in patterns" [17] which includes a repository of Java design patterns.

The *Observer* pattern is characterized by two roles: *Observer* and *Subject* (see section 2.13). The *Observer* and *Subject* roles are related by an observing relation, where multiple *Observers* subscribe to *Subjects* to monitor events of interest. When an event of interest takes places in a *Subject*, the *Subject* must notify its *Observers* and *Observers* update their internal state. In this scenario, the `Flower` class performs the *Subject* role, where the events of interest that may occur are the opening and closing of that flower's petals. The `Bee` and `Hummingbird` classes perform the role of *Observer* and are interested in the opening and closing events. When a `Flower` opens its petals, its *Observers* have breakfast; when a `Flower` closes its petals, its `Observers` go to sleep. Listing 1 presents the `Flower` class, where the shaded lines discriminate the code related to the *Observer* pattern, particularly the role of `Flower` as *Subject*. Listing 2 presents the `Bee` class, where the shaded lines discriminate the code related to the *Observer* pattern, particularly the role of `Bee` as *Observer*. The `Hummingbird` class is identical to the `Bee` class, therefore it is not presented.

This scenario takes advantage of the Java API classes, particularly the `Observable` class and the `Observer` interface to implement the roles of *Subject* and *Observer*, respectively. Classes that perform the *Subject* role must extend the `Observable` class to implement the logic related to the storage of interested *Observers* and to notify them of events of interest. The *Observer* role classes must implement the `Observer` interface and its `update` method to provide *Observers* the ability to refresh their state.

Instead of having the participant classes extending the `Observable` and `Observer` classes directly, inner classes are used to isolate the code related to the role played in the pattern.

```
01 class Flower {
02   private boolean isOpen;
03   private OpenNotifier oNotify = new OpenNotifier();
04   private CloseNotifier cNotify = new CloseNotifier();
05   public Flower() { isOpen = false; }
06   public void open() { // Opens its petals
07     isOpen = true;
08     oNotify.notifyObservers();
09     cNotify.open();
10   }
11   public void close() { // Closes its petals
12     isOpen = false;
13     cNotify.notifyObservers();
14     oNotify.close();
15   }
16   public Observable opening() { return oNotify; }
17   public Observable closing() { return cNotify; }
18   private class OpenNotifier extends Observable {
19     private boolean alreadyOpen = false;
20     public void notifyObservers() {
21       if(isOpen && !alreadyOpen) {
22         setChanged();
23         super.notifyObservers();
24         alreadyOpen = true;
25       }
26     }
27     public void close() { alreadyOpen = false; }
28   }
29   private class CloseNotifier extends Observable{
30     // Logic for the notifying closing events
31   }
32 }
```

Listing 1 Class Flower of the Java Flower Observer scenario.

```
01 class Bee {
02   private String name;
03   private OpenObserver openObsrv = new OpenObserver();
04   private CloseObserver closeObsrv = new CloseObserver();
05   public Bee(String nm)  { name = nm; }
06   // An inner class for observing openings:
07   private class OpenObserver implements Observer{
08     public void update(Observable ob, Object a) {
09       System.out.println("Bee " + name + "'s breakfast time!");
10     }
11   }
12 // Another inner class for closings:
13   private class CloseObserver implements Observer{
14     public void update(Observable ob, Object a) {
15       System.out.println("Bee " + name + "'s bed time!");
16     }
17   }
18   public Observer openObserver() {
19     return openObsrv;
20   }
21   public Observer closeObserver() {
22     return closeObsrv;
23   }
24 }
```

Listing 2 Class Bee of the Java Flower Observer scenario.

An inner class is defined for every type of event of interest that might occur. The use of inner classes is justified by their ability to refer members of the enclosing class, including private members [47]. By using inner classes rather than extending `Observable` and `Observer` directly, the enclosing classes are free to extend classes other than the ones related to the pattern code. This characteristic of inner classes provides a limited form of multiple inheritance.

The main limitation to this approach is that classes are concerned with more than one concern, as can be seen from the shaded lines. In fact, the `Flower` class does not represent solely a flower, it represents a flower that must notify its observers when it opens or closes it petals. The inverse situation is true for `Bee`. This leads to code tangling because code for the pattern logic is mixed if the code for the core class concern. Code scattering is also present because the pattern related code is not modularized but rather spread throughout the class [47].

CaesarJ presents a different solution to these problems. This solution and the modules that enable it are present in Sections 3.5 through 3.9.

3.5 Collaboration Interfaces

A CaesarJ component has different parts that interact with each other. A Collaboration Interface (CI) contains the declarations of the abstract roles of those participants as well as their operations. These characterize how the participants interact (or collaborate) with each other. A CI describes the roles of the objects that comprise any implementation of the component and the relations between them. These roles are defined by collaborating classes represented by virtual classes and are frequently mutually recursive in the sense that each virtual class refers to the other virtual class in its own definition. Also, each role represents an abstraction of the modular structure of the aspect [38].

Furthermore, a CI has two facets, the *provided* and *expected* facets of the component. The provided facet tells what the component provides to the context to which it is applied and the expected facet tells what the component expects from the context it is applied in, so it can deliver what the provided facet promised. Initially, the provided and expected facets were made explicit by the keywords `required` and `expected`. Nowadays, this is no longer true, as

the CaesarJ developers came to the conclusion that each concrete use of provided/expected would represent only one among many possible variants [6].

The CJImpl of a component comprises the implementation of the provided part while a CJBinding integrates the component with the base application to implement the expected part. Although CJImpls and CJBindings are placed in different modules, they are connected through their common CI which serves as a medium for bidirectional communication between them since both inherit from the same CI [38]. This way, CI's support loose coupling between implementations and bindings, contributing to more reuse opportunities, since different CJImpls and CJBindings of a common CI can be combined to produce different implementations of the component. The CI itself only contains design information, but acts as an interface for the implementation of the aspect component in the CJImpls and CJBindings.

Listing 3 presents an example of a CI related to the Flower Observer scenario of the *Observer* pattern. This example was previously developed by Sousa et al. [47][48].

```
01 public abstract cclass ObserverProtocol {
02        public abstract cclass Subject {
03               public abstract void addObserver(Observer obs);
04               public abstract void removeObserver(Observer obs);
05               public abstract void removeObserver();
06               public abstract void notifyObservers();
07               public abstract Object getState();
08        }
09
10        public abstract cclass Observer {
11               public abstract void refresh(Subject s);
12        }
13 }
```

Listing 3 Collaboration Interface of the Observer pattern in the Flower Observer scenario

The abstract **cclass** `ObserverProtocol` describes the collaboration between two other abstract virtual **cclasses,** `Subject` and `Observer`. These **cclasses** are mutually recursive, since they have references to each other on their own definition. In this example, there are two different levels of abstraction in the methods in `Subject` and `Observer`. Some methods have a closer relation to application details while others have a more abstract nature.

The methods `addObserver`, the two `removeObserver` methods and `notifyObservers` in `Subject` are abstract, since the way we add, remove or notify an *Observer* is not necessarily context sensitive. It is up to the programmers to define how these actions should

be performed and they are not bound to a specific application. Therefore, these methods should be implemented in a CJImpl. The methods `getState` in `Subject` and `refresh` in `Observer` belong to a concrete, specific case since they are directly dependent on the application they are applied to. This way, these methods should be implemented in a CJBinding.

Notice that although two roles are described, a collaboration typically involves several instances of multiple types. Furthermore, virtual classes allow for an arbitrary number of roles to be defined within a CI. These virtual classes can also define inheritance hierarchies between them.

Figure 15 illustrates the structure of the CI developed for the Flower Observer scenario.

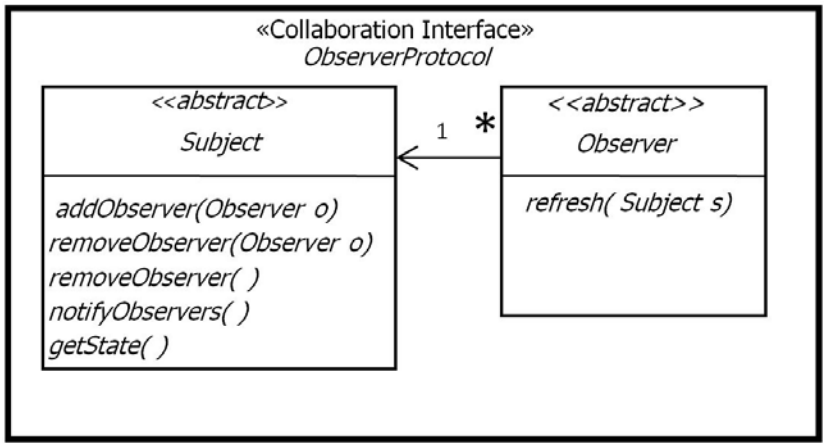

Figure 15 Collaboration Interface for the Flower Observer scenario

3.6 CaesarJ Implementations

CaesarJ Implementations (CJImpls) implement the context sensitive methods inherited from the CI. They correspond to AspectJ's abstract aspects since they are independent from any particular case. Because they are not application specific, CJImpls are reusable by nature. Application specific methods should be described in the CJBinding.

In order to implement the provided facet of a CI, it is necessary to create virtual classes within a family class that have the same name as in the CI, or nested classes that explicitly extend the nested class declared on the CI. This is a consequence of virtual classes. In Java, declaring a nested class with the same name as an inner class of the super class is a

phenomenom named *shadowing*. However, this is not the case with CaesarJ, since virtual classes are late bound and do not uniquely identify an inner class. Also, methods from the expected facet declared in the CI can be called in order to implement the provided facet.

Finally, additional state and behavior can be added to virtual classes in the implementation of the CJImpl. The virtual classes in CJImpls can extend and refine the virtual classes defined in their CIs by adding data members or methods necessary to the implementation of the provided facet of the CI. The reusable nature of CJImpls comes from the fact that they are not bound to a concrete scenario, so they have some level of abstraction from the concrete system and because it is possible to define CJImpls with different data members and method implementations, according to the developer's specific needs, depending on its goals and the systems characteristics.

Listing 4 shows a CJImpl developed for the Flower Observer scenario following that CI.

```
01 public abstract cclass ObsImpl extends ObserverProtocol{
02        public cclass Subject {
03               private ArrayList observers = new ArrayList();
04
05               public void addObserver(Observer obs){
06                      this.observers.add(obs);
07               }
08               public void removeObserver(Observer obs){
09                      this.observers.remove(obs);
10               }
11               public void removeObserver(){
12                      this.observers.clear();
13               }
14               public void notifyObservers(){
15                      Iterator it = this.observers.iterator();
16                      while(it.hasNext())
17                             ((Observer)it.next()).refresh(this);
18               }
19               public Object getState(){
20                      return null;
21               }
22        }
23 }
```

Listing 4 CaesarJ Implementation of the Observer pattern in the Flower Observer scenario

Listing 4 presents the implementation of the non context specific methods of the `ObserverProtocol` CI. As we can see, `ObsImpl` declares itself to be both abstract and extending `ObserverProtocol`. This illustrates the inheritance mechanism in CaesarJ and also the rationale behind CJImpls. `ObsImpl` is abstract because only the non-application sensitive methods of the virtual classes in `ObserverProtocol` are implemented. As far as the inheritance mechanism, `ObsImpl` contains a virtual class `Subject` that refines the abstract

virtual class `Subject` in `ObserverProtocol`. This virtual class adds an `ArrayList` member `observers` to `ObserverProtocol.Subject` and implements its `addObserver`, `removeObserver` and `notifyObservers` methods based on this data structure. To illustrate the possibility of reuse CJImpls offer, an alternative implementation could be performed where the `observers` member would have a different data structure, such as a `WeakHashMap`. This time, the operations regarding the set of observers would relate to a `WeakHashMap` instead of an `ArrayList`, but that change would only have an impact in the implementation of the CJImpl module. Reuse comes from the fact that these different implementations could be easily switched without having to change any from the remaining in the rest of the CaesarJ component.

The capability for CJImpls to call on methods from the expected facet of the component is exemplified by the `notifyObservers` method. It calls the `refresh` method which is declared in the `ObserverProtocol.Observer` virtual class.

Figure 16 represents the structure of the CJimpl developed for the Flower Observer scenario.

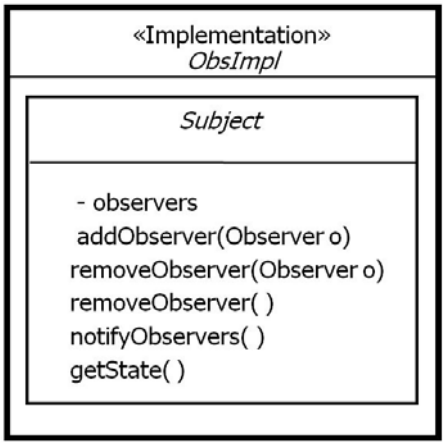

Figure 16 CaesarJ Implementation for the Flower Observer scenario

3.7 CaesarJ Bindings and Wrappers

CaesarJ Bindings (CJBindings) are the glue that binds the component to a specific application [39]. Application specific methods should be described in the component binding. The CJBinding is the module that complements the CJImpl. While the CJImpl

implements the abstract part of the CI, the CJBinding implements the methods that enclose the context specific logic of the component and are defined by the classes in the base application. CJBindings map the abstract roles in the component to classes in the application context through *wrapper classes* [37].

3.7.1 Wrapper classes

Wrapper classes are expressed by the *wraps* keyword. A wrapper class can map one or several application objects to a role defined in the CI. The wrapped objects of the application domain can be accessed by the *wrappee* keyword. Family classes cannot define wrapping relations, only virtual classes. The expression `cclass A extends B wraps X` is syntactic sugar for:

```
cclass A extends B {
      X wrappee;
      A(X wrappee) {this.wrappee = wrappee};
}
```

Listing 5 Wrapper declaration syntactic sugar

Like such, virtual classes that define wrappers cannot be declared abstract. This restriction poses limitations to the integration of multiple hierarchical structures into the component.

Wrappers add behavior and state to the **wrappee** and can use the **wrappee**'s interface to implement the expected facet methods defined in the CI. Wrapper objects are available until the **wrappee** is swept by the garbage collector. Wrappers are CaesarJ's mechanism that replaces AspectJ's inter-type declarations and the **declare parents** clause.

3.7.2 Wrapper recycling

Since wrappers establish the correspondence between objects in the application domain and instances of the roles defined in the component, it is necessary to ensure that each component role instance is associated with a single object in the application domain. Such consistency is necessary to assure the correct navigation between the abstractions in the component context and the abstractions in the base application [37]. The *wrapper recycling mechanism* ensures such consistency. Wrappers are not instantiated with the `new` constructor

call. Instead, wrappers are created by a wrapper constructor call. The difference between wrapper constructor calls and regular constructor calls is that a wrapper constructor call only returns a new instance if a wrapper for that object does not already exist. If so, the wrapper constructor call returns the existing wrapper. This way, the identity and state of the wrapper is preserved [37].

A wrapper constructor call is identified by a `outerClassInstance.W(wrappee)` signature, where `outerClassInstance` is the family object, `W` is the virtual class that defines the wrapper class, and `wrappee` is the application object to be wrapped. CaesarJ implements the wrapper recycling mechanism by keeping a `WeakHashMap` data structure in the family object that keeps the correspondence between application objects and wrapper instances. When a wrapper constructor call is performed, a key is created for the constructor argument and looked up in the `WeakHashMap` data structure. If the lookup fails, an instance of `outerClassInstance.W` is created for the `wrappee` object, stored in the `WeakHashMap` and returned. If the lookup does not fail, the already existing instance stored in the `WeakHashMap` is returned.

In cases where it is necessary, bindings can also contain AspecJ-like pointcuts and advices to collect data from the context of the running application.

The same way CJImpls correspond to AspectJ's abstract aspects, CJBindings correspond to AspectJ's concrete sub-aspects.

Listing 6 shows a CJBinding developed for the Flower Observer scenario following that CI. In Listing 6 we have an example of how a CJBinding glues the whole component to its specific application.

Like `ObsImpl`, `ObsBinding` is also abstract. It is the complementary part to `ObsImpl` as far as implementing the CI `ObserverProtocol` and, like `ObsImpl`, does not implement every method in `ObserverProtocol` so, it must be declared abstract.

The way `ObsImpl` binds itself to the application classes is through the **wraps** clause. There are six wrapper classes in this CJBinding, `FlowerOpening`, `FlowerClosing`, `BeeIsOpenObserver`, `BeeIsCloseObserver`, `HummingbirdIsOpenObserver` and `HummingbirdIsCloseObserver`, which map the virtual classes in the CI to the concrete classes of the application.

```
01 public abstract cclass ObsBinding extends ObserverProtocol{
02         public cclass FlowerOpening extends Subject wraps Flower {}
03         public cclass FlowerClosing extends Subject wraps Flower {}
04
05         public cclass BeeIsOpenObserver extends Observer wraps Bee {
06                 public void refresh(Subject s) { wrappee.dinner(); }
07         }
08
09         public cclass BeeIsCloseObserver extends Observer wraps Bee {
10                 public void refresh(Subject s) { wrappee.rest(); }
11         }
12
13         public cclass HummingbirdIsOpenObserver extends Observer wraps Hummingbird {
14                 public void refresh(Subject s) { wrappee.dinner(); }
15         }
16
17         public cclass HummingbirdIsCloseObserver extends Observer wraps Hummingbird {
18                 public void refresh(Subject s) { wrappee.rest(); }
19         }
20
21         pointcut openCloseEvents(Flower f) : (set(* Flower.isOpen)) && this(f);
22         void around(Flower f, boolean new_val) : openCloseEvents(f) && args(new_val) {
23                 boolean old_val = f.isOpen();
24                 proceed(f,new_val);
25                 if(old_val != new_val)
26                         if(new_val)
27                                 FlowerOpening(f).notifyObservers();
28                         else
29                                 FlowerClosing(f).notifyObservers();
30         }
31 }
```

Listing 6 CaesarJ Binding of the Observer pattern for the Flower Observer scenario

The first two are virtual classes that extend the *Subject* role of the CI, so they define events to be observed on their wrapped class, in this case, class `Flower`. Thus, `FlowerOpening` observes the opening of a flower and `FlowerClosing` its closing. These wrappers don't add any behaviour to their wrapped classes**.**

The last four wrapper classes extend the role of the *Observer* and define its behaviour relating it to the base application by wrapping the `Bee` and `Hummingbird` classes**.** These classes implement the method `update`, considering all cases possible between an event of the flower opening and closing and if the *Observer* is a `Bee` or a `Hummingbird`. In each case, the `update` method is mapped to the wrappee's `dinner` or `rest`, depending whether the `Flower` opened or closed.

`ObsBinding` makes use of an AspectJ-like pointcut to capture points of interest in the program execution. The pointcut captures information about the context of the control flow of the application and further connect the CJBinding with the concrete instance of the design pattern. In this case, a named pointcut is used to capture the event of a modification of the `isOpen` member in `Flower`. Finally, the advice associated to this pointcut implements the

update logic of the *Observer* according to the situation. If there is a change in the state of the `isOpen` member, the advice calls the `notifyObservers` method on the `FlowerOpening` or `FlowerClosing` wrapper class, depending on the event.

