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

Dissertação de Mestrado em Engenharia Informática

Quantitative Assessment of Modularity of CaesarJ Components

Miguel Santos Silva Baptista de Almeida

Orientador
Prof. Doutor Miguel Jorge Tavares Pessoa Monteiro

Co-Orientador
Prof. Doutor Miguel Carlos Pacheco Afonso Goulão

14 de Fevereiro de 2011

Nº do aluno: 26619
Nome: Miguel Santos Silva Baptista de Almeida

Título da dissertação: Quantitative Assessment of Modularity of CaesarJ Components

Palavras-Chave:

- Modularidade
- CaesarJ
- Programação orientada por Aspectos
- Métricas de Software

Keywords:

- Modularity
- CaesarJ
- Aspect-Oriented Programming
- Software Metrics

Acknowledgements

Em primeiro lugar quero agradecer aos meus orientadores, Professor Miguel Monteiro e Professor Miguel Goulão. A vossa contínua supervisão, orientação e sugestões e foram essenciais para a concretização deste projecto. Acima de tudo, obrigado pela vossa confiança em mim ao me terem aceitado, mesmo estando com um emprego a tempo inteiro.

À minha família e aos meus amigos, que sempre me apoiaram e incentivaram ao longo da minha vida.

E finalmente, à pessoa mais importante da minha vida. Obrigado Vera pela tua paciência e carinho enquanto estive a fazer a tese. Sem o teu apoio incondicional não teria conseguido.

Resumo

Os defensores do paradigma de programação orientada a aspectos afirmam que este paradigma oferece *melhor* modularidade que a programação orientada a objectos, assim como um melhor suporte para separação de facetas transversais. Embora o AspectJ seja a linguagem de AOP mais conhecida, e alvo de mais estudos, surgiram novas linguagens de programação que propõem diferentes formas de instanciar este paradigma. O CaesarJ é uma destas linguagens. Possui abstracções e mecanismos que o diferenciam do AspectJ, tais como classes virtuais, polimorfismo de família e uma maneira diferente de representar um aspecto.

Qualquer alegação de uma linguagem ser melhor, à luz de um critério bem definido (neste caso, a modularidade), tem que ser apoiada por avaliações rigorosas de implementações feitas nessa linguagem. Este trabalho pretende fazer isso com um estudo comparativo entre as duas linguagens em termos da modularidade que se obtém em software por elas implementado. Em particular, vai-se estudar uma faceta da modularidade: a coesão. Este estudo utiliza da estrutura padrão de relatórios experimentais em Engenharia de Software, assim como todos os testes estatísticos apropriados. Para este fim, foi desenvolvida uma métrica de coesão que foi usada, juntamente com várias métricas de tamanho para avaliar 51 exemplos de implementações de padrões de concepção. No contexto desta dissertação a ferramenta de recolha automática de métricas MuLATo foi adaptada para suportar esta nova métrica de coesão. Os resultados do estudo efectuado sugerem que o CaesarJ é mais verboso que Java mas contem componentes menos complexos e mais coesos.

Abstract

Proponents of the aspect oriented programming paradigm claim that this paradigm yields *better* modularity over object-oriented programming and provides a better support for separation of crosscutting concerns. Although AspectJ is the most popular aspect oriented programming language, and subject of most studies, more recent languages appeared that propose varying ways to realize the paradigm's concepts. CaesarJ is one such language, providing mechanisms that differentiate it from AspectJ, namely virtual classes, family polymorphism and a different way to represent an aspect module.

Any claim of a language being better with respect to some criterion should be supported by rigorous assessments based on that criterion. This work aims to do this with a comparative study using the "standard" experimental report structure for Software Engineering between the two languages in terms of modularity. To this end, a new cohesion metric was developed and used, along with several size metrics to evaluate 51 examples of design pattern implementations. In the context of this dissertation MuLATo, an automated metrics-collecting tool was adapted to support this new metric of cohesion. Results of the study suggest that CaesarJ is more verbose than plain Java but yields more cohesive and less complex components. These results are confirmed with the appropriate statistical tests.

Table of Contents

Index of Figures

Index of Listings

Index of Tables

1. Introduction

In this chapter the motivation behind this dissertation is presented in Section 1.1. 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, finally, in section 1.5, the structure of this document is presented.

1.1 Motivation

Aspect-oriented software development (AOSD) is characterized by a systematic approach to the abstractions, modularity and composability of crosscutting concerns (Rashid & Moreira, 2006).

Separation of Concerns (Parnas, 1972) refers to the ability to decompose and organize systems into manageable modules, which have as little knowledge about the other modules of the system as possible. Separation of concerns helps managing software complexity, enhancing understandability and traceability throughout the development process.

AOSD claims to improve the separation of concerns (Kiczales, et al., 1997) (Bergmans & Aksits, 2001) in software development and contributes to make it easier to maintain and reuse (in comparison to object-oriented programming). This leads to the reduction of the amount of code written and higher cohesion (Alexander, 2003) which makes for better software quality. These claims need to be backed up by rigorous evaluations of design and implementation. We need empirical and quantitative studies and the appropriate measuring tools to verify these claims. Software metrics are means of qualifying software but most existing metrics cannot be applied straightforwardly to AOSD (Zakaria & Hosny, 2003), (Sant'Anna, Garcia, Chavez, Pereira de Lucena, & von Staa, 2003).

Most of the few of these studies that measure modularity for AOP are mainly about AspectJ (Sant'Anna, Garcia, Chavez, Pereira de Lucena, & von Staa, 2003) (Hannemann & Kiczales, 2002) (Kiczales, et al., 1997) (Garcia A. , Sant'Anna, Figueiredo, Kulesza, Lucena, & von Staa, 2006), (Cacho, Sant'Anna, Figueiredo, Garcia, Batista, & Lucena, 2006). Although AspectJ (Ramnivas, 2003) is the first and most well-known AOP language, a great number of languages have been developed afterwards and offer alternatives to AspectJ. Among those languages is CaesarJ.

CaesarJ is an aspect-oriented programming (AOP) language that provides new features of supporting modularity by providing new language constructs and concepts (Aracic, Gasiunas, Mezini, & Ostermann, 2006). It makes use of mechanisms like virtual classes (Madsen &

Møller-Pedersen, 1989) and family polymorphism (Ernst, 2001), which are absent in AspectJ (Ramnivas, 2003). However, the alleged superiority of CaesarJ, when compared to AspectJ and Java is mostly supported by argumentation, usually based on a few illustrative examples (Mezini & Ostermann, 2003). To our knowledge, there has not yet been published any kind of quantitative study for the CaesarJ programming language. Systematic studies and quantitative data supporting such claims are lacking. This project contributes to fill this gap by enabling a rigorous comparison of Java and CaesarJ.

1.2 Problem description

There are not many studies focused on comparing the strengths and limitations of AOP programming languages when compared to Object-Oriented Programming (OOP), as well as their potential for modularity. Even less studies have been done with this aim focused on CaesarJ. One reason for the lack of publications with this objective is the inexistence of a metrics tool that supports this language. Presently, proper tools for collecting metrics for AOP are also lacking and collecting metrics manually is tedious, not scalable and error-prone, especially in complex, non-trivial systems.
The problem this dissertation aims at solving is the lack of cohesion metrics that can be applied to both the Java and CaesarJ languages taking into consideration the latter's specific characteristics. Also, aims to mitigate the lack of a rigorous study comparing CaesarJ and Java, that would provide some insight into the advantages and disadvantages of each one, with a focus on each language's impact in modularity and complexity.

1.3 Presented Solution

To evaluate 2 different programming languages, metrics that support them both are required.
In this dissertation, various *size* metrics are adapted to support CaesarJ and a new cohesion metric is proposed. The Lack of Cohesion in Operations (LCOO-HS) is formalized in an unambiguous manner and is an evolution from a well-known OOP metric, Lack of Cohesion in Methods (Chidamber & Kemerer, 1994) to support CaesarJ.
To validate the proposed metrics, we evolved the MuLATo[1] tool to include support for the automatic collection of the new metrics. Automatic metrics collection is essential, for the sake of the scalability and replicability of experimental studies. The former allows going beyond toy examples in the quantitative studies comparing Java with CaesarJ. As noted in the previous section, the costs of manually collecting metrics would be prohibitive, and the results of such collection would be error-prone. Furthermore, we would miss the potential economies of scale that metrics collection tool support brings to quantitative studies. Researchers trying to replicate our experimental studies in their own context would not benefit much from our metrics collection experience. In practice, this would make replication

[1] http://swen.uwaterloo.ca/~ttonelli/mulato/

very hard. As observed in (Sjoeberg, Hannay, & Hansen, 2005), the lack of replication of experimental studies in Software Engineering is one of the major weaknesses in the validation of claims such as the alleged modularity improvements brought by CaesarJ.
A rigorous comparative study between Java and CaesarJ is made based on 51 implementations of design patterns. This study followed the standard experimental report structure for Software Engineering and uses the appropriate statistical tests to confirm the drawn conclusions.

1.4 Contributions

The contributions this dissertation brings are the following:

- The proposal of a cohesion metric than can be applied to the CaesarJ programming language.
- The adaptation of an existing tool, the Multi Language Assessment Tool (MuLATo) to support new metrics for CaesarJ.
- A comparative analysis between Java and CaesarJ from the source code of 51 examples of Gang-of-Four design patterns, implemented in both Java and CaesarJ. This quantitative study focuses on 5 software metrics:
    - Lines of Code
    - Vocabulary Size
    - Number of Attributes
    - Number of Operations
    - Lack of Cohesion in Operations (HS)

1.5 Document Structure

The rest of this dissertation is organized as follows: Chapter 2 presents the CaesarJ programming language and its mechanisms are discussed, illustrated with the appropriate examples when necessary. Chapter 3 provides relevant background on software attributes and existing metrics that leads to the definition of a new cohesion metric that can be used with Java and CaesarJ, in Chapter 4. Chapter 5 describes the MuLATo tool and the development that was made in the context of this dissertation.
Chapter 6 provides a quantitative case study between Java and CaesarJ in terms of modularity and finally, Chapter 8 ends by presenting the dissertation conclusions and outlines future work.

2. Aspect-oriented programming and CaesarJ

This chapter is structured in the following manner: Section 2.1 introduces the main concepts of Aspect-oriented programming. Section 2.2 explains important concepts of CaesarJ like virtual classes (2.2.1) and family polymorphism (2.2.2).
Section 2.3 presents the CaesarJ programming language and Section 2.4 describes the four conceptual modules that make up and aspect in CaesarJ. Section 2.5 presents the implementation of the *Observer* pattern in both Java (2.5.1) and CaesarJ (2.5.2) to better illustrate the differences between them. Section 2.5.3 mentions some CaesarJ's limitations.

2.1 Aspect Oriented Programming

In computer science, a concern is a particular set of behaviours with a particular goal or purpose needed by a computer program. Hopefully each concern would be represented in its module in order to facilitate the understanding and maintainability of a system. The ability of identifying, encapsulating and manipulating these concerns is known as separation of concerns (Dijkstra, 1976)
Crosscutting concerns are pieces of a program that cut across other concerns and existing module boundaries. They are hard or impossible to be separated into their own modules, and eventually its implementation is scattered across multiple modules and intermixed with the implementation of other concerns (Kiczales, et al., 1997). This is called *code scattering* and *code tangling*, respectively. Some examples of crosscutting concerns are logging, tracing, security and persistence.
Aspect-oriented programming is a programming paradigm that aims to enhance modularity of software, with a focus on the modularization of crosscutting concerns (Rashid & Moreira, 2006). This yields more reusable code, and also more flexibility to couple/decouple, manage, maintain and evolve software systems. This is an improvement over Object Oriented Programming (OOP), which lacks the support for this systematic separation.

2.2 Background on CaesarJ Features

This section presents some relevant features that CaesarJ adds to Java. They are virtual classes (2.2.1) and family polymorphism (2.2.2).

2.2.1 Virtual Classes

Virtual classes (Madsen & Møller-Pedersen, 1989) are inner classes that can be treated like as class methods and subject to dynamic dispatch (the same way as methods in mainstream OOP languages).
The term "virtual" highlights the similarity with the virtual methods present in traditional OOP languages since they correspond to different blocks of code depending on the dynamic type of the running object (Ernst, Ostermann, & Cook, 2006).
The enclosing class of the inner classes is called *family class* and the instance of this family class is called *family object*.
The main difference between Java internal classes and virtual classes is that the latter allows classes the capability to treat inner classes polymorphically, subject to overriding and dynamic (or late) binding. This polymorphism applies to their own class names, with the result that even an expression that uses the "new" keyword to create a new instance is polymorphic, and such expressions are to be variation points of the program.
Each virtual class can be polymorphically refined in any subclass of the enclosing class (the family class). These refinements include adding new methods, fields and inheritance relationships as well as the overriding of inherited methods.
Virtual classes can only be accessed through the family object (the instance of its family class). Consequently, an inner class is identified by its name *and* its enclosing family object (Ernst, 2001). The implementation of the virtual class can be dynamically bound (or late bound) to multiple, different classes (the same way method calls are late bound to specific implementations in Java) depending on the particular instance of the family class, i.e. the family object that is called from.
This is accomplished by providing the type checker the ability to distinguish between multiple instances of a given family of classes, based on the identity of the family object.

2.2.2 Family Polymorphism

Thanks to the virtual class mechanism, CaesarJ can implement *family polymorphism*. Family polymorphism was first proposed by Ernst (Ernst, 2001) and is a mechanism that allows a set of unique classes to be grouped in a larger class, such that the member classes and their instances are uniquely owned by the enclosing instance. This feature solves the problem of expressing and managing family of related classes while enabling the type system to still guarantee type soundness while keeping the flexibility of using an unbounded number of families and ensures that their instances aren't mixed.
The main advantage of family polymorphism is to allow the type checker to allow a lot of new combinations of types without compromising the security of a sound type system. It is the family object's identity that provides the type checker necessary information. The

traditional OO languages do not provide the necessary information to the verifier to do this, which requires new language mechanisms that endow a language that expression.

2.3 CaesarJ

CaesarJ[2] is an AOP extension of the Java programming language that has plug-in support (the CDT: CaesarJ Developer Tool) for the Eclipse platform much like AspectJ[3], the first and most popular AOP language. Although both languages have much in common, there are significant differences between the two languages.
Being an extension of Java, it can be integrated with any Java program up to Java 2. The most recent version, used in this document, is 0.9.0 from April 2008.

CaesarJ uses the *CaesarJ class*, a new type of class that enhances a plain Java class. It is used with the keyword *cclass*. Cclasses extend normal classes by adding several additional constructs. In CaesarJ, every top-level cclass is a family class and any nested cclass is a virtual class. These virtual classes are used to implement family polymorphism.

2.4 CaesarJ Component

In CaesarJ, an aspect is represented by a CaesarJ component composed of several conceptual modules that collaborate with each other (Mezini & Ostermann, 2004). The following sections present the modules that make a CaesarJ's component. They are the *Collaboration Interfaces* (CI) (2.4.1), CaesarJ Implementations (CJImpls) (2.4.2), CaesarJ Bindings (CJBindings) (2.4.3) and Weavelets (2.4.4).
CaesarJ supports multiple inheritance and a typical implementation of a CaesarJ component consists of two "lines" of inheritance: one for the internal implementation of the component, one for the bindings. One of the roles of CI is to declare high-level operations that elements of the "lines" must know, so that the modules of each line can use other modules, without relying on them. Only depend on the "contract" established by the CI
Figure 1 shows a general structure of the aspect component and its modules.