Together with the CJImpl, the CJBinding allows the plain Java classes performing the roles of *Subject* and *Observer* in this scenario to be redefined, removing the crosscutting concerns of the pattern from the base logic of the classes `Flower`, `Bee` and `Hummingbird`. This way, the inner classes declared to perform the actions pertaining to the interactions between the roles need not appear. Listing 7, Listing 8 and Listing 9 show classes `Flower`, `Bee` and `Hummingbird` without any `Observable` or `Observer` members. The pattern code has completely disappeared from these classes into the modules of the aspect component.

```
01 public class Flower {
02         private boolean isOpen;
03         public boolean isOpen(){return this.isOpen;}
04         public Flower(){
05                 this.isOpen=false;
06         }
07         public void open(){
08                 this.isOpen=true;
09         }
10         public void close(){
11                 this.isOpen=false;
12         }
13 }
```

Listing 7 Flower class without Observable inner classes

```
01 public class Bee {
02         private String name;
03         public Bee(String name){
04                 this.name = name;
05         }
06         public void dinner(){
07                 System.out.println("Bee " + name + "'s dinner time!");
08         }
09         public void rest(){
10                 System.out.println("Bee " + name + "'s bed time!");
11         }
12 }
```

Listing 8 Bee class without Observer inner classes

```
01 public class Hummingbird {
02         private String name;
03         public Hummingbird(String name){
04                 this.name = name;
05         }
06         public void dinner(){
07                 System.out.println("Hummingbird " + name + "'s dinner time!");
08         }
09         public void rest(){
10                 System.out.println("Hummingbird " + name + "'s bed time!");
11         }
12 }
```

Listing 9 Hummingbird class without Observer inner classes

Figure 17 shows the structure of the CJBinding developed for the Flower Observer scenario, and it's wrapping relations to the classes in the base application.

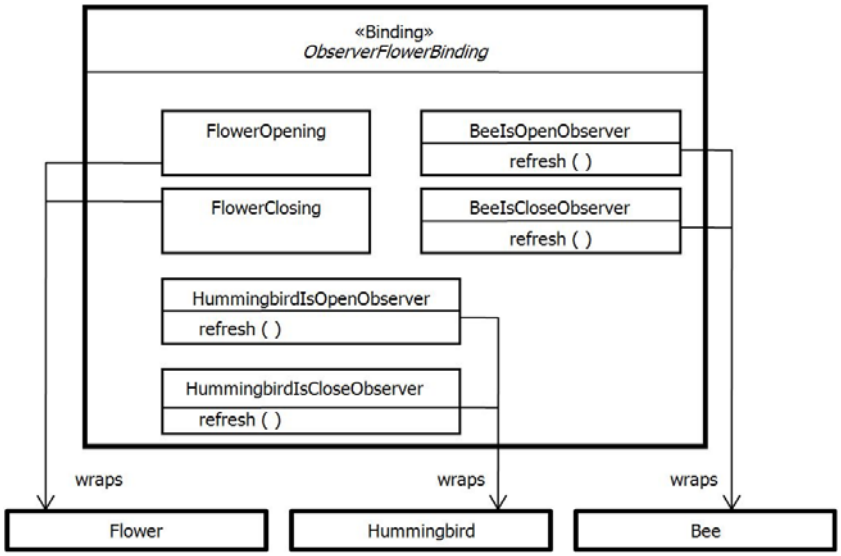

Figure 17 CaesarJ Binding for the Flower Observer scenario

3.8 Weavelets and Aspect Deployment

Different modules are used do define the complementary provided and expected facets of a CI. As both are incomplete parts of the same CI, they cannot be instantiated. A Weavelet takes the provided and the expected facets of a CI, and combines them to complete the definition of the CI.

3.8.1 Weavelets

A Weavelet is a **cclass** that takes a number of CJImpls and CJBindings and joins them together, creating the complete component. This procedure is called mixin composition [10]. Mixins are abstract subclasses that may be used to specialize the behavior of parent classes [10]. In CaesarJ, mixin composition takes abstract CJImpls and CJBindings to implement the operations in the CI. Mixins can be seen as a form of multiple inheritance, since it combines several different modules that implement complementary parts of a component. Since CaesarJ classes are mixins, mixin composition can be obtained by passing a mixin as the superclass parameter to another CaesarJ class.

Mixin composition is achieved in CaesarJ with the **&** operator and is characterized by the following sintax:

```
public cclass C extends A & B{ }
```

The **&** operator defines an inheritance chain between the mixin operands represented by classes `A` and `B`. The order of the operands defined in the mixin composition defines the order of the linear inheritance chain there the **&** operator is not commutative. The operand on the left hand side is more specific than the one on the right hand side [6]. Since CaesarJ classes can hold virtual classes, the mixin composition mechanism is propagated to the virtual classes where the linearization of the enclosing family class determines the linearization of the virtual classes.

3.8.2 Aspect instantiation and deployment

Unlike AspectJ, CaesarJ aspects can be explicitly instantiated. This corresponds to the instantiation of several Weavelets. This way it is possible to create several aspect instances in the same application and manage them as objects with special responsibilities. This provides the developer with enhanced control since it allows multiple instances of an aspect type with independent state, life-cycle, and scope of deployment.

After a Weavelet is defined and both provided and expected facets are composed, the Weavelet must still be deployed in order to activate its pointcuts and advices. In CaesarJ, aspect deployment can be made both statically and dynamically. This constitutes another difference from AspectJ, since AspectJ only supports static aspect deployment. Likewise, AspectJ cannot distinguish between multiple instances of the same class. Distinctions of these instances must be expressed programmatically.

In CaesarJ, static aspect deployment is obtained through the **deployed** modifier on the declaration of a **cclass**. This way, an aspect is deployed in compile-time. The **deployed** modifier can also be used in the instantiation of a **final static** object. Static deployment automatically deploys an aspect at load time.

Aspects can also be deployed dynamically. Dynamic deployment can be either *local deployment* or *thread-based deployment*. To carry out, an aspect must first be instanced by the instantiation of a Weavelet and then use the **deploy** statement do define the scope of that aspect. In local deployment, the activation scope of an aspect is defined by de **deploy** and

undeploy keywords, which activate and deactivate the aspect, respectively. The activation scope in threah-based deployment is defined by a **deploy** block. An aspect is deployed on the scope of the control flow inside the block and does not have any influence in concurrent executions.

Listing 10 shows a Weavelet developed for the Flower Observer scenario following that CI that completes the component by the composition of the corresponding CJImpl and CJBinding.

```
01 public cclass FlowerObserverDeploy extends ObsImpl  & ObsBinding{
02 }
```

Listing 10 CaesarJ Weavelet of the Observer pattern for the Flower Observer scenario

The Weavelet shown in Listing 10 simply connects the parallel hierarchies of the CJImpls and CJBindings that extend the CI. Mixin composition is the mechanism that makes this connection of different modules possible, by making `FlowerObserverDeploy` inherit both from `ObsImpl` and `ObsBinding`.

Since `FlowerObserverDeploy` is not abstract it can be instantiated in order to deploy the aspect component.

An example of local deployment is shown in Listing 11.

```
01 //instantiation of FlowerObserverDeploy
02 FlowerObserverDeploy asp = new FlowerObserverDeploy ();
03 …
04 deploy asp;
05 …
06 //the pointcut is active here
07 …
08 undeploy asp;
09 //the pointcut is no longer active
```

Listing 11 Instantiation, deployment and undeployment of a Weavelet

The constructor `new FlowerObserverDeploy()` creates an object of the same class that comprises the CJImpl and the CJBinding part of the component. However, the pointcuts in the binding are not yet active and need to be deployed. That happens only with the statement `deploy asp`, when the pointcuts become active. The statement `undeploy asp` does the opposite and makes the pointcuts become inactive.

Figure 18 illustrates the CaesarJ class diagram for the Flower Observer scenario.

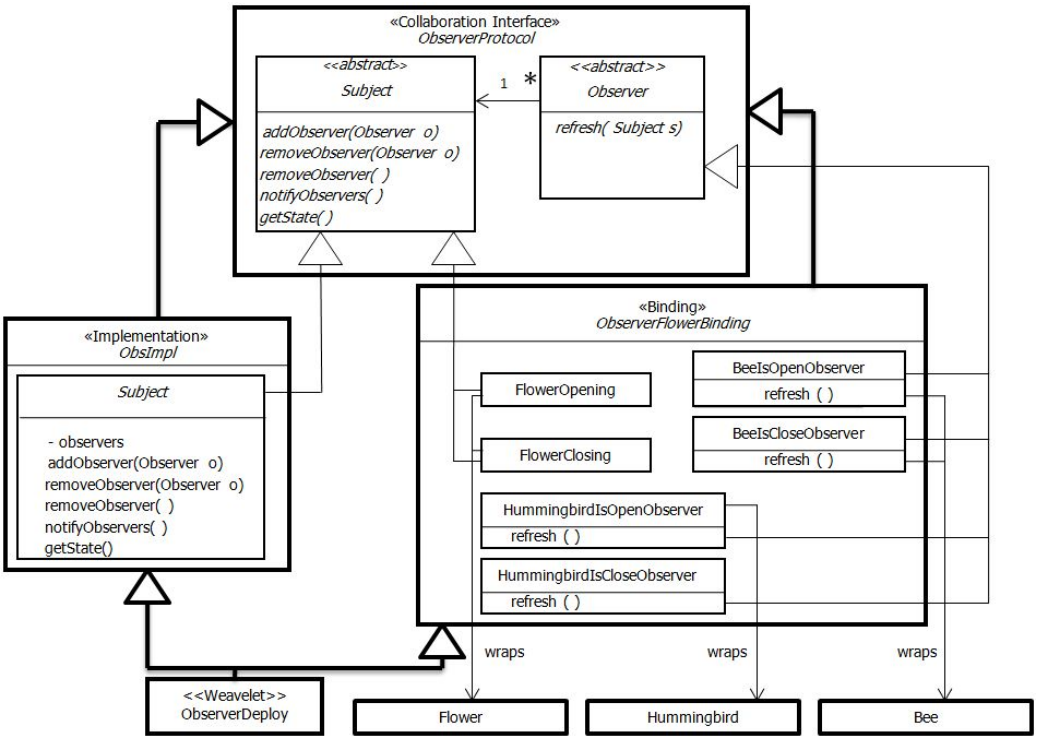

Figure 18 CaesarJ class diagram for the Flower Observer scenario

3.9 CaesarJ impact on client code

To further assess how a CaesarJ component has an impact on client code, an example is provided, once again using the Flower Observer scenario.

```
01 public static void main(String[] args) {
02                //aspect instantiation
03                final FlowerObserverDeploy asp = new FlowerObserverDeploy();
04                final FlowerObserverDeploy asp2 = new FlowerObserverDeploy();
05                //participant object instantiation
06                Flower f = new Flower();
07                Bee b1 = new Bee("Bee");
08                Hummingbird h1 = new Hummingbird("Hummingbird");
09                //Observer wrapper instantiation
10                asp.Observer b1_open = asp.BeeIsOpenObserver(b1)
11                asp.Observer b1_close = asp.BeeIsCloseObserver(b1)
12
13                asp2.Observer h1_open = asp2.HummingbirdIsOpenObserver(h1)
14                asp2.Observer h1_close = asp2.HummingbirdIsCloseObserver(h1)
15
16                //Subject wrapper instantiation
17                asp.Subject opening = asp.FlowerOpening(f)
18                asp.Subject closing = asp.FlowerClosing(f)
19                asp2.Subject opening2 = asp2.FlowerOpening(f)
20                asp2.Subject closing2 = asp2.FlowerClosing(f)
21                ...
```

Listing 12 Aspect, participant class and wrapper instantiations

Listing 12 provide code examples for the instantiation of aspects, the instantiation of objects that perform the participant roles of the pattern and the wrappers that map these Java objects to the context of the aspect component.

Lines 1-2 present the aspect instantiation mechanism in CaesarJ. Two aspect instances, `asp` and `asp2` are created in the same way plain Java objects are created. However, the `asp` and `asp2` objects are two different family objects from the same family class, thereby establishing two different families that are not allowed to mix, thanks to family polymorphism.

Lines 6-8 create the Java objects that will perform the roles of *Subject* and *Observer* in the pattern. The *Subject* will be performed by the `Flower` object `f`, and the *Observer* will be performed by the objects of classes `Bee` and `Hummingbird`, respectively `b1` and `h1`. The correspondence between objects in the application domain and roles in the aspect component must be performed by the creation of wrappers.

Lines 10-20 establish this correspondence. Lines 10-11 create wrappers that define that the `Bee` object `b1` will perform the role of *Observer* in the context of the `asp` family object. Since the same Java object `b1` must observe both opening and closing events, different wrappers map this object to the observed events. Lines 13-14 do the same for the `Hummingbird` object `h1`, but in the context of the family object `asp2`. The lines 17-20 map the `Flower` object `f` to the *Subject* role in both family objects `asp` and `asp2`. Since two events (flower opening and closing) are to be observed, two wrappers map these events for each of the family classes.

Next, it is necessary to define the relations between the participants in the pattern and deploy the aspects so they can activate their pointcut and therefore capture events of interest. Listing 13 gives some possible examples for these operations, taking the opportunity to demonstrate an application of the family polymorphism mechanism.

Lines 2-5 perform the relations between the *Subject* and *Observer* objects. *Observer* objects are added to *Subject* objects by the `addObserver` method. An effect of family polymorphism is that *Observer* objects can only be added to *Subjects* of the same family object. Lines 8-9 illustrate compiler errors that are detected as a consequence of trying to mix objects of different families. Since the compiler effectively sees different family objects as

repositories for virtual classes of different types, the attempt to mix objects from different family objects `asp` and `asp2` raises a type error.

```
01                //Definition of the relations between participants
02                opening.addObserver(b1_open);
03                opening2.addObserver(h1_open);
04                closing.addObserver(b1_close);
05                closing2.addObserver(h1_close);
06
07                //Family polymorphism examples
08                opening.addObserver(h1_open); //compiler error
09                closing2.addObserver(b1_close); //compiler error
10
11                //Local aspect deployment
12                deploy asp;
13                f.open();
14                undeploy asp;
15                //Thread-based aspect deployment
16                deploy (asp2){
17                f.close();
18                }
19 }
```

Listing 13 Participant relations definition, family polymorphism and aspect deployment

Finally, lines 12-18 demonstrate the dynamic aspect deployment mechanism. Lines 12-14 show the local deployment of `asp`. Line 13 deploys the aspect. Line 14 presents the event of a flower opening. Since the `asp` aspect is deployed, but `asp2` is not, only the advice triggered by the pointcuts in `asp` is performed. Line 15 defines the end of the deployment scope, undeploying the `asp` aspect. Lines 15-18 illustrate the thread-based deployment of aspect `asp2`. The same logic applies, except the deployed aspect is the `asp2` object and line 17 presents the event of a flower closing.

4. Background to the study

This dissertation is closely related to previous work regarding the study of AOP languages through the implementation of the GoF design patterns. The studies presented in this chapter constitute a framework for this dissertation, as they provide the basis for the motivation of this dissertation, the way to address this issue and provided previous examples of CaesarJ design pattern implementations. This chapter starts by describing the first study to address the issue of implementing the GoF design patterns using AOP languages, namely AspectJ. It goes on to present a study that pointed some downsides of the AspectJ implementations and finalizes with the studies that have previously explored the CaesarJ implementations of the GoF design patterns, lending a precious contribution to the making of this thesis.

Hannemann and Kiczales first tackled the issue of design pattern implementation using AOP languages [28]. The authors created Java and AspectJ implementations of the 23 GoF design patterns that served as the basis for comparisons between the object-oriented and aspect-oriented implementations. The AspectJ implementations were subjected to an evaluation according to the modularity criteria of:

- *Locality* – All the code implementing a pattern is placed in an aspect and removed from the participating classes. As a consequence, classes in the application domain are free of pattern code, hence there is no coupling between participants. Pattern implementation changes are confined to the aspect.
- *Reusability* – The pattern code is abstracted into a reusable aspect that generalizes the overall pattern behavior. The aspect can be reused in several instances of the pattern.
- *Composition transparency* – Multiple instances of the same pattern in one application are not confused. The same participant object or class can assume different roles in different instances of the same pattern.

- *(Un)pluggability* – It is possible to switch between using a pattern instance in a system or not. Therefore, participant classes must have a meaning outside the pattern implementation.

The pattern implementations are also characterized according to the nature of the roles involved in the pattern, where roles could be labeled as:

- *Defining* – The participants have no functionality outside the pattern. The roles define the participants completely.
- *Superimposed* – Roles are assigned to classes that have functionality outside the pattern. Roles are an augmentation of the existing classes.

The authors conclude that AspectJ design pattern implementation shows variable degrees of modularity improvement over Java implementations in 17 cases in terms of the criteria used. Out of these 17 patterns, 12 have resulted in reusable aspects. These improvements come from modularizing the implementation of the pattern into a separate unit. The reasons for these improvements are directly related to the crosscutting structure present in these design patterns, particularly in roles that have superimposed behavior.

The present study differs from the study of Hannemann et al. in that it draws a comparison between *two AOP languages*, using a narrower collection of patterns. Unlike Hannemann et al. the patterns covered were not developed completely anew, but rather collected from independent sources, which ensures greater independence of the results. The same qualitative criteria are applied to analyze the pattern implementation, but the present study is also concerned with CaesarJ's composition properties.

The AspectJ implementations of the GoF design patterns by Hannemann et al. are regarded as standards for good AspectJ design and programming. Nevertheless, these implementations have also exposed some problems in the way AspectJ deals with the separation of crosscutting concerns in the patterns as well as some shortcomings in AspectJ's structure as far as the support for reusable aspects. Monteiro et al. [41] have analyzed these implementations and found limitations on the *Command, Composite, Decorator and Memento* patterns, as well as the set of patterns identified by Hannemann et al. as the *multiple inheritance* patterns, i.e. *Abstract Factory, Bridge, Builder, Factory Method and Template Method*. The authors point out that, although aspects bring improvements in most cases, the adaptation of the Hannemann et al. reusable aspects to independent AspectJ

implementations of the GoF patterns sometimes resulted in awkward and inflexible interfaces that did not lighten the burden of client programmers, thereby defeating the purpose of reusability.

Sousa et al. first explored the implementation of the GoF design patterns with CaesarJ [48][47]. These studies have resulted in the first GoF design patterns implemented in CaesarJ. This study took the independent repositories of Java implementations of the GoF patterns listed in Chapter 1 and developed implementations for 7 patterns.

Table 4 describes the patterns developed by Sousa et al.

	Thinking in patterns	**DP Java companion**	**Fluffycat**	**Hannemann et al.**	**Huston**	**Guidi Polanco**
Abstract Factory		X	X			
Bridge			X		X	
Chain of Responsibility			X	X		
Decorator				XX		
Observer	XX					
Singleton				X		
Visitor	X		X			

Table 4 Previously developed design patterns

To better assess CaesarJ's opportunities for reuse, several implementations for each pattern were created. This approach allows a better understanding of CaesarJ's characteristics because the same design issue is reflected in different manners in different repositories. This is reflected by the rows with several marks.

The authors have also registered the mechanisms used by CaesarJ in each pattern. The mechanisms used are illustrated by Table 5.

Use of the mechanism:	**Pointcut/advice**	**CI**	**CJImpl**	**CJBinding**
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

Table 5 Use of mechanisms in the CaesarJ examples

(*) One implementation used pointcut and advice but was not considered good practice.

Sousa et al. have used these implementations to compare the use of the pointcut and advice mechanism in both languages to establish that CaesarJ uses it more sparingly than AspectJ.

This study has provided valuable code examples of CaesarJ design patterns and has served as the basis for this thesis. These examples served as guidelines for the new design pattern implementations. Chapter 5 describes the several new design pattern implementations that were developed, mentioning the cases where the original implementations suffered changes.

5. CaesarJ Pattern Implementations

This chapter presents the results obtained from the implementation of the design patterns described in Chapter 2 and the underlying CaesarJ features present in each one. A diagram representation of the pattern implementation is provided for each pattern. Sections 5.1 to 5.11 describe the accomplished pattern implementations while section 5.12 concludes by presenting a brief summary of the implementations.

5.1 Abstract Factory

The implementation of the *Abstract Factory* pattern has not resulted in any reusable CaesarJ module.

Figure 19 shows the general implementation structure of the developed *Abstract Factory* scenarios.

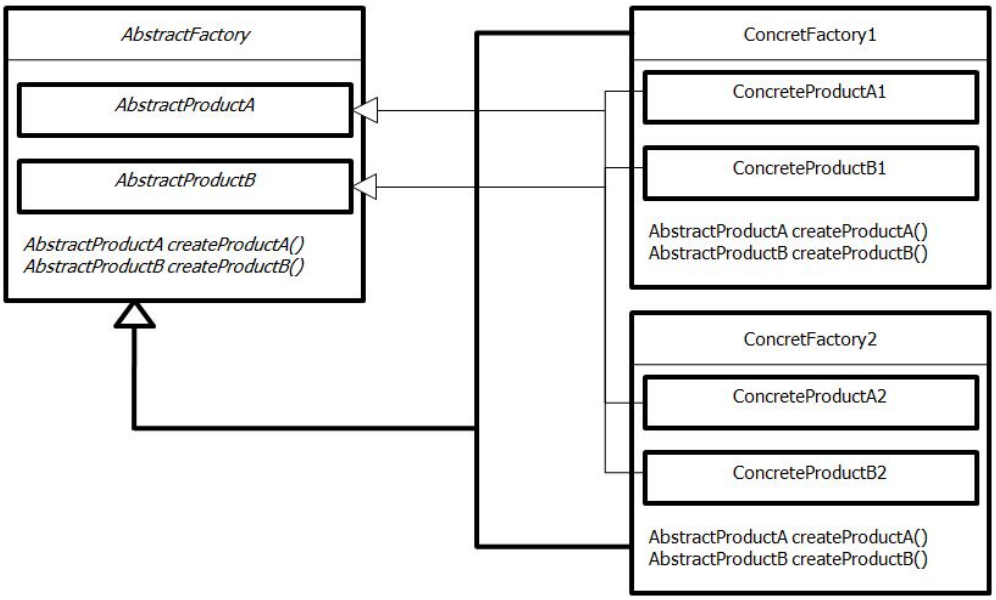

Figure 19 *Abstract Factory* CaesarJ implementation structure

The top level abstract classes represented by *AbstractFactory* contain a variable number of abstract *AbstractProduct* virtual classes that represent the classes of the objects this factory can produce. Each virtual class can declare a variable number of methods that are also declared abstract.

For every type of *AbstractProduct* present in *AbstractFactory*, there is a corresponding top level abstract `createProduct()` method. These `createProduct()` methods are declared in the *AbstractFactory* at the same level as the *AbstractProduct* virtual classes. This way, an *AbstractFactory* acts as an interface of the implementing *ConcreteFactory* sub-classes, specifying the set of virtual classes they must define, the methods these classes have and the methods for the creation of those classes.

Each concrete *ConcreteFactory* class is composed by a set of related *ConcreteProduct* virtual classes, which implement the *AbstractProduct* virtual classes declared in *AbstractFactory* and their methods. *ConcreteFactory* classes implement the `createProduct()` top level methods by returning an instance of this *ConcreteFactory*'s corresponding *ConcreteProduct*. Placing the `createProduct()` methods at the same level as the virtual classes allows the *ConcreteFactory* sub-class instances to create objects of the matching *ConcreteProduct* through methods with the same signature, regardless of the type of *ConcreteFactory*. Different *ConcreteFactory* objects create different *ConcreteProducts* using the same methods because these methods are defined at the level of the *AbstractFactory* they extend. *ConcreteFactory* classes act as a unit of confinement for families of related classes and their implementation, preventing incorrect classes to be mixed.

In one scenario the *AbstractProduct* virtual classes are performed by classes from the standard Java Swing API - `JLabel` and `JButton`. In these cases, the *AbstractProduct and ConcreteProduct* virtual classes disappeared from the *AbstractFactory* and *ConcreteFactory* modules, respectively, due to CaesarJ's support for the inclusion of Java native classes as members of top level classes. Only the `createProduct()` top level methods remained, returning customized `JLabel` and `JButton` objects. An alternative implementation for this scenario was developed, adding a *Label* and a *Button* virtual class as *AbstractProduct*'s to the *AbstractFactory* class, with a `JLabel` and a `JButton` data member in the corresponding virtual class. Although functional, this implementation seems a bit contrived and is not

considered to be as corrected. This alternative implies unnecessary programming overhead, since it adds unnecessary virtual classes that can be instead instanced by simple constructor calls.

Since the goal of *Abstract Factory* is to prevent objects of different family classes to be mixed incorrectly and CaesarJ's virtual class and family polymorphism mechanisms enable the creation of well defined and confined families, it is possible to consider that CaesarJ directly supports the pattern. Any top level class with virtual classes constitutes a factory object because it establishes a relation between the virtual classes it declares, defining a family of related classes. The family polymorphism mechanism present in CaesarJ ensures that objects from different families are not mixed. The impacts in the client code are that a family object of type *AbstractFactory* must be created before instances of type *AbstractProduct* can be created and that the created objects are not Java classes but CaesarJ classes. Since the definition of the family classes is declared in CaesarJ top level classes, the additional structures introduced by OO implementations of *Abstract Factory* disappear.

5.2 Bridge

The CaesarJ implementation of *Bridge* is an aspect component whose structure includes all three kinds of CaesarJ module.

Figure 20 illustrates the structure of the component developed for *Bridge*.

The *Bridge* aspect component includes the *BridgeProtocol* CI, a single CJImpl module and different CJBindings specific to each scenario where the aspect component has been applied. These CJBindings have a more complex structure, and can be seen as having three modules, *BridgeFamily*, *BridgeImpls* and *BridgeAbs*.

The *BridgeProtocol* module is a top level abstract class that comprises two virtual abstract classes that represent the roles involved in this pattern, classes *Abstraction* and *Implementation*.

The *Abstraction* virtual class declares the abstract methods related to the behavior of an entity playing the role of *Abstraction* must carry out, namely:

- `setImplementation(Implementation i)` - sets the *Implementation* of a specific *Abstraction*.
- `getImplementation()` - gets the *Implementation* of a specific *Abstraction*.

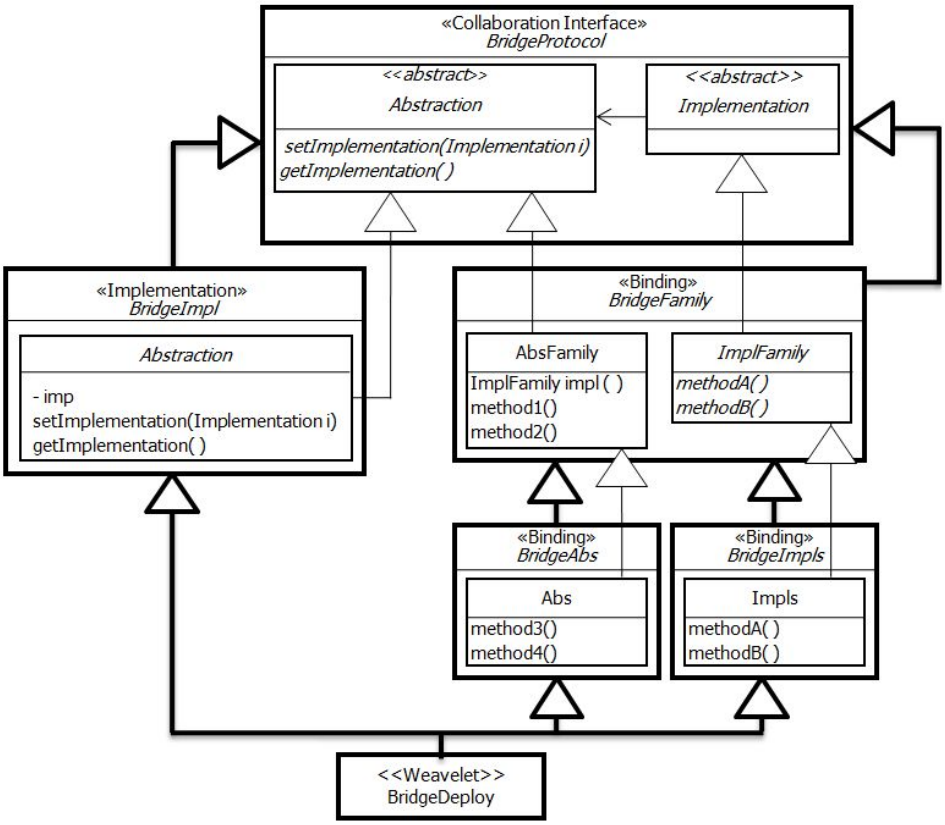

Figure 20 *Bridge* CaesarJ implementation structure

These methods define the operations necessary for a class playing the role of *Abstraction* to dynamically hold, change and retrieve an associated *Implementation*. The *Implementation* virtual class declares no methods because no methods can be generalized to every class playing the role of *Implementation*.

The CJImpls developed for *Bridge* contain a single virtual class *Abstraction* that implicitly extends the virtual class *Abstraction* present in *BridgeProtocol*. The *Abstraction* virtual class present in the CJImpl contains a private *Implementation* data member `imp` that is a reference to this *Abstraction*'s *Implementation*, dynamically associating an *Abstraction* with an *Implementation*. This class uses this data member to implement the `setImplementation(Implementation i)` and `getImplementation()` methods by storing and retrieving the *Implementation* object referenced by `imp`. The composition of the *Abstraction* class with an *Implementation* data member provides a more flexible way of associating these two roles than static inheritance. Since this logic can be generalized for all instances of the *Bridge* pattern, it can be expressed in the CJImpl.

All CJBindings follow a similar structure. There is a top level abstract CJBinding, *BridgeFamily*, which consists of two virtual classes, *AbsFamily* and *ImplFamily*. They declare virtual classes that extend the *Abstraction* and *Implementation* virtual classes in *BridgeProtocol* and adapt them to each concrete scenario of the pattern.

The *AbsFamily* and *ImplFamily* classes represent the *Abstractions* involved in each scenario and the *Implementations* used to implement the *Abstractions*. The *AbsFamily* virtual class is not abstract and must implement the method `impl()`. This method refines the `getImplementation()` method and returns an *ImplFamily* object that corresponds to this family class' specific *Implementation* type. Furthermore, the *AbsFamily* class can also implement a number of methods that make use of the `impl()` method, exemplified by methods `method1()` and `method2()`. These methods correspond to operations of the *AbsFamily* that make use of the methods declared in the *ImplFamily*.

The *ImplFamily* abstract virtual class can declare a number of abstract methods that can be used in the implementation of the methods of the *AbsFamily* virtual class exemplified by methods `methodA()` and `methodB()`. These abstract methods correspond to operations used by the *AbsFamily* that can have different implementations in *ImplFamily* sub-classes.

The structure of the *BridgeFamily* family class reflects the connection between families of related *Abstraction* and *Implementation* classes, represented by the *AbsFamily* and *ImplFamily* in specific scenarios. These classes reflect the concrete relations between an *Abstraction* and an *Implementation* in each specific scenario by the manner how the methods declared in *ImplFamily* are used to implement the methods in *AbsFamily*.

The abstract *BridgeAbs* family class can refine the *AbsFamily* virtual class in *BridgeFamily* by declaring virtual classes *Abs* that extend *AbsFamily*. This classes can then add extra methods to that class, as exemplified by `method3()` and `method4()`.

The abstract *BridgeImpls* family class can refine the *ImplFamily* virtual class in *BridgeFamily* by declaring virtual classes *Impls* that extend *ImplFamily*. This classes can then implement methods `methodA()` and `methodB()`. The alternative implementation of these methods is used to flexibly provide *Abstractions* with different *Implementations*.

To instantiate a *Bridge* aspect component, a Weavelet must be created to unite the Bridge*Impl*, *BridgeAbs* and *BridgeImpls* modules through mixin composition, illustrated by *BridgeDeploy*. The *BridgeAbs* and *BridgeImpls* modules could have been implemented in a

single class family, and the mixin would use the module where they were implemented. However, since the Weavelet enables mixin composition of more than two modules, it was chosen to create a Weavelet that would unite the three separate modules. This separation into three distinct modules has the advantage of placing related roles in smaller modules, which enhances readability and understanding of the code.

With this aspect component, all the logic related with the pattern facet has been removed from the original classes and placed in one of the aspect modules. The impact on the client code is that the Java classes were replaced by CaesarJ classes. A `final` *BridgeDeploy* family object must be created to instance new objects of types *Abstraction* and *Implementation*. This family object supplies family polymorphism to the instances it creates. This way, it is not possible to set an *Abstraction* with an *Implementation* if they are instances of different family objects. Family polymorphism ensures that objects of different families are not mixed.

5.3 Builder

The implementation of the *Builder* pattern has not resulted in any reusable CaesarJ modules.

Figure 21 illustrates the structure of the component developed for *Builder*.

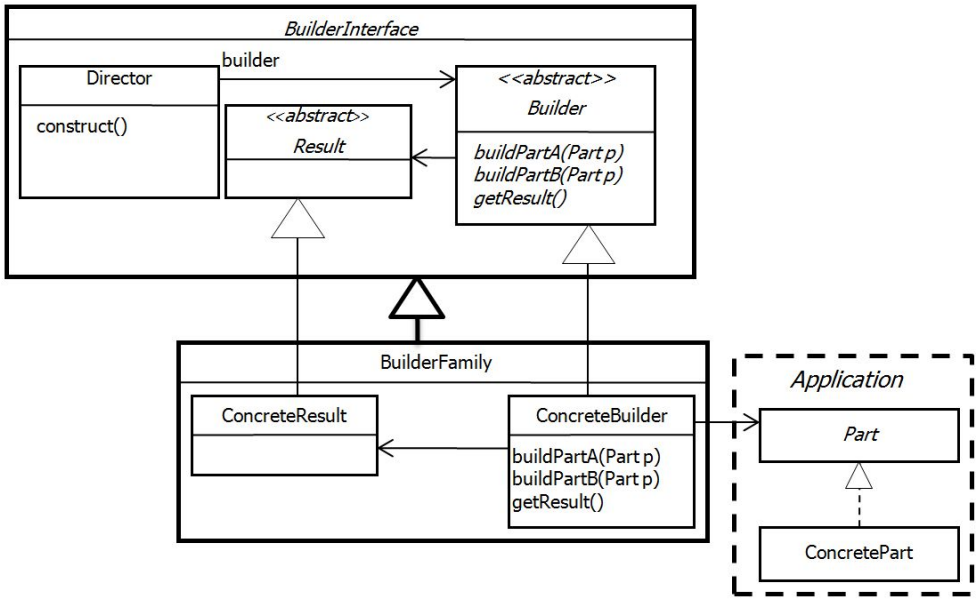

Figure 21 *Builder* CaesarJ implementation structure

The *Builder* aspect component is composed by the *BuilderInterface* and *BuilderFamily* modules.

The *BuilderInterface* includes three virtual classes, classes *Director*, *Result* and *Builder*.

The *Builder* abstract class is responsible for the building process. It declares a set of building operations exemplified by the `buildPartA(Part p)` and `buildPartB(Part p)` methods. The *Builder* class also declares the `getResult()` method that returns the final *Result* after the building operations are finished.

The *Result* class represents a structure where the built parts are stored while the building operation takes place. The *Result* class reflects the different manners by which the result of the building can be represented in a flexible way. In the two scenarios studied for this pattern, *Result* was performed by a custom Java class *Media* (in the Eckel scenario) and Java's `String` class (in the Hannemann et al. scenario).

In the Eckel scenario, class *Media* emulates an `ArrayString`. In the original Java scenario, *Media* extended `ArrayList`. Since CaesarJ **cclasses** cannot extend Java classes, the alternative was to compose *Media* with an `ArrayList` data member and adapt the default constructor call to create a new `ArrayList` and the `toString()` method to call the `toString()` method in `ArrayList`. It was also necessary to implement a `getList()` method that would return the `ArrayList` data member. In the Hannemann et al. scenario, the *Result* role is performed by the Java API `String` class. Therefore, it was not necessary to perform any adaptation.

The concrete *Director* class implements the `construct()` method, which represents the generic action of building all the parts of the final product and returning the finished *Result*. In the two scenarios implemented, the `construct()` method took an argument of different types. In the Eckel scenario, the *Director* class has a *Builder* data member and the `construct()` method took an argument of type `List`. The method would traverse through this data structure and make the refereneced *Builder* build all the *Part* objects in that structure. In the Hannemann scenario, the `construct()` method took an argument of type *Builder*. The method would execute a fixed sequence of building actions performed by the *Builder* in the argument.

The *BuilderFamily* top level class extends *BuilderInterface* and has two virtual classes, *ConcreteBuilder* and *ConcreteResult*. Placing *ConcreteBuilder* and *ConcreteResult* classes in

the same *BuilderFamily* illustrates the strong relation between both virtual classes. The definition of a *ConcreteBuilder* and *ConcreteResult* in a common *BuilderFamily*, reflects the connection between a *Builder* and the *Result* it produces.

ConcreteResult refines the *Result* class in *BuilderInteface*. This proved necessary in the Eckel scenario, where there were 3 different classes in the original Java scenario that extend the *Media* class. This way, different family classes of the *Builder* scenario produce different *Results*.

ConcreteBuilder refines the *Builder* class declared in *BuilderInterface* by implementing the `buildPart(Part p)` methods. This makes it possible that the same method can produce and store different part of the final result differently, while ensuring family class consistency. The `getResult()` method returns this *BuilderFamily* specific *ConcreteResult* object with the final result.

With this aspect component, all the logic related with the pattern facet has been removed and placed in one of the aspect modules, leaving the Java classes that perform the role of *Part* and the *ConcretePart* classes that extend them in the application domain. The implication on the client code is that, for every type of *BuilderFamily*, a family object of type *BuilderFamily* must be created before instancing *ConcreteBuilder* and *Director* objects. This provides additional safety given by family polymorphism. Family polymorphism ensures that *ConcreteBuilder* and *Director* objects from different *BuilderFamily* objects are not mixed.

5.4 Chain of Responsibility

The CaesarJ implementation of *Chain of Responsibility* is an aspect component whose structure includes all three kinds of CaesarJ module.

Figure 22 illustrates the structure of the component developed for *Chain of Responsibility*.