2.4.1 Collaboration Interfaces

The most high-level module of a CaesarJ component is the *Collaboration Interface* (Mezini & Ostermann, 2002). A CI is a family class that contains the declaration (as inner nested CaesarJ classes) of the roles that each participant will have and the collaborations between them. Each of these roles represents an abstraction of the modular structure of the aspect (Mezini & Ostermann, 2003). The CI describes all the methods that each participant class in

[2] http://caesarj.org/
[3] http://www.eclipse.org/aspectj/

the aspect must have. The actual implementation of the aspect component is in the CJBindings and the CJImpls. Different CJBindings and CJImpls can be combined to create distinctive implementations of the component.
CIs support reuse of the same functionality in different contexts (Aracic, Gasiunas, Mezini, & Ostermann, 2006).

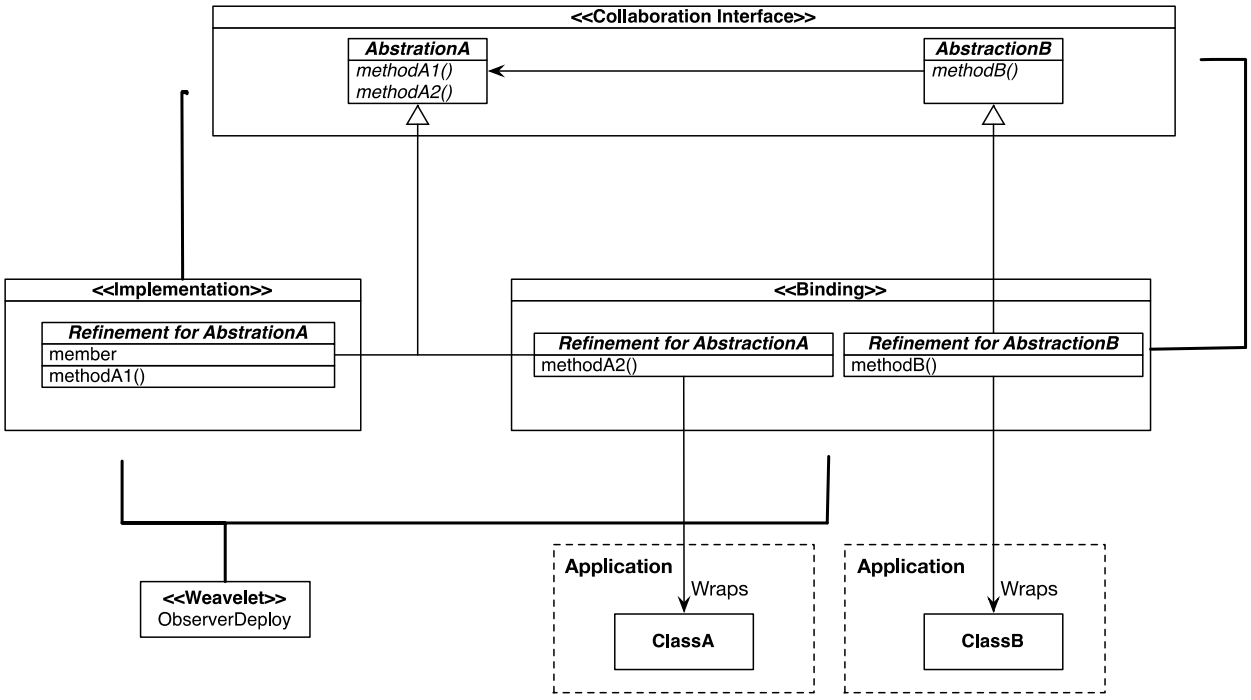

Figure 1. General structure of a CaesarJ component

Figure 1 illustrates the general structure of a CaesarJ component, describing all CaesarJ modules, their mutual relations and the relations to classes in applications as explained the previous sections.

2.4.2 CaesarJ Implementations

The *CaesarJ Implementation* is a cclass that implements the members and methods inherited from the CI that are common to the majority of the implementations i.e. it implements the context independent parts of the CI. There can be more than one CJImpl for a single CI. This means that at any given moment one CJImpl can be switched by another without impact on the code of the remaining modules. In addition, because the CJImpl is based on an interface (the CI), it can invoke methods or members from other modules in a transparent manner. The CJImpl can also add members and methods that are not defined in the CI.

2.4.3 CaesarJ Bindings

CJBinding acts like the "glue" between the component and a particular application.
All the application specific parts are implemented in the CJBinding. This module complements the CJImpl for the purpose of providing the implementation of the CaesarJ component. While the CJImpl implements the abstract part of the CI, the CJBinding implements the methods that enclose the entire logic specific to the application. This way the aspect is composed to that application. This mapping is done through Wrapper Classes (Mezini & Ostermann, Integrating independent components with on-demand remodularization, 2002).
CJBinding also support the use of the pointcut and advice mechanism, much like AspectJ (Kiczales, Hilsdale, Hugunin, Kersten, Palm, & Griswold, 2001). In this mechanism, a joinpoint is a clear point in the program flow. A pointcut picks out a certain joinpoint and value at that point and an advice is the code that is executed when a joinpoint is reached.
An important difference between the pointcuts of AspectJ and CaesarJ is that in the latter the pointcuts can be activated and deactivated during the deployment while the pointcuts of AspectJ are always active. This introduces a dynamic that doesn't exist in the pointcuts of AspectJ.

Wrapper Classes

Wrapper classes map one or more application objects to the roles defined in the CI. This application objects are the *wrappees* in the class.
To avoid multiple wrappers for the same object CaesarJ has a mechanism implemented called *wrapper recycling*. This mechanism guarantees a one and only one wrapper for each unique pair of a component role instance and a specific object in the application domain. To ensure this, instead of instantiating a new wrapper with the new constructor call, it is used an *outerClassInstance.innerClassInstance(constructarg)* construct, where *outerClassInstance* is the family object, *innerClassInstance* is the virtual class that defines the wrapper class, and *constructarg* is the wrappee. With this constructor whenever an instance with these arguments already exists, the existing object is returned; otherwise, a new instance is created.
Wrappers also have some limitations; currently, it is not possible to for one wrapper class to wrap two objects (Gasiunas, Mezini, & Ostermann, 2007). This prevents the mapping of new concepts, internal to the CaesarJ component, based on the application objects to which it wants to compose. Also, 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 (Braz, 2009)

2.4.4 Weavelets

A *Weavelet* is a cclass that composes the CJImpls with the CJBinding to create the complete aspect component. This procedure is done through *mix-in composition* (Bracha & Cook, 1990); it takes abstract subclasses (in this case the CJImpls and CJBinding) to specialize the behaviour of the parent class (the CI). The component that represents an aspect in CaesarJ is the instantiation of a Weavelet.

2.4.5 Component Instantiation and deployment

In CaesarJ and unlike in AspectJ, aspects can be explicitly instantiated through the "new" keyword, allowing the instantiation of several Weavelets.
These provide the developer the capability of creating various aspect instances in the same application and manage them as different objects.
Once a Weavelet is defined, it must be deployed in order to activate its pointcuts and advices. This deployment can be done statically or dynamically. Static deployment can be done in *compile time* (through the use of the *deployed* modifier) or in *load time* (with the *deployed* modifier in the instantiation of a *final static* object). Dynamic deployment can be either *local* or *thread-based*. To dynamically deploy the component must be instanced by the instantiation of the Weavelet. For local deployment, the keywords *deploy* (which defines the scope of the aspect) and *undeploy* are used to respectively activate and deactivate the pointcut/advice parts of the aspect. In thread-based deployment, the activation of the component is done with a *deploy* block. The aspect is deployed within the scope of the control flow inside the block without having influence in concurrent executions.