The *Chain of Responsibility* aspect component consists of the *ChainOfResponsabilityProtocol* CI, the *CoRImpl* CJImpl and the *CoRBindings* CJBindings to compose the aspect component to the individual scenarios where the aspect component was deployed.

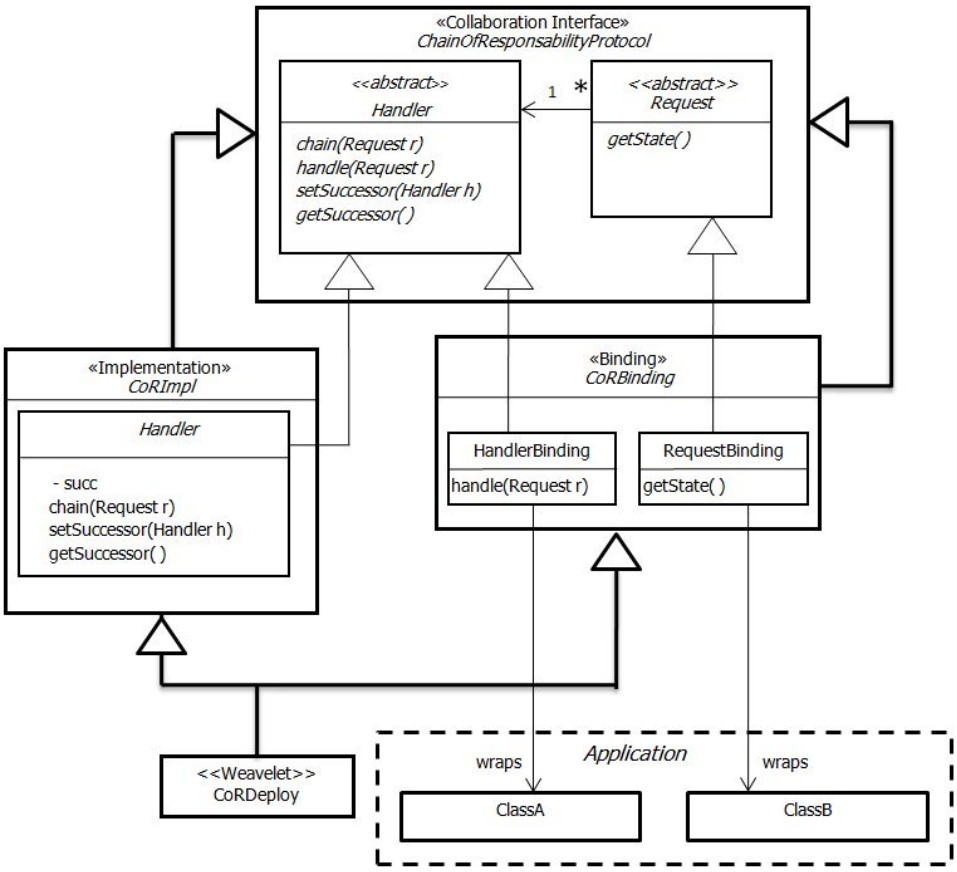

Figure 22 *Chain of Responsibility* CaesarJ implementation structure

ChainOfResponsabilityProtocol includes two virtual classes, *Request* and *Handler*, which represent the roles involved in the pattern.

The *Handler* virtual class represents an entity responsible for handling *Requests* or passing them along a chain of responsibility to other *Handlers*.

It declares the following methods:

- `chain(Request r)` – checks whether this *Handler* can handle this request or if should forward the *Request* to another *Handler* in the chain of responsibility.
- `handle(Request r)` – tries to handle this *Request* and returns a boolean value whether it handled the *Request* successfully or not.
- `setSuccessor(Handler h)` – sets the next *Handler* in the chain of responsibility.
- `getSuccessor()` – gets the next *Handler* in the chain of responsibility.

The *Request* virtual class declares one method:

- `getState()` – returns information about a particular *Request*'s state.

Both the virtual classes in *ChainOfResponsabilityProtocol* and their methods are declared abstract.

The *Handler* virtual class in the *CoRImpl* CJImpl implements the methods responsible for the request handling and forwarding policy. This is because the implementation of a chain of responsibility does not depend on a specific scenario and there can be several alternative ways of implementing this chain. This class implements all methods in the *Handler* class, except `handle(Request r)`. It contains a private *Handler* data member `succ` that is a reference to the next *Handler* in the chain. This class implements the `setSuccessor(Handler h)` and `getSuccessor()` methods by storing and retrieving the *Handler* object referenced by `succ`. If the `getSuccessor()` method is not successful, i.e., if there is no successor to this *Handler*, `getSuccessor()` raises a *ChainOfResponsibilityException*. The `chain(Request r)` method takes a *Request* and verifies if this *Handler* is able to handle it, by calling the `handle(Request r)` method. This is possible because `handle(Request r)` is declared, although not implemented, in the CI. If the *Handler* is not able to handle the *Request*, it gets the next *Handler* in the chain with `getSuccessor()` and forwards the *Request* to the successor *Handler* by calling the `chain(Request r)` method again. These methods are implemented in the CJImpl because they hold the rationale behind the *Chain of Responsibility* pattern. They do not determine if a *Handler* is able to handle a request, but rather the operations that must take place if a *Handler* succeeds in handling a *Request* or not. The logic of the handling operations is context sensitive and should be specified in the CJBindings.

The CJBindings for *Chain of Responsibility* declare *HandlerBinding* and *RequestBinding* virtual classes that extend the *Handler* and *Request* virtual classes in *ChainOfResponsabilityProtocol*. Other than extending the virtual classes in the CI, the virtual classes also wrap plain Java classes in an application, composing them with the additional logic from the roles of the pattern.

The *HandlerBinding* virtual classes wrap the classes in the application that will perform the role of *Handler* and implement the `handle(Request r)` method. This method will define

the condition under which a *Handler* is able to handle a *Request* using operations in its wrappee.

The *RequestBinding* virtual classes wrap the classes in the application that will perform the role of *Request* and implement the `getState()` method. The `getState()` method returns information about the wrappee's state. The *RequestBinding* makes it possible for custom classes to perform the role of *Request*, but also for native Java classes to perform that role, such as `String` or `Integer`. This design makes this component more extensible, since this role is not restricted to a single class.

Finally, the CJBinding must define which events should trigger the handling logic in the *Chain of Resposibility* module. This is accomplished by pointcut and advice mechanisms. The pointcut defined in the CJBinding must specify the events that raise handling request events and the advice for that pointcut starts the chain of responsibility, by calling the `chain(Request r)` method.

To combine the *CoRImpl* and *CoRBinding* modules, a *CoRDeploy* Weavelet must be created so the aspect component can be instantiated.

Although only one CJImpl was developed for the *Chain of Responsibility* pattern, others could have been created. It would be a question of defining a different handling logic or a different manner of building the chain of responsibility.

With this aspect component, all the logic related with the pattern facet has been removed and placed in one of the aspect modules. This implementation of the pattern has an impact on the client code in four manners:

- A family object of type *CoRDeploy* must be created.
- *HandlerBinding* and *RequestBinding* objects must be created to wrap the objects of the application domain that perform the corresponding roles.
- The creation of the chain of responsibility must use the *HandlerBinding* and *RequestBinding* wrapper objects instead of the plain Java objects.
- The *CoRDeploy* object must be deployed so the aspect can trigger the pattern logic at the events captured by the pointcuts defined in the CJBindings.

5.5 Composite

The CaesarJ implementation of *Composite* is an aspect component whose structure includes all three kinds of CaesarJ module.

Figure 23 illustrates the structure of the component developed for *Composite*.

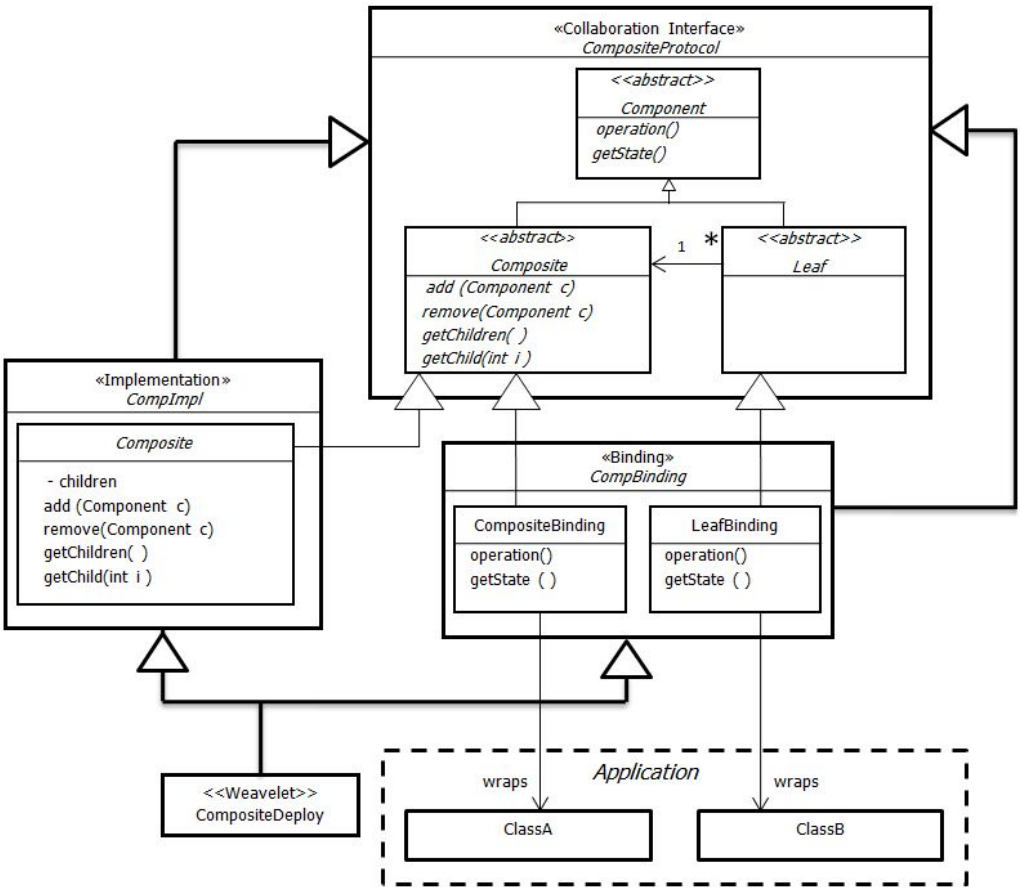

Figure 23 *Composite* CaesarJ implementation structure

The *Composite* aspect component includes the *CompositeProtocol* CI, two alternative CJImpl modules and different CJBindings specific to each scenario where the aspect component has been applied.

The *CompositeProtocol* module is a top level abstract class that comprises three virtual abstract classes that represent the roles involved in this pattern, classes *Component*, *Composite* and *Leaf*. Since both *Composite* and *Leaf* are specific kinds of *Component*, they are declared as sub-classes of that virtual class. This way, CaesarJ enables the *Component*

abstract virtual class to act as an interface for both kinds of components of the *Composite* pattern.

The *Component* virtual class declares the methods that both *Composite* and *Leaf* must implement, specifically:

- `operation()` – a generic operation performed on *Component* entities.
- `getState()` – returns information about a *Component*'s state.

Since these operations are common both to *Composite* and *Leaf* they can be abstract into a higher level, illustrated by the *Component* virtual class.

The *Composite* virtual class declares the methods specific to the entities that will perform the role of *Composite*. These methods are responsible for adding, removing and returning this *Composite* child *Components*. These methods are:

- `add(Component c)` – adds a child *Component* to this *Composite*.
- `remove(Component c)` – removes a child *Component* from this *Composite*.
- `getChildren()` – returns a `Collection` of every *Component* children of this *Composite*.
- `getChild(int i)` – returns a child *Component* of this *Composite* in a specific position.

Since *Leaf* must only implement the methods declared in *Component*, it declares no additional methods.

The *CompImpl* CJImpls developed for *Composite* contain a single virtual class *Composite* that implicitly extends the virtual class *Composite* in *CompositeProtocol* and implements all its methods. To deal with the addition and removal of *Components* of a particular *Composite*, a data structure `children` is included in the CJImpls. This data structure is responsible for the storage of *Components* related to each individual *Composite*. The `add(Component c)` and `remove(Component c)` methods perform operations on this data structure, the `getChildren()` methods returns the data structure with all the *Component* children and the `getChild(int i)` method traverses through the data structure returning a component in a specific position. Since `getChildren()` returns an object of type

`Collection` it is possible to create alternative CJImpls using different data structures. In the two developed CJImpls, `ArrayList` and `WeakHashMap` data structures were used.

Different CJBindings were developed to compose the aspect component to the specific scenario where the pattern is to be deployed. These methods are implemented in the CJImpl because they comprise the context independent part of the pattern. They are common to every scenario of *Composite* and can therefore be abstracted to a CJImpl module.

The *CompBinding* CJBindings declare virtual classes that extend the *Composite* and *Leaf* virtual classes in *CompositeProtocol* and wrap plain Java classes that will perform these roles.

Both the classes that extend the *Composite* and *Leaf* virtual classes in *CompositeProtocol* implement the `getState()` and `operation()` methods. The `getState()` method will return information about the Java class that performs the role of *Composite* or *Leaf*, and the `operation()` method will perform a different generic operation on a *Component* object, distinguishing whether it was called by a *Composite* or a *Leaf*.

In the *Composite* scenario by Cooper it proved necessary to add three auxiliary methods to the *Component* entities. These methods had to be added in the *Component* role, so both *Composite* and *Leaf* entities could perform them. Since this was necessary in only one scenario, these methods should not be added in *CompositeProtocol*. Due to their specific nature, it was appropriate to add them in the CJBinding. This way, an abstract virtual class *Component* was added to the CJBinding, containing three abstract methods. This virtual class implicitly refined the *Component* virtual class, adding these extra methods. This was a flexible way of extending the *Component* virtual class, adding extra methods to both *CompositeBinding* and *LeafBinding*. This approach allowed for polymorphic method calls, which removed the use of several `instanceof` clauses in the CaesarJ implementation of this scenario. The trade-off is that when it is necessary to access the methods of the *Component* objects defined in the CJBinding, it became mandatory to perform static casts to this particular family class.

With this aspect component, all the logic related with the pattern facet has been removed from the original Java classes and placed in one of the aspect modules. This implementation of the pattern has an impact on the client code in three manners:

- A family object of type *CompositeDeploy* must be created.

- *CompositeBinding* and *LeafBinding* objects must be created to wrap the objects of the application domain that perform the corresponding roles.
- The creation of the *Composite* structure must use the *CompositeBinding* and *LeafBinding* wrapper objects instead of the plain Java objects.

5.6 Decorator

The implementation of the *Decorator* pattern has not resulted in any reusable CaesarJ modules.

Figure 24 illustrates the structure of the component developed for *Decorator*.

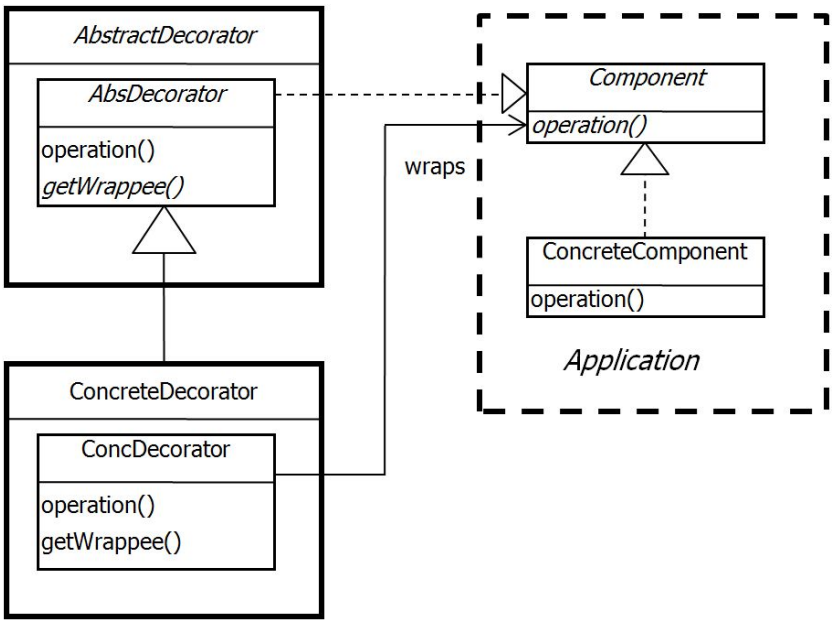

Figure 24 *Decorator* CaesarJ implementation structure

The *Decorator* aspect component is composed by the *AbstractDecorator* and *ConcreteDecorator* modules.

The *AbstractDecorator* abstract top class has one virtual class, also abstract, *AbsDecorator*. This virtual class implements the *Component* interface present in the application and its methods, represented by the `operation()` method. To implement this method, this virtual class declares an abstract method `getWrappee()` that returns the wrappee

of a wrapper class. This is necessary because *AbsDecorator* does not wrap any class but still, the `operation()` method must be implemented based on the classes in the application.

The *ConcreteDecorator* family class holds virtual classes that will act as wrappers for Java classes in the application. These classes are represented by the *ConcDecorator* virtual class and extend the *AbsDecorator* virtual class in *AbstractDecorator*. Since these classes declare they wrap the *Component* interface, they can wrap any class of type *Component*. This way, they can wrap both the Java classes in the application and other *AbsDecorator* classes and their sub-classes, like *ConcDecorator*.

This means that it is possible to compose wrappers independently of their order, since all wrapper classes can also be wrapped themselves. Nevertheless, one limitation to CaesarJ's wrapper mechanism has proven to be an obstacle. One of the scenarios of the *Decorator* pattern implicated that one wrapper class should be able to wrap *two* objects. This is currently not possible in CaesarJ. The alternative is to compose such classes with a data member of the extra class to be wrapped and manage it like common Java wrapper classes. This implies abdicating the wrapper recycling mechanism. The CaesarJ developers have been working on a mechanism that enables one wrapper class to wrap more than one class [23]. Another issue in the *Decorator* implementation is the fact that wrapping relations are not inherited by sub-classes. This is why it is necessary for every *ConcDecorator* virtual class to wrap *Component* classes. Preferably, *AbsDecorator* should wrap *Component* and provide a default implementation for the `operation()` method.

With this aspect component, all the logic related with the pattern facet has been removed from the original Java classes and placed in one of the aspect modules. In order to decorate a class in the application domain, a *ConcreteDecorator* object must be created. After this object is created, *Component* objects in the application domain can be dynamically decorated by wrapping them with *ConcDecorator* virtual classes.

5.7 Factory Method

The implementation of the *Factory Method* pattern has not resulted in any reusable CaesarJ module.

Figure 25 represents the generic structure of the implementations developed for *Factory Method*.

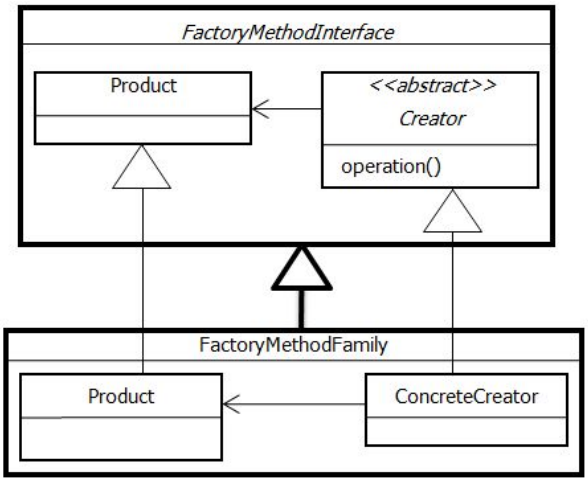

Figure 25 *Factory Method* CaesarJ implementation structure

The top level abstract class *FactoryMethodInterface* declares two virtual classes, *Product* and *Creator*.

The *Creator* abstract virtual class is characterized by an `operation()` method that creates an instance of the *Product* class by means of a constructor call. That is why the *Product* virtual class is not declared abstract, unlike *Creator*. However, since *FactoryMethodInterface* is declared as abstract, it is not possible to create an instance of a *FactoryMethodInterface* family object, and consequentially the concrete *Product* instance will be defined by the type of the *FactoryMethodFamily* family object.

FactoryMethodFamily class families refine the virtual classes defined in *FactoryMethodInterface* and act as a unit of confinement for related classes, preventing *Product* and *ConcreteCreator* classes from different families to be mixed incorrectly. More importantly, the *Product* virtual class in *FactoryMethodFamily* implicitly extends the *Product* virtual class in *FactoryMethodInterface*.

This implicit inheritance mechanism makes it possible to create different kinds of objects with the same constructor call in the `operation()` method. The kind of concrete *Product* object depends on the kind of *FactoryMethodFamily* family object instance, thanks to the CaesarJ's virtual class mechanism.

Since the same constructor call can create different kinds of objects, one can say that CaesarJ enables polymorphic constructors. As the goal of *Factory Method* is to provide a

single method that creates different objects, depending on the context of its call, it is possible to consider that CaesarJ directly supports the pattern.

5.8 Mediator

The CaesarJ implementation of *Mediator* is an aspect component whose structure includes all three kinds of CaesarJ module.

Figure 26 illustrates the structure of the component developed for *Mediator*.

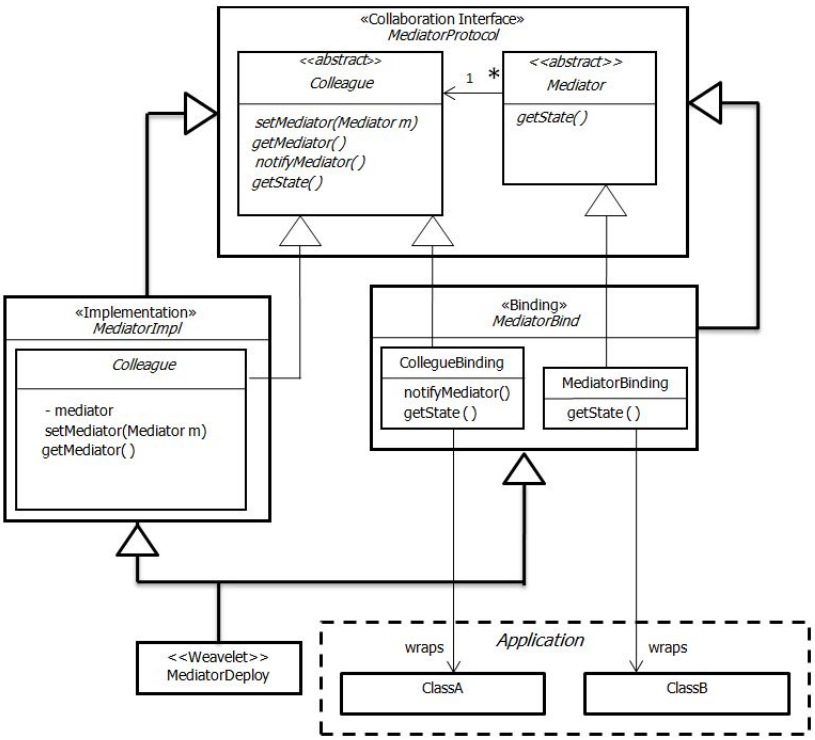

Figure 26 *Mediator* CaesarJ implementation structure

The *Mediator* aspect component includes the *MediatorProtocol* CI, a single CJImpl module and different CJBindings specific to each scenario where the aspect component has been applied.

The *MediatorProtocol* module is a top level abstract class that comprises two virtual abstract classes that represent the roles involved in this pattern, classes *Colleague* and *Mediator*.

The *Colleague* virtual class declares the methods related to the behavior an entity playing the role of *Colleague* must carry out. These methods are:

- `setMediator(Mediator m)` - sets the *Mediator* of a specific *Colleague*.
- `getMediator()` - gets the *Mediator* of a specific *Colleague*.
- `notifyMediator()` - responsible for the *Colleague*'s notification logic.
- `getState()` – returns information about a particular *Colleague*'s state.

The *Mediator* virtual class declares one method:

- `getState()` – returns information about a particular *Mediator*'s state.

The *MediatorImpl* CJImpls developed for *Mediator* contains a single virtual class *Colleague* that implicitly extends the virtual class *Colleague* present in *MediatorProtocol*.

The *Colleague* virtual class present in the CJImpl contains a private *Mediator* data member `mediator` that is a reference to this *Colleague*'s *Mediator*. This class implements the `setMediator(Mediator m)` and `getMediator()` methods by storing and retrieving the *Mediator* object referenced by `mediator`. These methods are implemented in the *Colleague* virtual class in *MediatorImpl*, because these methods constitute the context independent facet of the pattern.

The *MediatorBind* CJBindings declare virtual classes that extend the *Colleague* and *Mediator* virtual classes in *MediatorProtocol*. The classes from the CJBinding wrap plain Java classes that will perform the roles defined by the pattern.

The *MediatorBinding* virtual class extends *Mediator* and implements the `getState()` method in a context sensitive manner, suitable for the given scenario.

The *ColleagueBinding* virtual class extends *Colleague* and declares which Java class will perform the role of *Colleague*, again through a wrapper declaration. This class also implements the `getState()` method by returning information about the wrappee and the `notifyMediator()` method. The `notifyMediator()` method will use the `getMediator()` method to retrieve this *Colleague*'s *Mediator*, the `getState()` method to retrieve the Java class that performs that role in the application and then some operation with the notification logic contained in that Java class. Both the `getState()` and `notifyMediator()` methods are implemented in the CJBindings because they are context specific methods whose implementation depends on the specific scenario where the aspect is composed.

To capture the events in a *Colleague* that should start the notification of *Mediators*, the CJBinding also uses pointcut and advice mechanisms. The CJBindings contain pointcut declarations that capture the relevant change events in a *Colleague*. Then, an advice triggers the actions that should be performed when this event takes place. In the case of the *Mediator* pattern, this advice triggers the notification event on a *Colleague*, and the consequent `notifyMediator()` method.

There were scenarios where the *Colleague* virtual classes in the CJBinding also implemented the `setMediator(Mediator m)` method. Such was the case when the *Colleague* had to store its *Mediator* but the *Mediator* also had to store its *Colleague* in a particular data member of the *Mediator* wrappee class. There are two alternatives for carrying out this operation:

- Use a `super.setMediator(Mediator m)` call, taking advantage of the method implementation in *MediatorImpl* and then perform the additional actions on the *Mediator* wrappee class. This is possible because the CI effectively acts as a communication interface between the CJBinding and the CJImpl.
- Create pointcuts and advices in the CJBinding that would capture the calls to `setMediator(Mediator m)` and perform the actions on the *Mediator* Java class.

The first alternative has been chosen because it is more extensible, since pointcuts are not flexible by nature.

To instantiate a *Mediator* aspect component, a Weavelet must be created to unite both the *MediatorImpl* and *MediatorBinding* modules through mixin composition. However, in the case of the *Mediator* pattern, the mixin composition order must present the CJBinding module prior to the CJImpl module. This is because the mixin order must always be serialized [6]. Since there are two implementations of the `setMediator(Mediator m)` method, the mixin composition order must reflect that the most specific implementation of that method is in the CJBinding, hence it should be declared first.

With this aspect component, all the logic related with the pattern facet has been removed from the classes that perform the role of *Colleague* and placed in one of the aspect modules. This was not possible in the classes that perform the roles of *Mediator*. This was not to be expected, because the role of *Mediator* is *defining*, contrary to the *Colleague* role, which is

superimposed [28]. This implementation of the pattern has an impact on the client code in four manners:

- A family object of type *MediatorDeploy* must be created.
- *ColleagueBinding* and *MediatorBinding* objects must be created to wrap the objects of the application domain that perform the corresponding roles.
- The relations between *Colleague* and *Mediator* objects must be defined using the *ColleagueBinding* and *MediatorBinding* wrapper objects instead of the plain Java objects.
- The *MediatorDeploy* object must be deployed so the aspect can trigger the pattern logic at the events captured by the pointcuts defined in the CJBindings.

5.9 Observer

The CaesarJ implementation of *Observer* is an aspect component whose structure includes all three kinds of CaesarJ module.

Figure 27 illustrates the structure of the component developed for *Observer*.

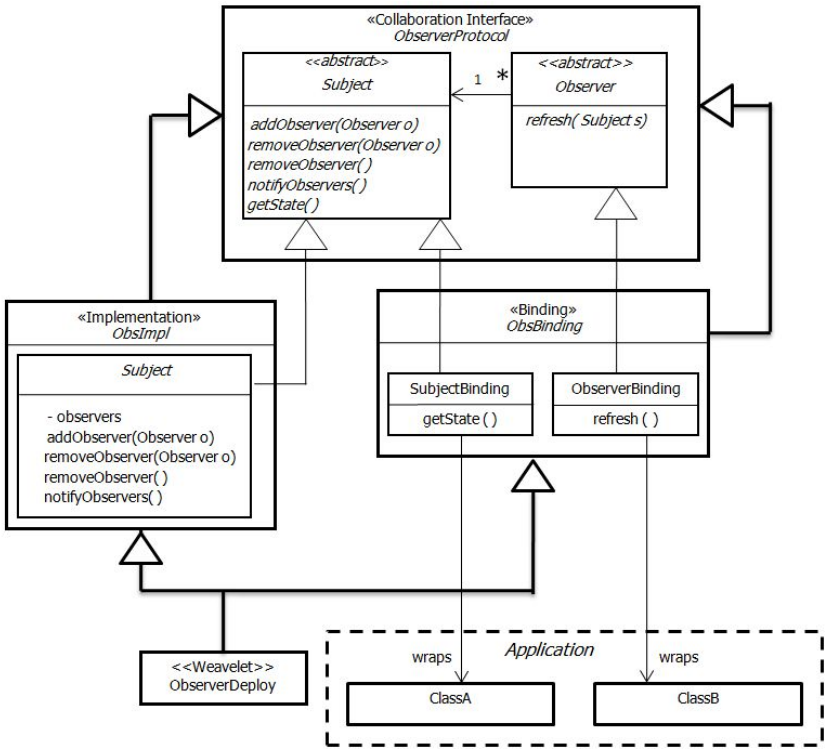

Figure 27 *Observer* CaesarJ implementation structure

The *Observer* aspect component includes the *ObserverProtocol* CI, several alternative CJImpl modules and different CJBindings specific to each scenario where the aspect component has been applied.

The *ObserverProtocol* module is a top level abstract class that comprises two virtual abstract classes that represent the roles involved in this pattern, classes *Subject* and *Observer*.

The *Subject* virtual class declares the methods related to the behavior an entity playing the role of *Subject* must carry out. These methods are:

- `addObserver(Observer obs)` - adds an *Observer* to this *Subject*'s set of interested *Observers*.
- `removeObserver(Observer obs)` - removes an *Observer* from this *Subject*'s set of interested *Observers*.
- `removeObserver()` - removes all *Observers* from this *Subject*'s set of interested *Observers*.
- `notifyObservers()` - responsible for the *Subject's* notification logic.
- `getState()` – returns information about a particular *Subject*'s state.

The *Observer* virtual class declares one method:

- `refresh(Subject s)` - responsible for the update of the *Observer* state after a *Subject* notifies that its state has changed.

Three different CJImpls have been developed for *Observer*. The developed CJImpls contain a single virtual class *Subject* that implicitly extends the virtual class *Subject* present in *ObserverProtocol* and implements all its methods, except for `getState()`. To deal with the addition, removal and notification of *Observers* of a particular *Subject*, a data structure `observers` is included in *ObsImpl1* and *ObsImpl2*. This data structure is responsible for the storage of *Observers* related to each individual *Subject*. *ObsImpl3* takes advantage of the Java API *Observer* and *Observable* classes. These methods are implemented in the *Subject* virtual class in *ObsImpl*, because these methods constitute the context independent facet of the pattern.

The *ObsBinding* CJBindings declare virtual classes that extend the *Subject* and *Observer* virtual classes in *ObserverProtocol*. The classes from the CJBinding wrap plain Java classes that will perform the roles defined by the pattern.

The virtual class that extends *Subject* implements the `getState()` method in a context sensitive manner, suitable for the given scenario.

The virtual class that extends *Observer* implements the `refresh(Subject s)` method and declares which Java class will perform the role of *Observer*, again through a wrapper declaration. Both the `getState()` and `refresh(Subject s)` methods are implemented in the CJBindings because they are context specific methods whose implementation depends on the specific scenario where the aspect is composed.

To capture the events of a *Subject*'s state change, the CJBinding also makes use of pointcut and advice mechanisms. The Bindings contain pointcut declarations that capture the relevant change events in a *Subject*. Then, an advice triggers the actions that should be performed when this event takes place. In the case of the *Observer* pattern, this advice triggers the notification event on a *Subject*, and the subsequent `notifyObservers()` method.

It is worth mentioning that, according to the scenario, a CJBinding can have a variable number of virtual classes depending on the number of classes that play a certain role. As a consequence, if several Java classes play a role within the pattern, we simply have to define a wrapper class for each.

Like the CJImpl virtual class, the CJBinding is also declared to be abstract. To instantiate the aspect component in a certain scenario, a related Weavelet has to be created to realize the complete component. The Weavelet uses mixin composition to unite the different parts of the component implemented both in the CJImpl and CJBinding.

With this aspect component, all the logic related with the pattern facet has been removed from the original classes and placed in one of the aspect modules. This implementation of the pattern has an impact on the client code in four manners:

- A family object of type *ObsDeploy* must be created.
- *SubjectBinding* and *ObserverBinding* objects must be created to wrap the objects of the application domain that perform the corresponding roles.

- The relations between *Subject* and *Observer* objects must be defined using the *SubjectBinding* and *ObserverBinding* wrapper objects instead of the plain Java objects.
- The *ObsDeploy* object must be deployed so the aspect can trigger the pattern logic at the events captured by the pointcuts defined in the CJBindings.

5.10 Prototype

The CaesarJ implementation of *Prototype* is an aspect component whose structure includes all three kinds of CaesarJ module.

Figure 28 illustrates the structure of the component developed for *Prototype*.

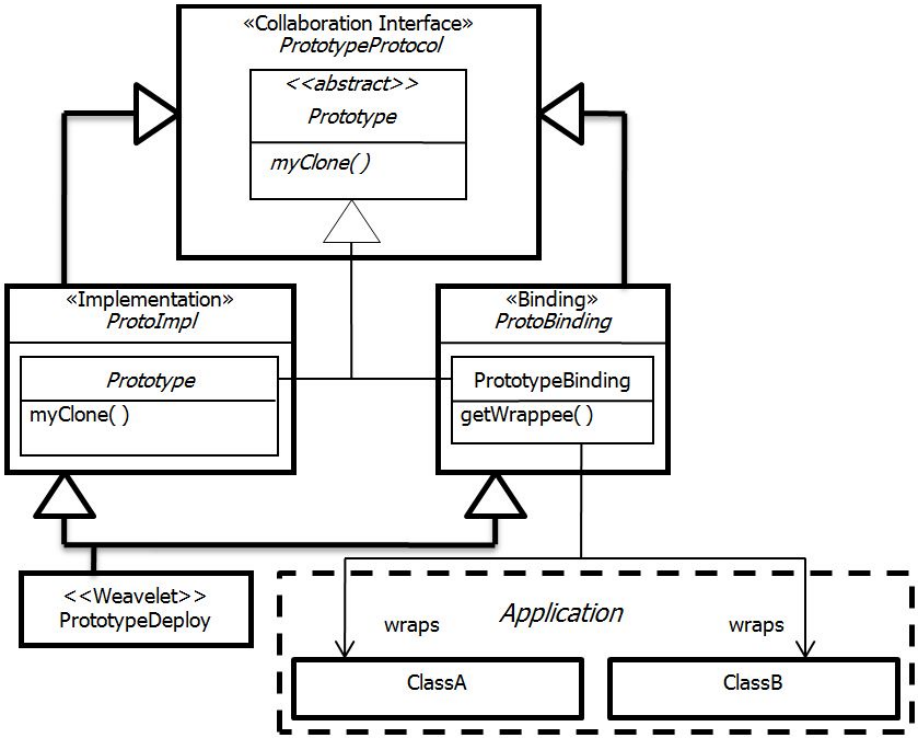

Figure 28 *Prototype* CaesarJ implementation structure

The *Prototype* aspect component includes the *PrototypeProtocol* CI, two CJImpl modules and different CJBindings specific to each scenario where the aspect component has been applied.

Since *Prototype* deals only with an action to perform copies of objects, this is the aspect component with the simpler structure.

The *PrototypeProtocol* module is a top level abstract class that comprises one virtual abstract class that represents the role involved in this pattern, class *Prototype*.

The *Prototype* virtual class is concerned with one operation, creating copies of itself. Hence, *Prototype* has one abstract method:

- `myClone()` – creates a replica of this object.

To implement this virtual class and its single method, two CJImps have been developed. These CJImps distinguish between two cloning methods, shallow or deep clones. Although two CJImps were developed, only one of them is functional, the shallow clone variation. The reasons why the deep clone variation is not functional will be discussed in the end of this section.

The *Prototype* virtual class present in the CJImp implements the `myClone()` method in the *Prototype* virtual class declared in the *PrototypeProtocol* class. The shallow clone CJImp makes use of the Java API maker interface `Cloneable` to produce a shallow copy of the object. This is possible because **cclasses** can implement Java interfaces. The virtual class in the CJImp declares it implements the `Cloneable` interface and calls the Java API clone method.

The CJBindings in this pattern simply attach the cloning operation to Java classes in the application through wrappers. The virtual classes in the CJBinding extend the *Prototype* class in *PrototypeProtocol* and declare which classes in the application they wrap, enabling them to create clones of themselves. The wrappee can be retrieve by calling the `getWrappee()` method. Since the cloning operation is implemented in the CJImp module, neither the inner classes in the CJBindings or the Java classes they wrap need to implement the `Cloneable` marker interface. This effectively separates the aspect facet from the classes of the application.

The CJImp that should generate a deep clone was not successfully implemented. The `myClone()` method implementation that should produce a deep clone copy of an object would had to resort to the Java API `Serializable` marker interface. However, it was discovered that CaesarJ's inner class mechanism differs from the one in Java. Several attempts were made to produce a functional CJImpl that supported the deep clone implementation of `myClone()`, however that was not possible. Although the aspect

component is compilable, when the `myClone()` method is called a `NotSerializableException` is raised.