2.5 Illustrating example: The *Observer* Pattern

A design pattern (Gamma, Helm, Johnson, & Vlissides, 1995) is a description of a common software engineering problem and its solution. The patterns implementation is affected by its implementation language. The most popular design patterns are the 23 Gang-of-Four (GoF) patterns that suggest flexible solutions for several design and structural issues. This collection of patterns provide a rich catalogue of problems and corresponding solutions that can be found in complex systems so its implementations make for good case studies for research (Hannemann & Kiczales, 2002) (Sousa & Monteiro, 2008).
To better illustrate the differences between Java and CaesarJ, as well its advantages, the *Observer* design pattern is used in this section.
The primary aim of *Observer* is to define a one-to-many relation between interdependent objects. More specifically, it defines one (or more) object's dependency of another object's state while avoiding their direct dependency on each other at the level of the source-code. It

must be possible to remove one or the other from the system without giving rise to compilation errors.

Figure 2 illustrates the structure of the *Observer* pattern. *Observer* prescribes two roles: *Subject* and *Observer*. *Subject* is the class that Observer classes depend upon. When a Subject object's state changes, the Observer objects interested in it must be notified. Class *Subject* keeps track of everybody who wants to be updated when a change happens. For this purpose the *Subject* class must always have a *Notify()* function that notifies observer objects whenever the subject changes its state. The Subject must also provide the operations *addObserver(Observer)* and *removeObserver(Observer)* to add or remove respectively an observer and a list of all the observers that are interested in the subject. Class *Observer* is interested in the state of class Subject. If there is a change in the Subject class, the observers must be updated. As such, its class must have an *Update()* function that is called by the *notify()* function of the *Subject*. Class *ConcreteSubject* holds the state that the observers are interest in and the class *ConcreteObserver* implements the *Update()* method.

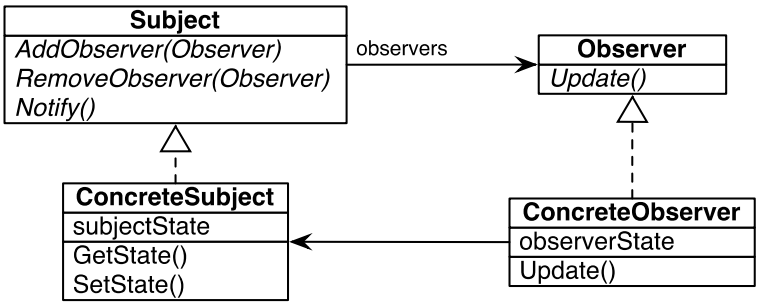

Figure 2. Structure of the design pattern *Observer*

2.5.1 Implementing *Observer* in Java

The following example of *Observer* was been taken from Bruce Eckel's book "Thinking in Patterns" (Eckel, 2003). The subject in this case represents a *Flower*. A Flower can open or close its petals. Listing 1 represents the Flower class. This class extends the *Observable* class of the Java API to implement all the logic related to the *Subject* role, that includes the storage of the interested observers as well the means to notify them of the status updates. The *Bee* and the *Hummingbird* will take the role of *Observer*s. They are interested in knowing when the flower opens or closes its petals. When the petals open its time for the Bee and Hummingbird to eat and when it closes it is time for them to sleep.

Listing 2 represents the Bee class. This class implements the *Observer* interface as well as its *update()* method. The Hummingbird class is similar to the Bee's so it won't be depicted here.

In Listing 1 and Listing 2, instead of extending and implementing the *Observable* and *Observer* classes directly, inner classes are used to isolate the code related to the assigned roles of 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. Flower Class for the Observer example in Java

```
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. Bee Class of the Observer example in Java

2.5.2 Observer in CaesarJ

The CaesarJ implementations of the Flower example of *Observer* were produced by Sousa et al. (Sousa & Monteiro, 2008). Figure 3 illustrates the class diagram for the CaesarJ's implementation of the Flower Observer example. Figure 4 represents the class diagram for the collaboration interface of the Flower Observer example.

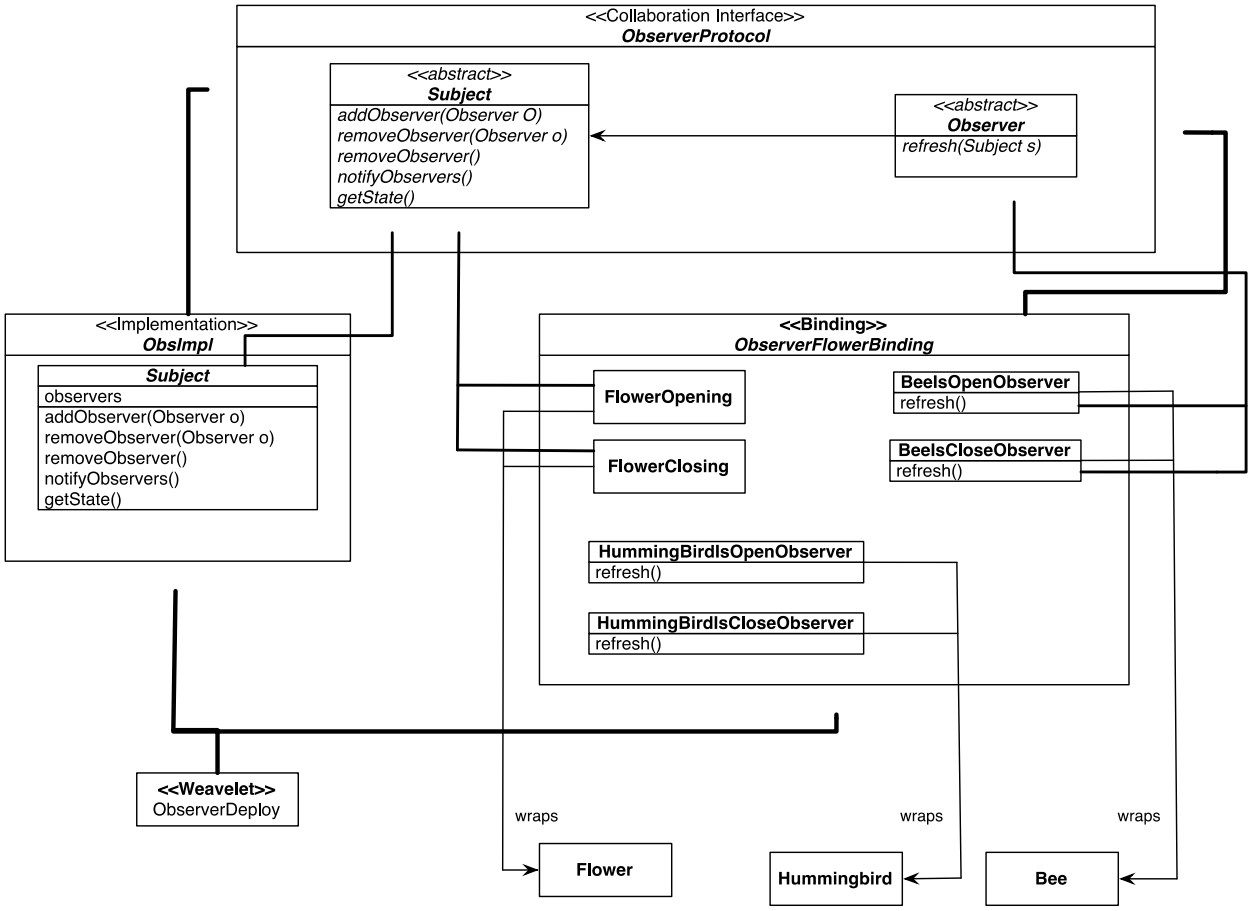

Figure 3. Class Diagram of the CaesarJ Flower Observer example

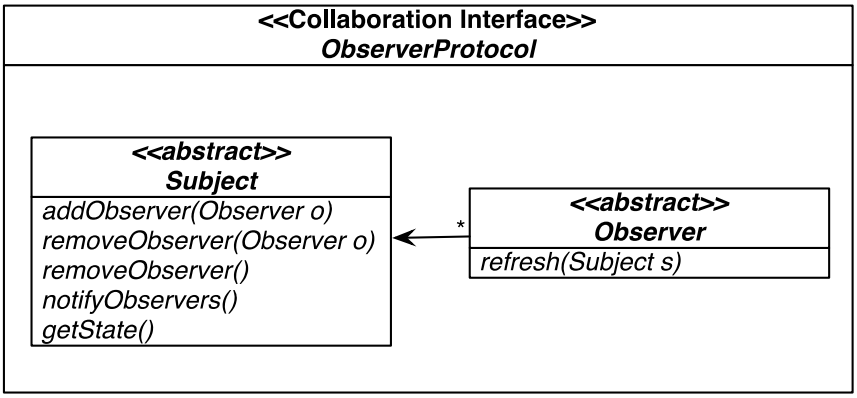

Figure 4. Class Diagram of the Collaboration Interface for the Flower Observer example

```
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 CaesarJ Flower Observer example

In Listing 3, the abstract cclass *ObserverProtocol* describes the collaboration between the two roles of the *Subject* and the *Observer*. Both these classes are abstract virtual classes that are mutually recursive in that the name of one type is used in the declaration of the other.
All of these classes are inner nested CaesarJ classes (and therefore virtual) and their implementations are in the modules presented next.
The CJImpl in Listing 4 implements the CI shown in Listing 3, comprising the *addObserver*, *removeObserver*, and *notifyObservers* abstract methods from the previous listing because they aren't context sensitive.

```
01 public abstract cclass ObsImpl1 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 example

Listing 5 illustrates an alternative CJBinding for the Observer example where the programmer chooses to implement the list of observers with a *Hashmap* instead of an *ArrayList* as in the first example. This illustrates the reuse that CJImpls provide since one can replace one implementation with another without impact on the remaining modules of the CaesarJ component.

```
01  public abstract cclass ObsImpl extends ObserverProtocol{
02      public cclass Subject {
03              private HashMap observers = new HashMap();
04
05              public void addObserver(Observer obs){
06                      this.observers.put(obs, 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.keySet().iterator();
16                      while(it.hasNext())
17                              ((Observer)it.next()).refresh(this);
18              }
19              public Object getState(){
20                      return null;
21              }
22      }
23  }
```

Listing 5. Alternative CaesarJ Implementation of the Observer example

In the CJBinding of Listing 6, the pointcut *openCloseEvents* captures the method call *isOpen* of the *Flower* class so it notifies the interested observers of this state change. This CJBinding has six wrappers classes: *FlowerOpening*, *FlowerClosing*, *BeeIsOpenObserver*, *BeeIsCloseObserver*, *HummingbirdIsOpenObserver* and *HummingbirdIsCloseObserver*.
In *BeeIsOpenObserver* for instance, *Bee* takes the role of wrappee.

```
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
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) &&
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 example

Figure 5 shows the structure of the CJBinding of Flower Observer example, and its wrapping relations to the classes in the base application.

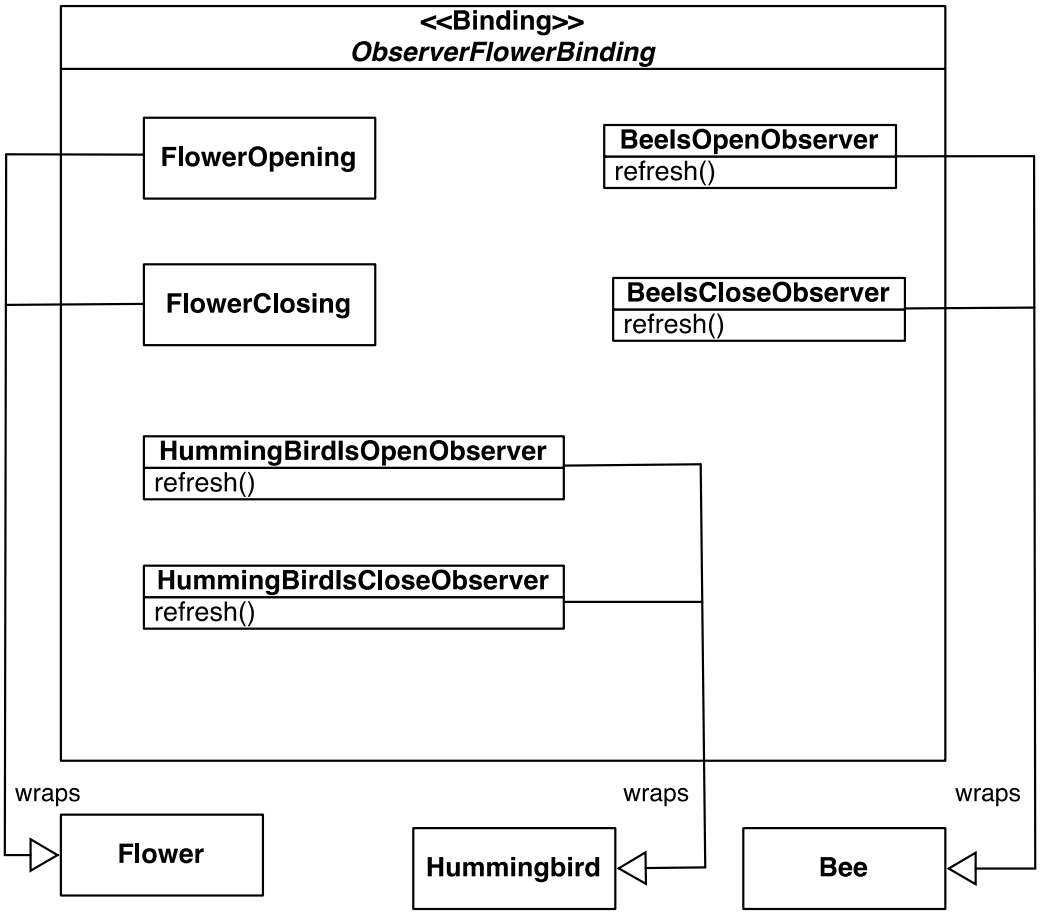

Figure 5. Class Diagram of the CaesarJ Binding for the Flower Observer example

Listing 7 and Listing 8 illustrate two possible Weavelets that use the CJImpls *ObsImpl1* and *ObsImpl2* respectively that were depicted in and Listing 5.

```
01 public cclass FlowerObserverDeploy extends ObsImpl1  & ObsBinding{
02 }
```

Listing 7. Weavelet for the Observer example

```
01 public cclass FlowerObserverDeploy extends ObsImpl2  & ObsBinding{
02 }
```

Listing 8. Alternative Weavelet for the Observer example

The following listings, Listing 9 and Listing 10, present the Flower and Bee classes respectively without the all the Observer pattern logic that went to the aspect. It is visibly simpler since all the code relative to the pattern is in the aspect's module.

```
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 9. Class Flower without the Observer Logic

```
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 10. Class Bee without the Observer Logic

2.5.3 CaesarJ Limitations

Cclasses also have some limitations. Cclasses cannot be casted as regular java classes, or vice-versa, because cclasses cannot extend Java's regular classes (although they can implement Java interfaces). In addition, arrays of cclass modules are not supported.