With this aspect component, all the logic related with the pattern facet has been removed from the original classes and placed in one of the aspect modules. Its limitation to greater reusability is its constant use of static casts. This implementation of the pattern has an impact on the client code in three manners:

- A family object of type *PrototypeDeploy* must be created.
- *PrototypeBinding* objects must wrap the objects of the application domain to enhance them with the cloning operation.
- The cloning operations must be performed on the wrapper objects, which will return copies of the wrapper objects. The plain Java class can be retrieved by a `getWrappee()` call.

5.11 Visitor

The implementation of the *Visitor* pattern has not resulted in any reusable CaesarJ modules.

Figure 29 illustrates the structure of the component developed for *Visitor*.

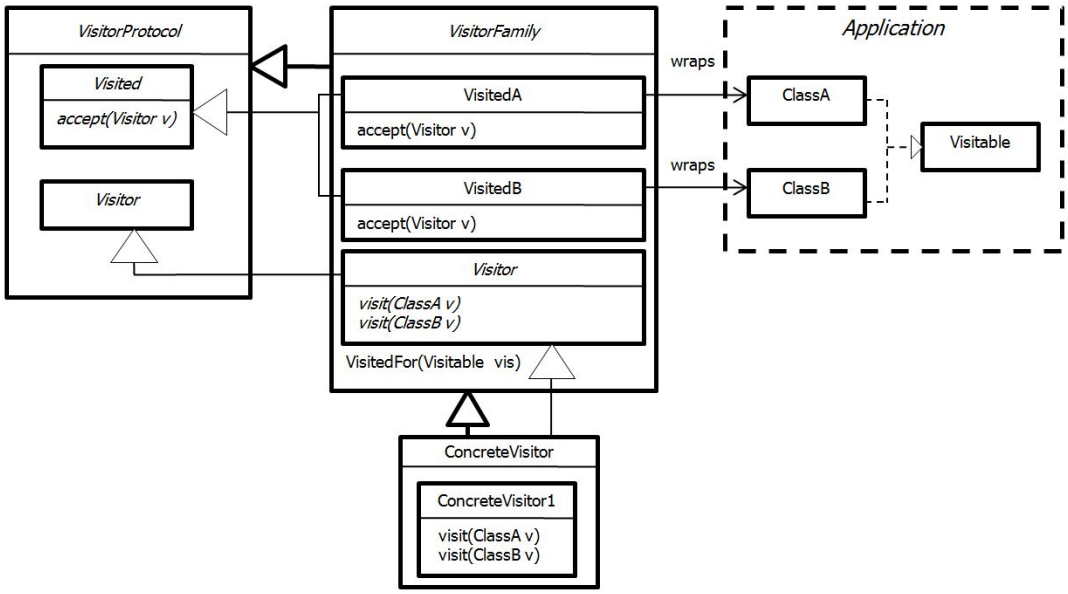

Figure 29 *Visitor* CaesarJ implementation structure

The *Visitor* aspect component includes the *VisitorProtocol*, *VisitorFamily* and *ConcreteVisitor* modules.

The *VisitorProtocol* top level abstract class encloses two virtual abstract classes that represent the roles involved in this pattern, classes *Visited* and *Visitor*, both abstract.

The *Visited* abstract virtual class declares the `accept(Visitor v)` method. This method tells an object of class *Visited* to accept a *Visitor* object that will then add an extra operation to that *Visited*. This method is declared abstract.

The *Visitor* abstract virtual class in *VisitorProtocol* declares no methods and serves only as a virtual class for other virtual classes to refine, according to the scenario where the pattern is applied and the operations to be added.

The *VisitorFamily* module contains two kinds of virtual classes, the classes that extend the *Visited* virtual class in *VisitorProtocol* represented by classes *VisitedA* and *VisitedB*, and the *Visitor* virtual class, that implicitly extends the *Visitor* class in *VisitorProtocol*.

The *VisitedA* and *VisitedB* classes in *VisitorFamily* wrap different plain Java classes in the application, enabling them to receive the additional behavior added by the *Visitor* objects.

VisitorFamily also has a top level method `VisitedFor(Visitable vis)`. The `VisitedFor(Visitable vis)` method is necessary for the component to choose the appropriate wrapper class for the *Visitable* object in the argument. Given an object `vis` of class *Visitable*, this method determines the correct *Visited* wrapper for the object, depending whether it is of class *ClassA* or *ClassB*, resorting to the `instanceof` clause. This method must be placed at the top level because it does not belong to any of the virtual classes, but must rather choose the appropriate wrapper between. The `VisitedFor(Visitable vis)` method has been implemented to overcome a limitation in CaesarJ. Although described in [6], the mechanism of dynamic wrapper selection was never implemented. This method programmatically emulates that mechanism. Currently, the CaesarJ developers are studying ways to implement this functionality in the language [24].

The *Visitor* class declares the extra behavior to be added to the classes in the application. Each different class must have its own visit method, represented by the `visit(ClassA v)` and `visit(ClassB v)` methods.

The *ConcreteVisitor* module contains the virtual classes that implement the different concrete visitors and their additional behavior to classes in the application. The virtual

classes are illustrated by the *ConcreteVisitor1* class, and the methods that add behavior to classes in the application by methods `visit(ClassA v)` and `visit(ClassB v)`. Each different implementation of these methods adds extra behavior to objects in *ClassA* and *ClassB*, respectively. This implementation of the pattern has an impact on the client code in three manners:

- A family object of type *ConcreteVisitor* must be created.
- Individual visitors must be created through the *ConcreteVisitor* object.
- The wrappers for objects of the application domain must be selected by calling the `VisitedFor(Visitable vis)` method.
- After the visitors and the wrappers for the objects in the application domain are created, the `accept(Visitor v)` performs the operation to be added to the objects in the application domain.

5.12 Summary

A total of thirty CaesarJ design pattern implementations were developed from the existing Java design patterns. These implementations are spread through different scenarios from different repositories. Table 6 lists the design pattern implementation distribution, specifying the number of implementations per pattern, and the original repository from which the scenario was taken.

	Thinking in patterns	**DP Java companion**	**Fluffycat**	**Hannemann et al.**	**Huston**	**Guidi Polanco**
Abstract Factory	X			X		
Bridge	X			X		
Builder	X			X		
Chain of Responsibility		X				X
Composite	X	X		X	XX	
Decorator	X		X			X
Factory Method				X		X
Mediator		X	X	X		
Observer			X	X		X
Prototype		X	X	X		
Visitor		X	X		X	

Table 6 CaesarJ design pattern implementations by repository

This work has also used the implementations previously developed by Sousa et al. [48]. However, some of the modules provided as a basis were subjected to design modifications. Table 7 describes the cases where the pattern modules suffered changes (Yes) and where the module design remained the same (no). This table lists only the patterns in common between this dissertation and the studies of Sousa et al. [47][48].

	CI	**CJImpl**	**CJBinding**
Abstract Factory	n/a	n/a	no
Bridge	no	no	no
Chain of Responsibility	Yes	Yes	no
Decorator	n/a	n/a	Yes
Observer	no	Yes	Yes
Visitor	no	n/a	Yes

Table 7 Modified CaesarJ modules

An additional six completely new pattern implementations have been created in the context of this dissertation for analysis purposes, making the total number of CaesarJ design pattern implementations rises to 36 implemented patterns. Chapter 6 describes the analysis performed on the 30 patterns implemented from existing Java repositories, while section 6.4 describes the need for further design pattern implementations and the analysis that ensued.

6. Analysis

This chapter discusses the design pattern implementations in Chapter 5 as a basis for considerations on CaesarJ's support for reuse and draws a comparison with the AspectJ implementations by Hannemann and Kiczales [28]. It also suggests some directives on programming with CaesarJ, based on the experience gained with the implementation of the design patterns and the underlying study of the language.

The rest of this chapter is structured as follows. Section 6.1 introduces the criteria under which the patterns are evaluated; section 6.2 analyzes the patterns as far as the criteria for language mechanisms and reusability levels; section 6.4 presents the results of scenarios developed to further assess the composition capabilities of six selected patterns; section 6.5 draws a general comparison between the CaesarJ and AspectJ languages and section 6.6 gives some general guidelines on the design of CaesarJ components.

6.1 Assessment criteria

This study uses a set of qualitative criteria to assess the attributes of the pattern implementation. These criteria evaluate the pattern implementations as far as their language mechanisms, reuse and composition.

Table 8 is used to present the criteria for this analysis and brief description of each criterion.

The *language mechanism* criteria review the language constructs and modules used in the pattern implementations. Their intent is to portray the support for reuse provided by CaesarJ as reflected by the implementations' constructs and modules.

a. *Pointcut/advice utilization* criterion reports whether it was necessary to use these mechanisms. This criterion reflects whether CaesarJ is able to cope with crosscutting behavior using other mechanism besides pointcuts and advices.

b. *Component modules used* criterion lists the CaesarJ modules used in the pattern implementations.

Properties	Criteria	Description
Language mechanisms	Pointcut/advice utilization	Pointcuts and advice are used to implement the pattern
	Component modules used	The CaesarJ modules used to implement the pattern
Reuse level assesment	Reuse level	Reuse level resulted in the pattern implementation
	Same module, different scenarios	Possibility of a given component to be composed in multiple scenarios
	Same module, same scenario, several instances	Different instances of the same pattern can be composed in the same scenario
	Same module, same scenario, different implementations	Possibility of multiple implementations of a component to co-exist in the same application having different pattern implementations
Composition ability	Ability to discriminate between instances of a class	Possibility of a component to compose to just a selected subset of the existing instances of a given class.
	Observable composition order	The order by which the pattern is composed to the application produces different results.
	(Un)pluggability	The pattern can be easily removed or added to a system, maintaining a functional system.

Table 8 Assessment criteria description

The *reuse level assessment* criteria evaluate the extent to which the modules resulting from the pattern implementation are reusable. Different levels are considered, reflecting how reusable the module is.

a. *The Reuse level* criterion establishes different levels of reuse according to the modules used in the pattern implementations. It differentiates between 4 distinct levels of reuse, listed in descending order:

1. *Direct language support* – CaesarJ language constructs provided direct support for the pattern implementation. This criterion is considered the most favorable level for reuse because the pattern is inherent to the language. As a consequence, no reusable modules are necessary.

2. *Reusable modules* – This level of reuse reflects the generalization of the pattern code into a module with reusable code. This is considered the second most favorable level of reuse because the pattern logic is modularized into a reusable module.

3. *Composition flexibility* – This criterion reflects that, although no reusable modules were achieved, the pattern implementations still present enhanced composition

abilities. This means that the pattern logic could not be abstracted into a reusable module but the pattern implementation can be easily composed with classes in the base application domain. It is considered the third most favorable level of reuse.

4. *No reuse* – This criterion reflects that neither of the above advantages could be accomplished. It is considered the forth and lower level of reuse.

If the patterns have originated reusable modules, an additional assessment can be made.

a. The *Same module, different scenarios* criterion determines if the pattern implementation can be used in different scenarios. If a reusable pattern implementation is obtained, it must be able to be applied to different scenarios. This criterion is the most basic level of reuse for reusable modules.

b. The *Same module, same scenario, several instances* criterion tells if more than one aspect instance can be used in one scenario without the aspect instances interfering with each other. If several autonomous aspect instances can be created and composed within the same scenario, pattern management can be dealt in a flexible manner. This criterion corresponds to an intermediate level of reuse.

c. The *Same module, same scenario, different implementations* criterion judges if it possible to have several autonomous aspect instances in the same scenario, functioning with different implementations. If this criterion is positive, it illustrates the possibility to compose different aspect instances with different functionality in the same scenario. If so, it is possible to select which aspect to compose with particular, depending on the desired functionality. This is considered the highest level of reuse.

The composition criteria assess the composition characteristics of the pattern implementations that have achieved produced reusable modules or modules with composition flexibility.

a. The *Ability to discriminate between instances of a class* criterion assesses how far the component can go in managing individual instances. If this criterion is positive, the aspect can determine which objects can perform the roles defined in the pattern. This allows the object level definition of the pattern participants.

b. The *Observable composition order* criterion tells if the order by which the participants are composed to the pattern has any influence on the pattern

functionality. If it does, the pattern should present different results according to the composition order.

c. The *(Un)pluggability* criterion states whether the pattern can be easily removed or added to the base application. Furthermore, the removal of the pattern from the application must not imply that it will not function or not make sense.

6.2 Mechanism usage

The CaesarJ pattern implementations invite analysis between the different mechanisms CaesarJ uses to cope with crosscutting concerns present in the 11 approached patterns. This analysis can serve as the basis for comparisons with AspectJ.

6.2.1 Pointcut and advice

An immediate comparison can be made between the pointcut/advice mechanism use in the two languages. Table 9 summarizes the use of pointcuts and advices in every pattern.

Use of pointcut/advice:	**CaesarJ**	**AspectJ**
Abstract Factory	No	No
Bridge	No	No
Builder	No	No
Chain of Responsibility	Yes	Yes
Composite	No	No
Decorator	No	Yes
Factory Method	No	Yes
Mediator	Yes	Yes
Observer	Yes	Yes
Prototype	No	No
Visitor	No	No

Table 9 Pointcut and advice use in CaesarJ and AspectJ GoF implementations

Of the 11 CaesarJ pattern implementations, 3 patterns make use of the pointcut and advice mechanism, against 5 of the AspectJ implementations. The 3 patterns where both languages make use of this mechanism are *Chain of Responsibility*, *Mediator* and *Observer*. The two patterns where AspectJ uses pointcuts and CaesarJ does not are *Decorator* and *Factory Method*. These patterns share a common factor: they are not concerned with the dynamic behavior of objects. *Chain of Responsibility*, *Mediator* and *Observer* are indeed labeled as behavioral by Gamma et al [21], and the use of pointcuts seems adequate to

capture events of interest that trigger actions through advices. However, *Decorator* and *Factory Method* are not the typical case that lends itself to the use of pointcuts, such as scattered method calls. These patterns deal with specific situations: additional functionality attached to an object by enclosing it in another object (*Decorator*) and the appropriate instantiation of objects that will ultimately compose families of related objects (*Factory Method*). CaesarJ provides other language mechanisms that replace the use of pointcuts and advices.

For *Decorator*, CaesarJ is able to use the wrapper mechanism. CaesarJ's wrapper mechanism has a close relation to the intent of the *Decorator* pattern, and this has proven a more flexible way to compose objects with *Decorators*. With the wrapper mechanism, it is possible to decorate the same object with the same object several times (as it is in AspectJ), but it is also possible to decorate the same object with several different decorators, in an arbitrary order. This proves to be an advantage relatively to the mechanisms used by AspectJ, where it is necessary to declare precedence between aspects. Furthermore, in CaesarJ the same object can be composed multiple times with the *same Decorator*.

For *Factory Method*, CaesarJ uses the virtual class mechanism and implicit inheritance to allow for the polymorphic instantiation of classes. If virtual classes are declared as concrete they can be polymorphically instantiated because the created object will depend on its enclosing family class. This provides direct language support for the pattern. AspectJ makes use of the pointcut and advice mechanisms to intercept the calls to each class' factory method and define different implementations.

6.2.2 CaesarJ modules

The different pattern implementations have resulted in a diverse use of the available CaesarJ modules. Table 10 lists the modules that have resulted from the pattern implementations. As portrayed in chapter 3, the different CaesarJ modules present varied reuse nature.

	CI	CJImpl	CJBinding w/ wrappers	CJBinding	Weavelet
Abstract Factory	No	No	No	Yes	No
Bridge	Yes	Yes	No	Yes	Yes
Builder	No	No	No	Yes	No
Composite	Yes	Yes	Yes	No	Yes
CoR	Yes	Yes	Yes	No	Yes
Decorator	No	No	Yes	No	No
Factory Method	No	No	No	Yes	No
Mediator	Yes	Yes	Yes	No	Yes
Observer	Yes	Yes	Yes	No	Yes
Prototype	Yes	Yes	Yes	No	Yes
Visitor	Yes	No	Yes	No	No

Table 10 CaesarJ module usage in pattern implementation

6.3 Reuse level

As it could be expected, not all GoF patterns yielded modules with the same level of reusability. This is a result of the nature of the patterns themselves, but also of the mechanisms CaesarJ provides to developers for the separation of concerns into reusable modules. Table 11 describes the CaesarJ level of reuse obtained in the implementation of the 11 design patterns.

	Direct support	Reusable module	Composition flexibility	No reuse
Abstract Factory	X	-	-	-
Bridge	-	X	-	-
Builder	-	-	-	X
Composite	-	X	-	-
CoR	-	X	-	-
Decorator	-	-	X	-
Factory Method	X	-	-	-
Mediator	-	X	-	-
Observer	-	X	-	-
Prototype	-	X	-	-
Visitor	-	-	X	-

Table 11 CaesarJ support for reusability

The criteria shown in Table 11 provide an approximate overview of the level of reuse that was obtained in the patterns. Direct language support is considered the highest level of reuse. If the pattern has direct language support, it does not produce any reusable modules because

the language itself has mechanisms that serve the purpose of the pattern. It is worth mentioning that direct language support can be mistaken for lack of reuse, because no reusable modules are produced, but it is not the case. Reusable modules can be produced to overcome a shortcoming in a programming language. If it directly supports a design pattern, there is no need to create a reusable module.

If the pattern does not support the pattern directly, it is necessary to distinguish between the modules produced. If the only produced CaesarJ module with a reusable nature is a CI, only the general component design information has been captured in a reusable manner. The logic of the pattern is described in the structure of the CI but it is not a functional module by itself. There must be a CJBinding, or preferably a combination of CJImpl and CJBinding to compose a concrete module.

If a CJImpl is obtained, it is therefore possible to have a pattern with alternative implementations. This allows the developer to choose among a set of options for how the pattern will function in a flexible manner. The CJImpl reflects that it was possible to remove the pattern logic from classes in the domain application into a localized and context independent module. It also enables the pattern specific code to be composed into several scenarios, by composing it with CJBindings. If a pattern implementation is not directly supported by a CaesarJ language mechanism, but has originated a CJImpl, it is placed in the *Reusable modules* level of reuse.

CJBindings are always present in every scenario of every pattern, being the CaesarJ module that reflects the context specific module of the pattern. Nevertheless, it is possible to make a distinction between CJBindings that wrap classes in the domain application and ones that must completely move the classes from the domain application into the CJBinding. CJBindings with wrappers are more flexible to compose to existing applications. Wrappers are also less intrusive than placing the code in a CaesarJ module. Sometimes the code might not even be available to developers. Therefore, if a pattern implementation has not originated a CJImpl, but it originated a CJBinding with wrappers it is placed in the *Composition flexibility* level of reuse.

If neither a CJImpl nor a CJBinding have been produced, it is considered that the pattern implementation provided no reuse and is therefore placed in the *No reuse* level.