3. Existing metrics for modularity

The current chapter is structured in the following manner: Section 3.1 presents some software attributes related to modularity. Afterwards, section 3.2 describes 5 AOP size metrics formalized to CaesarJ (3.2.1) and provides some background of available cohesion metrics (3.2.2).

3.1 Software Attributes

One of the most fundamental principles of solving large and complex problems is that breaking up the problem into smaller parts enhances understandability and tractability (Polya, 1957)

Modularity is the division of a software system in smaller parts (modules). This software attribute allows a program to be intellectually manageable (Myers, 1978). The way the division of these modules is made is essential to achieve good modularity. A module is a unit whose structural elements are powerfully connected among them and relatively weakly connected to elements in other units (Baldwin & Clark, 1999).

One of the most well accepted ways modules should be separated is according to their functionality, i.e. a module should be functionally independent (Pressman, 2000).

Modularity can be applied at several levels of abstraction, from the requirements specification's level to the executable code one. In this dissertation, the term modularity refers to modularity at the code level.

Good modularity helps to decrease the complexity of a system, in order to make it more likely to be managed.

In this chapter, two classic software attributes (Chidamber & Kemerer, 1994) related to modularity are explained, specifically cohesion (3.1.1) and coupling (3.1.2) and one attribute related to complexity, namely size (3.1.3).

3.1.1 Cohesion

Cohesion is defined by how strongly related are the interactions within a software module (Myers, 1978). These interactions must have a common functional objective. Cohesion is positively correlated with the number of interactions. Cohesiveness of methods within a class is desirable, since it promotes encapsulation of objects (Chidamber & Kemerer, 1994). Low

cohesion increases complexity (Zakaria & Hosny, 2003) (Rosenberg & Hyatt, 1997), thereby increasing the likelihood of errors during the development process, increasing the difficulty to maintain, reuse and understand a software system. One way modularity can be assessed is by combining coupling and cohesion. (Pressman, 2000) (Tsang, Clarke, & Baniassad, 2004).

3.1.2 Coupling

Coupling is the degree of interaction between modules (Myers, 1978), i.e., the measure of relative interdependence between modules. Even if, by definition, modules must interact with each other to satisfy requirements, one module should depend as little as possible on other modules. Low coupling is positively correlated with understandability, reusability and reduction on the impact of modifications and consequently error propagation (Pressman, 2000).

3.1.3 Size

Size is the most well-known software attribute. It is the measure of the physical size of the system's design and code. Size is a key factor in a program complexity (Fenton & Pfleeger, 1998).
Complexity has a large impact on modularity (Kumar, Kumar, & Grover, 2008).

3.2 Software Metrics

Software metrics measure properties of software artefacts defined at different abstraction levels. Metrics are often used to get information about quality attributes related to the design and implementation of software applications. A software metric is a function that has as input a software artefact (or a set of related software artefacts) and returns a numeric value that can be interpreted and evaluated (Kaner & Bond, 2004). This helps to decide what good design and implementation choices one must make in a consistent manner with an objective and repeatable evaluation, regardless of who makes it. Unfortunately, measuring software can be a difficult task. It is not always clear what to measure, or how to interpret the collected metrics. Also, some metrics are ambiguously defined, which leads to the same code having different results for allegedly the same metric, depending how the metric's definition is interpreted and the metric's collection is implemented. For instance, Lincke et al. surveyed a number of tools and concluded that for the same software system and metrics, the metrics values are tool depended (Lincke, Lundberg, & Löwe, 2008).
A set of AOP metrics is required to evaluate AOP systems since OOP metrics can't be applied to them in a straightforward manner (Zakaria & Hosny, 2003), (Zhao, Towards A Metrics Suite for Aspect-Oriented Software, 2002).

A number of metrics AOP metrics have been proposed but they seem to be specific to AspectJ (Gélinas, Badri, & Badri, 2006), or for AspectJ-like languages. (Zhao, Measuring Coupling in Aspect-Oriented Systems, 2004) (Sant'Anna, Garcia, Chavez, Pereira de Lucena, & von Staa, 2003). Although AspectJ and CaesarJ share some features (like the pointcut-advice mechanism), they have many differences. AspectJ's aspect module is very different from traditional classes. Not only it has new constructs, but it also lacks some capabilities common in classes such as explicit instantiation with "*new*" (Rajan & Sullivan, 2005). Also, CaesarJ doesn't have the "aspect" as an individual construct and has several features that are not present in AspectJ. For these reasons, most AOP metrics are only suitable for AspectJ, disregarding other AOP languages.

This chapter describes some metrics adapted for CaesarJ grouped by the software attributes that they measure.
Section 3.2.1 lists Size metrics and section 3.2.2 and describe existing cohesion metrics.
The term *component* in the current chapter has a different meaning from the previous one. In chapter 2, a (CaesarJ) component is an aspect that is composed by several conceptual modules. In the current chapter, a component has a more general meaning (it can be a class, interface or a cclass).

3.2.1 Size Metrics

Size metrics measure the size of the software (in terms of the length of the system's design and code). Usually the bigger the system, the harder is to understand it. In other words, size and complexity are often positively correlated (Sant'Anna, Garcia, Chavez, Pereira de Lucena, & von Staa, 2003), (Fenton N. , 1994). The following size metrics are taken from Sant'Anna et al.'s suite of metrics (Sant'Anna, Garcia, Chavez, Pereira de Lucena, & von Staa, 2003). Though it claims to be for AOP in general, its definitions do not take CaesarJ's specific constructs into account. Therefore its definitions were updated, in the context of this dissertation, to reflect them. They are the following:

- **Vocabulary Size** (VS): Also known as *Number of Components*, this metric counts the number of system components (classes and aspects) in in the application. It covers only the name of the components and not instances. Although its original definition (Sant'Anna, Garcia, Chavez, Pereira de Lucena, & von Staa, 2003) only considers a component to be a class or an aspect, this definition is extended to consider also cclasses.

- **Lines of Code** (LOC): The simplest and most well-known metric, it counts the number of code lines. The count criteria must be consistent when the results are compared i.e., the formatting style should be the same. Documentation and

implementation comments as well as blank lines are not taken into account for this purpose.

- **Number of Attributes** (NOA): This metric counts the number of attributes per component. The term "component" is the same as the one defined in the VS metric. Inherited attributes are not counted.

- **Number of Operations** (NOO): This metric counts the number of operations per component. Inherited operations are not counted, unless they are overridden. Although the original definition of "operation" only comprised of methods and advices, it is extended in the context of this thesis to include also, constructors, wrapper constructors, pointcuts, and declare statements.

- **Weighted Operations per Component** (WOC): WOC (Sant'Anna, Garcia, Chavez, Pereira de Lucena, & von Staa, 2003) measures the *complexity* of a component in terms of its operations. The complexity of each operation can be measured by the sum of parameters each operation has (the more parameters it has, the more complex the operation is). WOC is an extension of CK's *Weighted Methods Per Class* metric (Chidamber & Kemerer, 1994).

3.2.2 Existing Cohesion Metrics

Cohesion metrics measure how strong the relation between internal components is in terms of responsibilities. High cohesion is typically desirable because the readability and reusability of a highly cohesive system is greater while its complexity is kept manageable (Barnes, Jr., Hale, Hale, & Smith, 2006).

Lack of Cohesion in Methods (LCOM-CK): Chidamber and Kemerer define (Chidamber & Kemerer, 1994) the cohesion of a class as the degree of similarity of the methods within a class. In this OO metric, we take each pair of methods in the class and determine the set of fields each access. If the two methods have disjointed sets of field accesses (i.e., no common attribute references), the count P increases by one. If the two methods share at least one field access, Q is incremented by one. After considering each pair of methods:

$$LCOM\text{-}CK = \begin{matrix} P - Q\ ,\ P > Q \\ 0, otherwise \end{matrix}$$

If a class has a large number of similar methods, it results in a low LCOM-CK value, which indicates high cohesion between them. This also indicates potentially high reusability and good class design. The viewpoints listed for this metric are:

1) Cohesiveness of methods within a class is desirable, since it promotes encapsulation of objects.
2) Lack of cohesion implies classes should probably be split into two or more sub-classes.
3) Any measure of disparateness of methods helps to identify flaws in the design of classes.
4) Low cohesion contributes to complexity, thereby increasing the likelihood of errors during the development process.

However this metric has some problems. Since it isn't normalized and there is no guideline on the interpretation of any particular value, it is not obvious by the results the degree of cohesiveness of a class (Henderson-Sellers, 1996): if LCOM-CK = 0 the class is maximally cohesive, but there are not any reference values to evaluate the need to split that class if LCOM-CK > 1. Also, this metric can give a value a zero for very different reasons such as $/P/ = /Q/$ (which, by itself, shouldn't imply the maximum cohesiveness).
Chidamber and Kemerer (CK) do not state if inherited methods or attributes are included in this metric.

Lack of Cohesion in Operations (LCOO): LCOO (Sant'Anna, Garcia, Chavez, Pereira de Lucena, & von Staa, 2003) is an AOP metric that measures the quantity of method/advice pairs of a component (classes and aspects) that do not access the same instance variable. Consequently, it measures the lack of cohesion of a component. Given a component C1 and operations (methods and advices) $O_i, \ldots, O_n$

- $\{I_j\}$ is set of instance variables used by operation O_j.
- $/P/$ is the number of null intersections between instance variables sets.
- $/Q/$ is the number of non-empty intersections between instance variables sets.

- $LCOO = \begin{matrix} P - Q\ ,\ P > Q \\ 0, otherwise \end{matrix}$

This metric extends the CK's metric *Lack of Cohesion of Methods* (LCOM-CK) (Chidamber & Kemerer, 1994) and, by extension, inherits all its shortcomings (Gélinas, Badri, & Badri, 2006). Also, this metric is based on the principle that all AOP languages are similar to AspectJ. Treating advices as methods and aspects as classes seems a rather simplistic view for CaesarJ and doesn't cover the constructs not shared with AspectJ.

Lack of Cohesion in Methods (LCOM-HS): This is an improvement on the previous LCOM and is proposed by Henderson-Sellers (Henderson-Sellers, 1996). It is defined as follows:

$$LCOM\text{-}HS = \frac{\frac{1}{a}\sum_{j=1}^{a}\mu(A_j) - m}{1 - m}$$

where a and m are the number of attributes and methods of the class, respectively, and $\mu(Aj)$ is the number of methods that access the datum $Aj\ (1 \leq j \leq a)$.

The LCOM-HS value varies between 0 and 1. If LCOM-HS = 0, each method of the class references every attribute of the class (which results in perfect cohesion). If LCOM-HS = 1, each method of the class references a unique attribute each. Values between 0 and 1 are to be interpreted as percentages of the perfect value.
If there is only one method or less in a class, or if there are no attributes in a class LCOM-HS is undefined. An undefined LCOM-HS is displayed as zero. This way, the metric is normalized and simplified.
The authors do not mention whether inherited methods and attributes are accounted for.

4. A new cohesion metric for Java and CaesarJ

As previously stated, OOP existing metrics cannot be applied straightforwardly to AOP and although there are some metrics for the latter, most of them are for AspectJ-like languages. This chapter proposes an adapted version of the LCOO metric with the CaesarJ features in mind. It also clarifies some ambiguities not covered in its original definition.
This metric measures the lack of cohesion of a component in Java *and* CaesarJ.
As this metric is an extension of the LCOM metric of Henderson-Sellers to be used with CaesarJ, the same theoretical concepts are used; only the calculated variables will be revised to reflect the features of CaesarJ.
Many of the current OOP cohesion (and coupling) metrics have very ambiguous definitions and can yield very different results depending on their interpretation. When defining a software metric, one should establish a proper terminology and formalism in an unambiguously and fully operational manner so that no additional interpretation is required on behalf of the user of the metric (Briand, Daly, & Wust, 1998). This metric uses the same terminology and formalism presented by Bartolomei et al.. (Bartolomei, Garcia, Sant'Anna, & Figueiredo, 2006). In this context we will only use the definitions applicable for Java and CaesarJ.

4.1 Terminology and formalism

The key elements of this metric are:

- **Component**: includes classes, interfaces, and cclasses.
- **Operation**: includes methods, constructors, wrapper constructors, pointcuts, advices, declare statements and static initializers.
- **Attribute:** includes all fields, (static and non-static, public, private and protected).

Another option pondered would be to consider as a component the group of cclasses that comprise the internal structure of an aspect in CaesarJ (explained in section 2.4). The reasoning is that since the "aspect component" is so closely connected because of virtual classes and family polymorphism that it should be evaluated together. However, the abstraction level for this "aspect component" is different than that used considering just individual classes (or cclasses) and it does not seem reasonable to consider individual classes and a group of cclasses at the same level. Furthermore, the "aspect component" is just a recommendation made by the CaesarJ's authors to compose an aspect, taking full advantage

of its features. It is not mandatory to use it that way. Also, precisely identifying this "aspect component" automatically is not feasible.

4.2 Definition of the metric

This new metric is defined as:

$$LCOO\text{-}HS = \frac{\frac{1}{a}\sum_{j=1}^{a} \mu(A_j) - o}{1 - o}$$

Where o, a and $\mu(Aj)$ are defined as

- o: number of operations in a class
- a: number of attributes in a class.
- $\mu(Aj)$: number of operations that access the that access the datum $Aj\ (1 \leq j \leq a)$.

If the component doesn't have any attributes ($a = 0$) or there are no more than one operation in it ($o <= 1$) then LCOO-HS = 0.
The scale of this metric remains the same as LCOO-HS:

- The measure yields 0, if each operation of the component references every attribute of the component (perfect cohesion).
- The measure yields 1, if every operation of the component accesses a unique attribute.
- Values between 0 and 1 are interpreted as percentages of the perfect value.

4.3 Inheritance

Next some typical problems and ambiguities usually found in cohesion metrics will be reviewed and a proposal to address them is presented.
Most cohesion metrics do not address the influence of inheritance, or how to deal with it (Briand, Daly, & Wust, 1998). There are several approaches available one can take:

1. Count only newly implemented (not inherited or overridden) operations and attributes.
2. Count overridden and newly implemented operations and attributes
3. Count all inherited (overridden and non-overridden) and newly implemented operations and attributes.

The first option doesn't make much sense in the context of this dissertation. Since the attribute/operation relation is the main element being analysed in cohesion, excluding overridden operations that could use new attributes would influence the result without any empirical justification.

This metric will only count overridden and newly implemented operations and attributes because these are the ones that add something new to the class. The inherited (non-overridden) operations are already counted in the superclass, so counting them again in the sub-classes would artificially increase their value.

4.4 Access Methods

An access method provides read or write access to an attribute of the class. Access methods typically reference only one attribute, specifically the one they provide access to.
The presence of access methods artificially increases the value of LCOM. They increase the number of pairs of methods in the class that do not use attributes in common (Briand, Daly, & Wust, 1998).
Access methods can also inflate the value of LCOM in some circumstances (Hitz & Montazeri, 1995). If a method M references an access method M' instead of the attribute A, the relation M-A is not counted in the metric.
Unfortunately, excluding these methods in the implementation of a tool is very challenging, because, to our knowledge, the only way to identify these methods automatically is through the name (for example, getter methods are those whose names start with "get"). The same naming convention would have to be ensured in all the systems where this metric is applied, thus narrowing the set of systems with which the tool can be used. Furthermore, this could lead to inaccurate results. For example, if the convention were *setAttributeName*, a method with the name *settleAccount* or *setupWebsite* would also be excluded. Therefore, access methods are counted.