6.3.1 Direct language support

From Table 11 we see that 2 of the 11 patterns are directly supported by CaesarJ, *Abstract Factory* and *Factory Method*. In *Abstract Factory*, CaesarJ's implementation of family polymorphism through virtual classes directly supports the patterns. As discussed in Section 5.1, a top level class that comprises several virtual classes acts as unit of confinement. This top level class sets up the group of classes that are related with each other, preventing unrelated classes to mix. Classes can be grouped by putting them inside the same top level class. The virtual class mechanism enables classes of the same family to be refined in sub-classes while still maintaining family consistency. This assures both type safety and flexibility. The downside to this approach is that classes have to be removed from the application domain into a CaesarJ module. *Factory Method* makes use of the virtual class and implicit inheritance mechanisms to produce polymorphic constructors as mentioned in Section 6.2.1. Polymorphic constructors solve the problem addressed by *Factory Method*, by enabling different classes to be instantiated by the same constructor call, without losing control over the exact concrete type of the object created. The created object is defined by its family class. To use polymorphic constructors it is first necessary to create an object of the desired family class, a family object. This family object will define the context of the virtual class created, hence allowing control over the object created by the polymorphic constructor.

6.3.2 Reusable modules

The following 6 patterns have originated an implementation with reusable modules: *Bridge*, *Composite*, *Chain of Responsibility*, *Mediator*, *Observer* and *Prototype*. Still, these patterns can be divided into 2 smaller sets.

In a higher level of reuse, a set of 5 patterns has resulted in pattern implementation separated into CIs, CJImpls and CJBindings modules with wrappers. These patterns are *Composite*, *Chain of Responsibility*, *Mediator*, *Observer* and *Prototype*. These implementations have allowed the removal of pattern specific code from classes in the application into easily composable CaesarJ modules. The modules are straightforward to compose thanks to the wrapper mechanism. Wrappers attach roles in the pattern to specific

instances of desired classes in the application domain. This enhances composition flexibility because wrappers function at object level, rather than of the class level. The virtual classes in the CJBindings declare they wrap classes in the application, but the wrapper instantiation mechanism selects the desired object to which to compose the patterns. The wrapper recycling mechanism also maintains mappings between each wrapper and the corresponding object in the application. This way, wrapper objects of the pattern logic are uniquely identified by objects in the application domain. The CJImpls allow the deployment of different pattern implementations, allowing developers to choose between a number of alternative ways to implement the context independent part of the component of the pattern. Generalizing the context independent part of the component into a separate module from the context specific allows for code locality and separation of concerns, which leads to reusability. Within this group, *Mediator* constitutes a particular case where full separation of concerns was not achieved. As mentioned in Section 5.8, the role of *Mediator* was not fully removed from classes in the application domain. This is contrary to the findings of Hannemann et al. [28]. Hannemann et al. argue that the role of *Mediator* is superimposed, which proved not to be the case in the studied scenarios. In [28], the *Mediator* role is actually scattered through 2 classes, Main and Label, and it is the Main class that holds static references to the objects playing the roles of *Colleague*. Since Label holds no references to *Colleague* objects or methods to manage these references, it only holds the notification logic. It then resorts to conditional tests to check which static reference in the Main class triggered the notification operation to establish which *Colleague* should be informed.

The *Bridge* pattern originated a CI, a CJImpl, and a CJBinding but that CJBinding module does not declare any wrappers. This makes it necessary to place all the code in a CaesarJ module. The *Bridge* pattern presents the flexibility CJImpls bring to pattern implementation, but the disadvantages of placing the pattern code entirely in a CaesarJ module.

6.3.3 Composition flexibility

The *Decorator* and *Visitor* patterns can be placed in the following level of reuse. Although these patterns did not result in the implementation of CJImpls, they still benefit

from the composition advantages brought by the wrapper mechanism. *Decorator* and *Visitor* differ because the latter has originated a CI. The CI adds the benefits of design information to the pattern. The formation of a CI is useful because the two roles, *Visited* and *Visitor*, can be abstracted into a higher level in the pattern, as well as an operation that can be placed into one of the roles, as described in Section 5.11. The *Decorator* pattern places only one role in a CaesarJ module, a *Decorator* that is composed with *Component* objects in the application. Since the operations a *Decorator* performs depend solely on the *Components* they decorate, no operations can be placed in a CI. The benefits mentioned for the wrapper mechanism for the previous set of patterns still apply to this group, but the benefits of deriving a CJImpl do not. That is why this group should be placed in a lower level of reuse.

6.3.4 No reuse

Finally, *Builder* can be considered the pattern that has presented the worst reusability results, since no reusable modules were produced and neither was a CJBinding with wrappers. The advantage that can be recognized for the *Builder* pattern implementation in CaesarJ is improved type safety. By declaring a *BuilderFamily* class, family polymorphism prevents unrelated *ConcreteBuilder* and *ConcreteResults* to be mixed. See Section 5.3 for illustration. This advantage can nevertheless be found in all CaesarJ modules that define virtual classes with the corresponding family class.

6.4 Pattern composition capabilities

As could be expected, not all CaesarJ pattern implementations proved reusable to the same degree. This results either from the intrinsic nature of the pattern but also from the mechanisms provided by CaesarJ for supporting modularity of aspects and their consequent composition with the specific application. Among the 11 developed patterns, 6 have been selected for an in-depth analysis of the possibilities CaesarJ provides for composing its independently developed modules with modules previously developed in other applications. The 6 patterns are:

- *Chain of Responsibility*
- *Composite*
- *Decorator*
- *Mediator*
- *Observer*
- *Visitor*

The analysis from this section is focused on CaesarJ's ability to compose independently developed modules, so the 6 patterns consist of the patterns that use the wrapper mechanism to compose themselves to classes of existing applications, with the exception of *Prototype*. The reason for the exclusion of *Prototype* is that its CI only has a virtual class, and its composition mechanism is straight forward because the pattern does not imply the interaction of objects playing different roles. Wrappers exist in *Prototype* solely to add the cloning operation to individual objects in the application.

The 6 patterns included in this analysis can be divided in two different sets: patterns that have originated reusable modules with the possibility for different implementations (*Chain of Responsibility*, *Composite*, *Mediator* and *Observer*), and patterns that have not originated reusable modules (*Decorator* and *Visitor*).

For this evaluation, new scenarios were developed for each of the patterns. The aim was to test each pattern implementation's level of reusability. Together with the implementations from previously existing Java repositories, the total number of CaesarJ design pattern implementations ascends to 36 design pattern implementations.

Table 12 is used to describe the properties of the CaesarJ modules that have originated CJImpls, i.e., modules of the first set.

	Same module, different scenarios	**Same module, same scenario, several instances**	**Same module, same scenario, different implementations**
Composite	Yes	Yes	Yes
CoR	Yes	Yes	Does not apply
Mediator	Yes	Yes	Does not apply
Observer	Yes	Yes	Yes

Table 12 Reusable modules implementation properties

The columns from Table 12 correspond, from left to right, to increasing levels of reusability.

The second column from Table 12 indicates whether the reusable module can be composed with several scenarios. This criterion always yields a positive result, as it corresponds to the minimum level from which a module can be considered reusable.

The third column from Table 12 indicates whether it is possible to create several instances of the same modules in the same scenario. This criterion is also always positive due to CaesarJ's aspect instantiation mechanism. CaesarJ enables a user to create an arbitrary number of aspect instances in typical object oriented fashion. The tests have proven that every aspect instance is fully autonomous. Consequently, creating several aspect instances in the same scenario does not interfere with each instance's execution.

The forth column from Table 12 indicates if it is possible to create multiple aspect instances in the same scenario, but with each aspect instance comprising different CJImpls modules. These instances share a common interface but have different functioning. The experiments have proven that this is true for the *Composite* and *Mediator* patterns. It has not been possible to apply this criterion to the *Chain of Responsibility* and *Mediator* because only one CJImpl has been developed. In the patterns where it was possible to test this criterion, all cases yielded a positive result. These results are due to two factors: loose coupling between modules and mixin composition. CaesarJ effectively enables loose coupling between its modules, which results in enhanced flexibility when composing modules into a full aspect component with mixin composition. Mixin composition allows the abstract and concrete facets of an aspect to be implemented in clearly separated modules and then combined into one module that corresponds to the developers needs.

In order to evaluate CaesarJ capability for composing independently developed aspects with the domain application classes, the pattern implementations were evaluated under different criteria.

Table 13 shows the results the pattern implementations have displayed for these composition criteria.

The first column indicates whether the modules are able to distinguish between instances of the same class. This is true for all cases due to CaesarJ's wrapper mechanism. Each wrapper declaration creates a unique relation between an object in the application, the wrappee object, and the object performing a role in the pattern, the wrapper object. This way,

	Ability to discriminate between instances of a class	Observable composition order	(Un)pluggability
Composite	Yes	Yes	Yes
CoR	Yes	Yes	Yes
Decorator	Yes	Yes	Yes
Mediator	Yes	Does not apply	Yes
Observer	Yes	Yes	Yes
Visitor	Yes	Does not apply	Yes

Table 13 Reusable modules composition properties

wrapper objects establish a mapping between objects in the application context and roles in the context of the aspect. These wrapper objects are dynamic extensions to the objects in the application domain can be treated individually as regular Java objects.

The second column indicates whether the module composition order has any impact on the aspect behavior. This is true for all cases except *Mediator* and *Visitor*, where the criterion does not apply. *Visitor* aims to add operations to all instances of a class. Therefore, this criterion is not applicable. In the case of *Mediator*, the application of the criterion depends on the notifying logic of the *Mediator* role. If the *Mediator* notifies its *Colleagues* by traversing a data structure, the composition order is observable. If it holds references to its *Colleagues* by keeping data members, then the composition order is not observable. In the *Composite* pattern, the composition order is reflected in the children nodes of objects performing the role of *Composite*. In the case of the *Chain of Responsibility* pattern the composition order is reflected in the order of the chain of responsibility. The *Decorator* pattern exhibits the most clear observable effect of the composition order. The *Decorator* module enables an object in the application to be decorated with several *Decorators*. However, if an object `Obj` is decorated with two decorators `A` and `B`, the composition order of the decorators defines two distinct results. Finally, the composition order can be seen in the *Observer* pattern in the notification logic of the *Subject* role. The order by which *Subjects* add *Observers* is reflected in the order by which *Observers* are notified of changes in the *Subject*'s state.

The third column indicates whether the application will still be functional if the CaesarJ pattern module is removed from the system and the participants in the pattern have some meaning outside the pattern implementation. In all patterns, the CaesarJ module can be easily removed because the pattern-specific code has been completely removed from the domain application. This way, the pattern can be composed to instances of the classes of the application while they still maintain their responsibilities outside the pattern.

6.5 Reuse comparison with AspectJ

This section summarizes a comparison between the support for reuse given by CaesarJ and AspectJ. Section 6.5.1 discusses the modules that have originated reusable modules in both languages, while section 6.5.2 discusses the subject of reusability in the two languages in from a more general point of view.

6.5.1 Reusable modules comparison

Due to the pattern implementations resulting from this study, and the analogue implementations in the Hannemann and Kiczales study [28] it is possible to draw a direct comparison between the patterns that have resulted in a reusable module. Hannemann and Kiczales have obtained reusable modules in the form of AspectJ abstract modules. The CaesarJ patterns that have originated reusable modules are the pattern implementations that have been created resorting to language mechanisms in CaesarJ that provide direct language support and patterns that derived CJImpls (see section 6.3). Table 14 summarizes this comparison.

	Reusable modules	
Pattern name	**CaesarJ**	**AspectJ**
Abstract Factory	D.L.S.*	No
Bridge	Yes	No
Builder	No	No
Chain of Responsibility	Yes	Yes
Composite	Yes	Yes
Decorator	No	No
Factory Method	D.L.S.*	No
Mediator	Yes	Yes
Observer	Yes	Yes
Prototype	Yes	Yes
Visitor	No	Yes

Table 14 Reusable module comparison between CaesarJ and AspectJ

* D.L.S. – Direct Language Support

A total of 8 CaesarJ design pattern implementations have resulted in a reusable module, including patterns with direct language support, while the analog AspectJ implementations have originated 6 reusable modules. The results for the pattern implementation largely match. The differences occur in the *Bridge* pattern, where it was possible to derive a reusable CJImpl and AspectJ was not able to produce a reusable abstract aspect, in the *Abstract Factory* and *Factory Method* patterns, where CaesarJ provided direct language support for the pattern implementation and for *Visitor* where AspectJ was able to produce a reusable abstract aspect and CaesarJ was not able to produce a reusable module.

6.5.2 General comparison

Except for the case of the *Visitor* pattern, CaesarJ has obtained similar results to AspectJ as far as the number of reusable modules. Except for *Visitor*, all patterns that originated reusable modules in AspectJ did so with CaesarJ. Nevertheless, the analysis in Sections 6.1 and 6.2 have established differences in the level of reuse among the 2 languages. The highest level of reusable modules in AspectJ corresponds to developing abstract and concrete aspects. In CaesarJ it corresponds to developing modules with CI, CJImpls and CJBindings with wrappers. The advantage this brings is that it is possible to have several alternative implementation strategies for each pattern. Mixin composition allows developers to choose the desired implementation strategy for the pattern, compose it to the CJBinding for the concrete scenario and derive a concrete aspect component. AspectJ does not allow this level of flexibility.

Another difference between CaesarJ and AspectJ is their module's internal structure. In AspectJ, pattern aspects present a flat internal structure, where interfaces, methods and data structures are at the same level. With virtual classes, CaesarJ offers a richer internal structure to aspects, clearly defining role responsibilities between the virtual classes declared within the aspect module. These classes are able to represent the roles involved in the pattern, but can also hold methods and data structures that are related to them. This approach is closer to object oriented languages, where the logic associated with a concept is enclosed by the class that modules that same concept. This structural difference constitutes a basic advantage to

CaesarJ because it enables dealing with the roles associated with patterns and the operations they must perform in a more intuitive manner.

Since CaesarJ allows for the explicit instantiation of aspects, aspects can be managed as objects with additional constructs. This shortens the conceptual gap between aspects and classes. Also, it makes for a more natural control over aspect deployment and composition. Since several instances can be created, this corresponds to several aspect components functioning in the same scenario. The deployment scope of these scenarios can also be explicitly controlled. When an aspect instance is created, it must still be deployed before it is effective. CaesarJ has mechanism to dynamically deploy and undeploy aspect instances, allowing developers to control several aspect instances' scope in an intuitive manner. Another advantage of creating aspect instances is that it allows different instances to compose themselves to selected objects in the application domain. The composition of aspects to objects in the application domain is carried out by wrappers.

A further advantage of the wrapper mechanism was observed in the implementation of the *Prototype* design pattern. The CaesarJ pattern implementation can make use of the Java API marker interface `Cloneable`. This marker interface allows classes to use the `clone` method to produce copies of its instances. CaesarJ implements the cloning operation in the CJImpl and glues the pattern implementation to classes in the application through CJBindings with wrappers. This approach removes the need for classes in the application domain to use the `Cloneable` marker interface, becoming oblivious of the role they play in the pattern. AspectJ is able to produce a reusable abstract aspect that implements the cloning operation however the classes in the domain application must still declare they implement `Cloneable`.

Nevertheless, the *Visitor* pattern revealed some limitations to CaesarJ's wrapper mechanism. *Visitor* exposes the limitations of CaesarJ's wrapper mechanism when dealing with inheritance hierarchies in classes of the application domain. Since CaesarJ does not allow classes with wrapper declarations to be refined in sub-classes that declare different wrappers, CaesarJ lacks a mechanism to integrate with inheritance hierarchies polymorphically. The developer is forced to declare different wrappers for subclasses of already wrapped classes. This lack of subtype polymorphism defeats the double dispatch intent of the *Visitor* pattern. It is necessary to programmatically enforce mechanisms to deals

with the selection of the appropriate wrapper for the class, in the base application. The `VisitedFor` method is a direct consequence of this need. See Section 5.11 for the CaesarJ implementation of the *Visitor* pattern. In comparison CaesarJ, the intertype declaration mechanism of AspectJ yields better results. AspectJ uses intertype declarations to introduce marker interfaces that assign the roles of the pattern to classes in the base application. The difference is that the aspect is able to hold the inheritance hierarchy between the marker interfaces. Since the marker interfaces keep their inheritance hierarchies, AspectJ is able to remove the pattern specific code into an aspect and still allow for double dispatch.

6.6 CaesarJ component design guidelines

This section presents some guidelines for the design of CaesarJ components. The following considerations derive from the experience gained in the context of this dissertation. Nevertheless, these guidelines do not aim to be strict rules for the refactoring of Java code into CaesarJ. Such studies would presume deeper research on this subject and formal description of refactoring processes [40]. However, the CaesarJ implementations developed during this dissertation and their Java equivalents can serve as code examples for such future studies.

When design a CaesarJ component, it is first advisable to consider the roles involved in the component. Components can sometimes deal with several participant classes. These classes should be generalized into abstract virtual classes that model functional roles in a CaesarJ component. Each role is in turn responsible for specific operations it must carry out in the context of the functioning of the component. Each of these operations should be placed in the corresponding virtual class. Together, the description of the roles that abstract participant classes and the operations these roles must carry out form the interface of the pattern. Therefore, the constructs should be placed into a CI because they describe the component through abstract classes and roles, but do not implement any.

In the functionalities a component adds to a system, components should be able to distinguish between functionalities that can be implemented independently of the system where the component is deployed or functionalities that directly depend on classes in the base system. These are normally seen as the provided and expected facets of a component. The

provided facet comprises the functionalities that the component adds to the base system and the expected facet is the functionalities that are dependent on the classes in base system in order to be implemented. CaesarJ supports the separation of these two facets with the CJImpl and CJBinding modules.

The key to deriving reusable modules in the form of a CJImpl is that it must not reference classes in the base system, as that leads to tight coupling to a concrete system. If that is the case, that module should be considered a CJBinding, as it is strongly context dependent. To keep CJImpls context independent, they should refer only to abstractions described in the CI in the implementation of the provided facet. This way, the functionalities of the provided facet can be implemented resorting solely to abstractions contained in the component, which can be considered higher level abstractions of the participant classes that take part in the component. Thanks to the CI, the CJImpl is able to use the functionalities of the expected facet without knowing their specific implementation. This loose coupling provided by the CI is paramount to the development of alternative and reusable CJImpl modules.

Since the provided facet implemented in the CJImpl modules resorts to the functionalities implemented in the CJBindings, the CJBindings must be able to correctly map the operations of the classes in the base system to the abstract operations of the collaborating roles described in the CI. Wrapper classes are able to incorporate objects of the classes in the base system and translate them into the abstractions defined in the CI, and accessing their wrappee's methods. Wrappers present advantages over moving a class of the base system into a CaesarJ module because different wrappers can be created to wrap the same class multiple times. This can be useful if the same class can perform different roles in the component context or variations of the same role. CJBindings can be seen as specialized classes that make possible the transition between the context of the base system and the context of the component, therefore enabling CJImpls to remain oblivious of the base system implementation details.

If a component must react to specific events of a general nature, typically scattered through different classes, CJBindings should define pointcuts to define which events the component must react to and advices to detail which operations should be triggered. Combining wrappers and advices allows different pointcuts to trigger different events in a

flexible way, where an operation performed by a single class can trigger actions on several different wrappers, depending of the pointcut.

Finally, the family polymorphism mechanism provides additional expressiveness and safety for the definition of interactions between related implementations of the participants of a component. Family classes that extend the CI can have multiple refinements of the abstract roles defined in the CI. However, not all of these refinements may be compatible with each other. Therefore, classes that define refinements of the abstract roles of the CI should be placed in a common family class, while classes that are not compatible should be placed in different family classes.

Weavelets comprise the complete realization of the pattern and are put together through mixin composition. A small detail must be kept in mind when defining the order of the mixin. Mixins define superclasses in a serialized order, which means that the modules that implement the most context specific methods must be declared first. This detail is revealed when CJImpls and CJBindings contain overlapping implementations of the same method declared in the CI. Such was the case in the *Mediator* pattern, where the CJBinding implemented a method that was also implemented in the CJImpl. Because the CJBinding is more closely related to the pattern, the mixin composition order reflected this conflict. See section 5.8 for further detail.

7. Related Work

The work related to this thesis can be placed in 3 different categories: AOP implementation of the GoF design patterns, the evaluation of these implementations and the appearance of aspect-oriented design patterns that has come from the increasing experience of programming with aspect-oriented languages. Section 7.1 details other AOP implementations of the GoF design patterns, section 7.2 describes methodologies for the evaluation of AOP implementations of GoF design patterns and section 7.3 presents some AOP design patterns that have been suggested as the use of AOP languages has become more widespread.

7.1 AOP implementation of GoF design patterns

Nordberg has elaborated on the potential of AOP to significantly reshape or even make obsolete many common object-oriented design patterns [43]. According to Nordberg, object-oriented design patterns anticipate change at the price of extra overhead for object-oriented indirection. This overhead can be reduced by introducing aspect-oriented design patterns with better designs. Nordberg's study also presents an AspectJ implementation of the *Factory Method* pattern.

Rajan has provided a case study of implementation of the GoF design patterns in the Eos AOP language [45]. Unfortunately, the source code that has resulted from this case study is not freely available. The drive behind this study is the concept that the notions of aspect and class can be unified in a new module. The Eos language supports this concept in the form of the *classpect* module construct. The author has taken the AOP design pattern implementations in [28] and created equivalent implementations in Eos, for comparison purposes. These comparisons were based on the modularity qualitative criteria used by Hannemann et al. but also on two metrics, the number of lines of code used in the aspect and if the implementation keeps a *Close Match to Pattern Intent (CMPI)*. The author concluded

that 7 pattern implementations showed improvement over the AspectJ implementations and the remaining 16 patterns showed no worse results. Our work shares the intent of comparing 2 different AOP languages based on design pattern implementation by developing implementations of independently developed design patterns. However, Rajan's study is based in Hannemann et al. AspectJ implementation where ours is based in several Java implementations. Furthermore, our study does not contemplate the metrics used by Rajan, focusing instead in characterizing CaesarJ's composition abilities. Another resemblance between Rajan's study and ours is the concept of unifying classes and aspects. Although this concept is shared by both languages, CaesarJ still separates the method and advice constructs in an AspectJ-like manner, whereas Eos unifies the constructs of methods and advices. Furthermore, CaesarJ presents the virtual class mechanism to establish structural collaborations between classes of related families and Eos does not. Finally, Eos' underlying language is C# while CaesarJ is an extension to Java.

Hachani et al. also recognized that objected-oriented implementation of design patterns could be improved by aspect-oriented technologies [27]. This study lists a set of 4 problems associated with the objected-oriented design approach, namely *Confusion*, *Indirection*, *Encapsulation Breaching* and *Inheritance Related* problems as particular cases of code-scattering and code-tangling. The authors take the *Visitor* pattern as an example of a design pattern that could be improved using AOP and offer an alternative AspectJ implementation for this pattern. This study has served as motivation for another work by Hachani et al. [26] where the same 4 problems are addressed and an implementation of the *Strategy* pattern is presented. The study argues that not only do design patterns gain from aspect-oriented implementation but also that the aspect-oriented pattern implementation should be complete with aspect-oriented description, similar to the descriptions in [21], so that pattern description also benefits with easier documentation evolution. This study mentions the implementation of the 23 GoF design patterns, but presents no evaluation besides mentioning benefits in code locality and pattern traceability. The AspectJ implementation of the design patterns can be found in [4]. The same page also holds a HyperJ implementation of the GoF design patterns, but mentions no subsequent studies.

Hirschfeld et al. tackle the question of design pattern implementation using the aspect oriented language AspectS [29]. These authors state that object-oriented design pattern can

be enhanced by aspect-oriented representation, but mainly from a native AOP approach to design patterns, improving design pattern solutions both in development time and at run-time. The authors discuss the need for explicit variation points in order to allow the development of system parts independently and later join them together to form the desired system with no performance degradation. The authors characterize the parts of a system as the fixed and variable parts as well as the glue code that binds the two. While AOP representations of design pattern effectively improves the separation of the fixed and variable parts of a system and removes the need for glue code in the fixed part, the weaving process necessary to compose both parts results in performance degradation because the end system run-time behavior is hindered by messaging overhead caused by indirection levels and context-dependent change of identity. The authors defend that a native AOP approach can provide support for the separation of fixed and variable parts of a system but also to seamlessly combine the two parts at run-time, eliminating glue code and performance issues like messaging overhead. The *Visitor* and *Decorator* design patterns are used to illustrate this approach.

7.2 AOP implementation evaluation

Garcia et al. have produced an exhaustive study in the quantification of modularity improvements in the AspectJ implementations of the GoF design patterns [22]. This group of authors has established a quantitative study that compares the Java and AspectJ solutions for the 23 GoF patterns presented in [28] to claim that most aspect-oriented showed improvement in the separation of pattern-related concerns but only the aspect-oriented implementations for *Composite*, *Mediator*, *Observer* and *Visitor* exhibited significant reuse. The authors replicated the study described in [28] but with a larger number of participant classes to perform pattern roles which is justified by the authors by the small number of participant classes in the original study. The resulting implementations were then subject to the measurement process with the aid of a CASE tool. This tool gathered data in metrics for attributes such as separation of concerns, coupling, cohesion and size. Our study has privileged the 4 patterns that were considered significantly reusable in the CaesarJ implementations to assess if this would also be true for this study. Although our study is not quantitative in nature, it confirms the reusability for *Composite*, *Mediator* and *Observer*.

However the CaesarJ implementation of *Visitor* exposed some limitations in CaesarJ's support for reuse.

The study documented in [22] explored the scalability factor of the AspectJ implementations. The study of issue was further continued in the study of Cacho et al. [14]. This study focused not only on the scalability of aspect-oriented implementations of design patterns in large system, but also how the composition of these patterns scales up. Again, the *separation of concerns*, *coupling*, *cohesion* and *size* attributes were used to evaluate the pattern compositions according to 4 categories for composition issues: *invocation-based composition*, *class-level interlacing*, *method-level interlacing* and pattern *overlapping*. The authors studied 3 medium-sized systems implemented in Java and AspectJ and evaluated 62 compositions in these systems to conclude that the results depend greatly on the patterns involved, the composition intricacies and the application requirements. The authors also consider that the aspectization of the pattern composition is not straightforward and that several design options need to be considered and a global reasoning of the system is sometimes necessary to understand the impact of each design option in the context of the whole system implementation.

Bartholomei et al. recognize the need for a framework that evaluates coupling measures for languages other than AspectJ [7]. The authors present a coupling measurement framework that takes into account both AspectJ and CaesarJ as representatives of 2 of the most well known families of AOP languages. This framework accommodates the definition of different coupling metrics that enable the comparison of Java, AspectJ and CaesarJ implementations. This framework takes into account the different composition mechanisms inherent to both languages. The design pattern implementations provided by this study can be considered good candidates for use cases for this framework, since they provide grounds for coupling comparison between CaesarJ and AspectJ.

7.3 AOP design patterns

The growing number of studies concerning AOP has resulted in a considerable body of knowledge. This accumulating experience can now be used to analyze the common design practices when using aspect-oriented technologies, namely aspect-oriented design patterns.

Noble et al. have produced a study that catalogs 5 patterns of aspect-oriented design [42]. These patterns are called *Spectator*, *Regulator*, *Patch*, *Extension* and *Heterarchical Design*. The authors also describe the problem solved by the pattern, show how aspect-oriented language features are used in the pattern, give characteristic examples of the pattern's use and assess its benefits and liabilities.

Bynens et al. present the aspect-oriented *Elementary Pointcut* design pattern [13]. This pattern aims to improve the reusability of aspects, more specifically, aspects that combine pointcuts and advice in one module. It does so by decomposing the structure of a pointcut in a base aspect into elementary pointcuts that be overridden by concrete sub-aspects. This pattern depends on two language features to take full advantage of its benefits. These features are aspect inheritance with both advice and pointcut inheritance and pointcut overriding and explicit aspect deployment. The former is necessary to reuse pointcut expressions and refer to the inherited pointcut expression inside a redefinition. The latter is necessary to choose which aspects are active. Although CaesarJ supports both this features, this pattern is not present in this work because there is no redefinition of pointcuts present. Pointcuts are used scarcely and at specific occasions. This pattern can be considered as the *Template Method* pattern applied to pointcut definitions.

Horne describes another study about an aspect-oriented design pattern [30]. The *Availability Manager* pattern is described as a solution for applications that are not self-sufficient and need to communicate with external applications and system running locally or remotely, which may not be available at some point. This pattern allows the business part of applications to handle the unavailability the systems on which it depends. This pattern can be related to the *Façade* design pattern because it accommodates for the communication between different applications, but focusing on the particular case of the unavailability of a component.

8. Conclusions and future work

This chapter presents the final conclusions of this dissertation and points some research directions for the future.

8.1 Conclusions

This dissertation has created 30 CaesarJ implementations for 11 design patterns from already existing Java examples. These implementations are described in chapter 5 as well as expressed by a diagram illustrating the structure of the pattern implementation.

The implementations have been characterized by the CaesarJ mechanism used in the pattern implementation and a direct comparison for the use of the pointcut and advice mechanisms has been established. Section 6.2 describes this characterization.

According to the modules used to implement the pattern, the implementations have been submitted to an analysis regarding the level of reuse achieved, differentiating between 4 levels of reuse. The 11 patterns have shown different reuse abilities, where 2 patterns showed direct language support in the CaesarJ implementation, 6 patterns originated reusable modules, 2 patterns presented composition flexibility abilities and 1 pattern demonstrated no ability for reuse. Section 6.3 describes the analysis of the level of reuse achieved.

The patterns that originated reusable modules or presented composition flexibility have been further analyzed as to their composition features. To access these implementations abilities, an additional 6 new pattern implementations have been developed. This analysis is described in section 6.4.

A direct comparison between CaesarJ's and AspectJ's support for deriving reusable modules from pattern implementation is made in section 6.5.

Finally, some general CaesarJ component design guidelines are suggested in section 6.6.

8.2 Future work

This section presents some research directions for future work. It points out some limitations in our work and opportunities for further studies that can use this work as its basis.

This work has created implementations for 11 out of the 23 GoF design patterns. The implementation of the remaining patterns may provide additional insights that could not be derived from this set of implementations. Furthermore, other patterns from different authors should also be the subject of AOP implementations so that aspect-oriented languages can be evaluated in more situations. Such implementations would further expose the strengths and liabilities of AOP languages to new design issues.

This work tackles the composition of individual patterns to an application. A supplementary test to the patterns composition abilities would be to systematically access the problem of composing several patterns into a single application.

Since this work is focused in the comparison between the implementation of 11 design patterns in CaesarJ and AspectJ, more case studies would provide a more significant background for similar studies. Extra implementations in both languages would ease the generalization of the findings presented in this work or challenge them.

This study presents a qualitative analysis of the pattern implementations. Quantitative studies would provide further considerations about the patterns developed and their properties. Such studies have previously focused on AspectJ to measure its modularity, scalability and composition capabilities as well as its coupling attribute. Similar studies would also increase the knowledge about CaesarJ's potential, relative to other AOP languages.

The investigation of AOP languages and AOP itself in general could also benefit from the extension of this work to other AOP languages. Since pattern implementation brings insights regarding the mechanisms existing in a certain language, extending this study to other languages would benefit the study of those languages' mechanisms. Increasing the number of repositories of AOP design pattern implementations would establish a broader basis for the comparison of multiple AOP languages and their respective constructs.

The patterns developed can also be used as the case study for the refactorings for CaesarJ. Since this study comprises the implementation of at least two scenarios of the same design pattern, these implementations can provide the basis on which to derive refactorings. Similarly, the implementations can serve as the subject for the investigation of aspect-oriented design patterns.

These implementations have focused on the implementation of object-oriented design pattern using an aspect-oriented language, namely CaesarJ. The CaesarJ implementations also lend themselves to an investigation of the existence of aspect-oriented design patterns existing in the developed examples.

9. Bibliography

[1] AspectJ implementation of GoF design patterns http://www-lsr.imag.fr/Les.Personnes/Ouafa.Hachani/GoFPatternsInAspectJ.zip

[2] AspectJ project home page, http://www.eclipse.org/aspectj

[3] CaesarJ homepage, http://caesarj.org

[4] HyperJ implementation of GoF design patterns http://www-lsr.imag.fr/Les.Personnes/Ouafa.Hachani/GoFPatternsInHyperJ.zip

[5] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S., A Pattern Language, Oxford University Press, New York, USA, 1977.

[6] Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K., An overview of CaesarJ. Transactions on Aspect-Oriented Software Development I. LNCS, Vol. 3880, pp. 135-173, Feb 2006.

[7] Bartolomei, T., Garcia, A., Sant'Anna, C., Figueiredo, E., Towards a unified coupling framework for measuring aspect-oriented programs, SOQUA'06, Portland, Oregon, USA, 2006.

[8] Baumgartner, G., Läufer, K., Russo, V. F., On the Interaction of Object-Oriented Design Patterns and Programming Languages, Technical report CSD-TR-96-020, Perdue University, 1996.

[9] Bosch, J., Design Patterns as Language Constructs, Journal of Object-Oriented Programming, 11(2): 18-32, 1998.

[10] Bracha, G., Cook W., Mixin-Based Inheritance. Proceedings of ECOOP/OOPSLA, Ottawa, Canada, 1990.

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

[12] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Pattern-Oriented Software Architecture: A System of Patterns, Jon Wiley and Sons, 1996.

[13] Bynens, M., Lagaisse, B., Joosen, W., Truyen, E., The elementary pointcut pattern, BPAOSD'07, Vancouver, British Columbia, Canada, 2007.

[14] Cacho, N., Sant'Anna, C., Figueiredo, E., Garcia, A., Batista, T., Lucena, C., Composing design patterns: a scalability study of aspect-oriented programming, AOSD'06, Bonn, Germany, 2006.

[15] Chambers, C., Object-Oriented Multi-Methods in Cecil, ECOOP'92, Ultrecht, The Netherlands, 1992.

[16] Coplien, J.O., Schmidt, D. C., Pattern languages of program design, Addison-Wesley, 1995.
[17] Eckel, B., Thinking in patterns, revision 0.9. Book in progress, May 20, 2003. Available at http://www.mindviewinc.com/downloads/TIPatterns-0.9.zip.
[18] Ernst, E. Family Polymorphism. ECOOP 2001, Heidelberg, Germany, 2001.
[19] Ernst, E., Ostermann, K., Cook, W. R., A Virtual Class Calculus. 33rd ACM Symposium on Principles of Programming Languages (POPL'06). ACM SIGPLAN-SIGACT, 2006.
[20] Filman, R. E., Elrad T., Clarke S., Aksit M., Aspect-Oriented Software Development, Addison-Wesley, 2005.
[21] Gamma, E., Helm, R., Johnson R., Vlissides, J., Design Patterns – Elements of Reusable Object-Oriented Software, Addison–Wesley, 1995.
[22] 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.
[23] Gasiunas, V., Mezini, M., Ostermann, K., Depedent classes, OOPSLA 2007, Montréal, Quebec, Canada, 2007.
[24] Gasiunas, V., Ostermann, K., Mezini, M., Multidimensional Virtual Classes, Techinical Report TR TUD-ST-2006-03, Technische Universität Darmstadt, 2006.
[25] Greenfield, J., Short, K., Cook, S., Kent, S., Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley Publishing, 2004.
[26] Hachani, O., Bardou, D., On Aspect-Oriented Technology and Object-Oriented Design Patterns. AAOS'03, Darmstadt, Germany, 2003.
[27] Hachani, O., Bardou, D., Using aspect-oriented programming for design patterns implementation, OOIS'02, Montpellier, France, 2002
[28] Hanneman, J., Kiczales, G., Design Pattern Implementation in Java and AspectJ, OOPSLA 2002, Seattle, Washington, USA, 2002.
[29] Hirschfeld, R., Lämmel, R., Wagner, M.,Design Patterns and Aspects – Modular Designs with Seamless Run-Time Integration, 3rd German GI Workshop on AOSD, 2003
[30] Horne, J., The availability manager design pattern, OOPSLA'06, Portland, Oregon, USA, 2006.
[31] Johnson, R. E., Frameworks = (Components + Patterns), Communications of the ACM 40, vol. 10, pp. 39-42, 1997.
[32] Kiczales, G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J-M., Irwin J. Aspect-Oriented Programming, ECOOP'97, vol.1241, pp. 220–242, Jyväskylä, Finnland, 1997.
[33] Kiczales, G., Mezini, M., Aspect-Oriented Programming and Modular Reasoning, ICSE '05, St. Louis, Missouri, USA, 2005.

[34] Laddad, R., AspectJ in Action, Manning, 2003.
[35] Liebermann, H., Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, OOPSLA'86, Portland, Oregon, USA, 1986.
[36] Madsen, O. L., Møller-Pedersen, B., Virtual classes: a powerful mechanism in object-oriented programming, OOPSLA'89, New Orleans, Louisiana, USA, 1989.
[37] Mezini, M. Ostermann, K, Integrating Independent Components with On-Demand Remodularization, OOPSLA'02, New York, New York, USA, 2002.
[38] Mezini, M., Ostermann K., Conquering Aspects with Caesar, AOSD'03, Boston,USA, 2003.
[39] Mezini, M., Ostermann, K., Untangling Crosscutting Models with Caesar, Chapter 8 of [20].
[40] Monteiro, M. P., Fernandes, J. M., Towards a Catalogue of Refactorings and Code Smells for AspectJ. Transactions on Aspect-Oriented Software Development (TAOSD), A. Rashid, M. Aksit (Eds.), Springer LNCS vol. 3880/2006, p. 214 – 258
[41] Monteiro, M. P., Fernandes, J.M., Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns, DSOA'2004, Málaga, Spain, 2004.
[42] Noble, J., Schmidmeier, A., Pearce D. J., Black A. P., Patterns of Aspect-Oriented Design, EuroPLoP'07, Irsee, Germany, 2007.
[43] Nordberg, M. E., Aspect-Oriented Dependency Inversion, OOPSLA'01, Tampa Bay, Florida, USA, 2001.
[44] Pree, W., Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1994.
[45] Rajan, H., Design Patterns in EOS, PLoP '07, Monticello, Illinois, USA, 2007.
[46] Rashid, A., Moreira A., Domain Models are NOT Aspect Free, Models'06, Genoa, Italy, LNCS, Vol 4199, Springer-Verlag (2006): pp. 155-169.
[47] Sousa, E., Monteiro, M. P., An Exploratory Study of CaesarJ Based on Implementations of the Gang-of-Four patterns. Technical report FCT-UNL-DI-SWE-2008-01, New University of Lisbon, 2008
[48] Sousa, E., Monteiro, M. P., Implementing Design Patterns in CaesarJ: an Exploratory Study, SPLAT 2008, Brussels, Belgium, 2008.