4.5 Validation of the metric

To validate the LCOO-HS metric, Abreu's set of criteria for the development of software metrics (Brito e Abreu & Carapuça, 1994) is used. It comprises of 7 criterions:

i) Metrics determination should be formally defined

LCOO-HS is formally defined so different users at different times or places yield the same results when measuring the same system. As the results of the metric are considered a percentage, they are also objective.

ii) Non-size metrics should be system size independent

LCOO-HS can be collected, analyzed and compared with many different projects with different sizes because its results have a fixed scale. It is not dependent on the project's size.

iii) Metrics should be dimensionless or expressed in some consistent unit system

As previously stated, the units of measurement of LCOO-HS are not subjective.

iv) Metrics should be obtainable early in the life cycle

LCOO-HS does not meet this criterion. This metric is only collectable when code is available. This metric was developed to be used the context of a study that has the purpose of determining to what extent the use of a programming language affects the modularity and complexity. This criterion is particularly relevant if we want to detect potential quality problems with a particular design before we get to the implementation phase. The "relaxation" of this criterion is not compromising for the context of this work.

v) Metrics should be down scalable

LCOO-HS can be applied to a whole system or to each one of its modules or sub-systems.

vi) Metrics should be easily computable

A tool has already been developed, in the context of this dissertation, to collect LCOO-HS.

vii) Metrics should be language independent

LCOO-HS is valid for 2 programming languages: Java and CaesarJ. These languages are the subjects of the study developed in the context of this dissertation
It is planned to extend it to support more AOP languages.

4.6 Illustrating example: The *Observer* pattern

In this section the LCOO-HS metric is illustrated using the same example of the Observer pattern that was previously described in chapter 2.5. Table 1 lists the components that make the Flower Observer example and its LCOO-HS metric values. The components named *component1$component2* represent an inner component *component2* of the top-level component *component1*.

Component	Attributes	Operations	$\sum_{j=1}^{a} \mu A_j$	LCOO-HS
ObserverProtocol	0	0	0	0
ObserverProtocol$Subject	0	5	0	0
ObserverProtocol$Observer	0	1	0	0
FlowerObserverDeploy	0	0	0	0

ObserverFlowerBinding	0	2	0	0
ObserverFlowerBinding$FlowerOpening	0	0	0	0
ObserverFlowerBinding$FlowerClosing	0	0	0	0
ObserverFlowerBinding$BeeIsOpenObserver	0	1	0	0
ObserverFlowerBinding$BeeIsCloseObserver	0	1	0	0
ObserverFlowerBinding$HummingbirdIsOpenObserver	0	1	0	0
ObserverFlowerBinding$HummingbirdIsCloseObserver	0	1	0	0
ObsImpl1	0	0	0	0
ObsImpl1$Subject	1	5	5	0,25
Flower	1	4	4	0
Bee	1	3	3	0

Table 1. LCOO-HS results for the Flower Observer example

Neither the **ObserverProtocol** component (and its two inner classes) depicted in Listing 3 nor the **FlowerObserverDeploy** component (depicted in Listing 7) have any attributes, so LCOO-HS = 0

The **ObserverFlowerBinding** component (showed in Listing 6) is an interesting case that is very common in CaesarJ. It does not have attributes so LCOO-HS = 0; this component also has 6 internal components (the wrapper classes), each one with o >= 1, so, for each one of them, LCOO-HS = 0.

The **ObsImpl1** component (illustrated in Listing 4) has LCOO-HS = 0. It has no attributes or operations). It has one inner class **Subject** with the following variables:

- $a = 1$;
- $o = 5$;
- $\sum_{j=1}^{a} \mu A_j = 5$;

replacing these values to the LCOO-HS formula results in LCOO-HS = ((4/1) – 5 / (1-5) = 0,25

The **Flower** component represented in Listing 9 the variables of the component are

- $a = 1$;
- $o = 4$;
- $\sum_{j=1}^{a} \mu A_j = 4$;

applying these values to the LCOO-HS formula ends in LCOO-HS = ((4/1) – 4) / (1-4) = 0

In the **Bee** component represented in Listing 10 the variables of the component are

- $a = 1$;

- $o = 3$;
- $\sum_{j=1}^{a} \mu \ A_j = 3$;

which applied to the LCOO-HS formula results in LCOO-HS = ((3/1) – 3) / (1-3) = 0

5. Tool support for metric collection

Collecting metrics manually, especially in complex, non-trivial systems, is a tedious, not scalable and error-prone activity. For this reason, proper tool support is advisable for metrics collection. There are several applications to collect metrics for Java available on the Internet. However, the same metric calculated by different tools, often gives different results (Lincke, Lundberg, & Löwe, 2008). This seems to be particularly frequent in cohesion and coupling metrics. For this reason, when comparing the same metric in different systems, one way of mitigating this problem is to have the same metrics collection tool be used to ensure the coherence and comparability of those results. Since this study comprises of two different programming languages, a tool that supports both languages is necessary. To the best of our knowledge, there is only one metric collection tool that works for CaesarJ. The Multi Language Assessment Tool (MuLATo)[4] is a Java application for collecting metrics from programs written in several languages. The core module provides parsers for Java, AspectJ and CaesarJ programs and can be extended for other languages. The GUI was developed as an Eclipse plug-in, but other types of user interfaces can be built on top of the core module. The most recent version is 0.1.1 from September of 2006 and it is available under open source in (Bartolomei T. T., 2007).

MuLATo can use Java, AspectJ or CaesarJ projects as input and creates a CSV file with its supported metrics.

Table 2 shows the metrics MuLATo currently supports for CaesarJ, aggregated by software attribute.

Software Attribute	Metric
Size	Weighted Operations per Component (WOC)
	Number of Attributes (NOA)
	Number of Operations (NOO)
	Vocabulary Size (VS)

Table 2. CaesarJ metrics supported by MuLATo 0.1.1

Unfortunately only a few size metrics are available in the current version.

In the context of this dissertation, the MuLATo tool was further developed to support new metrics. These new metrics are:

[4] http://swen.uwaterloo.ca/~ttonelli/mulato/

- ***Lack of Cohesion in Operations* (LCOO-HS)**: metric adapted from the LCOM metric of Henderson-Sellers.
- ***Lack of Cohesion in Operations* (LCOO-BDW)**: metric adapted from the LCOM metric of Briand et al. (Briand, Daly, & Wust, 1998) using the same formalisms depicted in 4.1
- ***Vocabulary Size (VS)*:** counts the number of classes in the system.
- ***Number of Inner components (NIC)*:** counts the number of inner components in the system.

In the Eclipse Metrics plugin[5], the LCOM-CK and LCOM-HS are supported. In these implementations, operations are only included if they access at least one attribute. The MuLATo implementation of LCOO-HS supports this count as well as counting all the operations of a component. The results of the LCOM-HS metric in Java with the Metrics plugin and the MuLATo plugin are the same (when the latter only counts operations that access at least one attribute)

The VS metric was re-developed because the version that was already implemented only gave a result at a system level, not allowing computing the metric for a specific package, for example.

Developing MuLATo is not a trivial task since it required. In addition to knowledge of the metric, a proper understanding of the MuLATo tool and the CaesarJ compiler (which is an extension of the Polyglot (Nystrom, Clarkson, & Myers, 2003) Java compiler) is required. Neither MuLATo nor the CaesarJ compiler has any documentation so the only way to understand them is by examining the source code.

[5] http://metrics2.sourceforge.net/

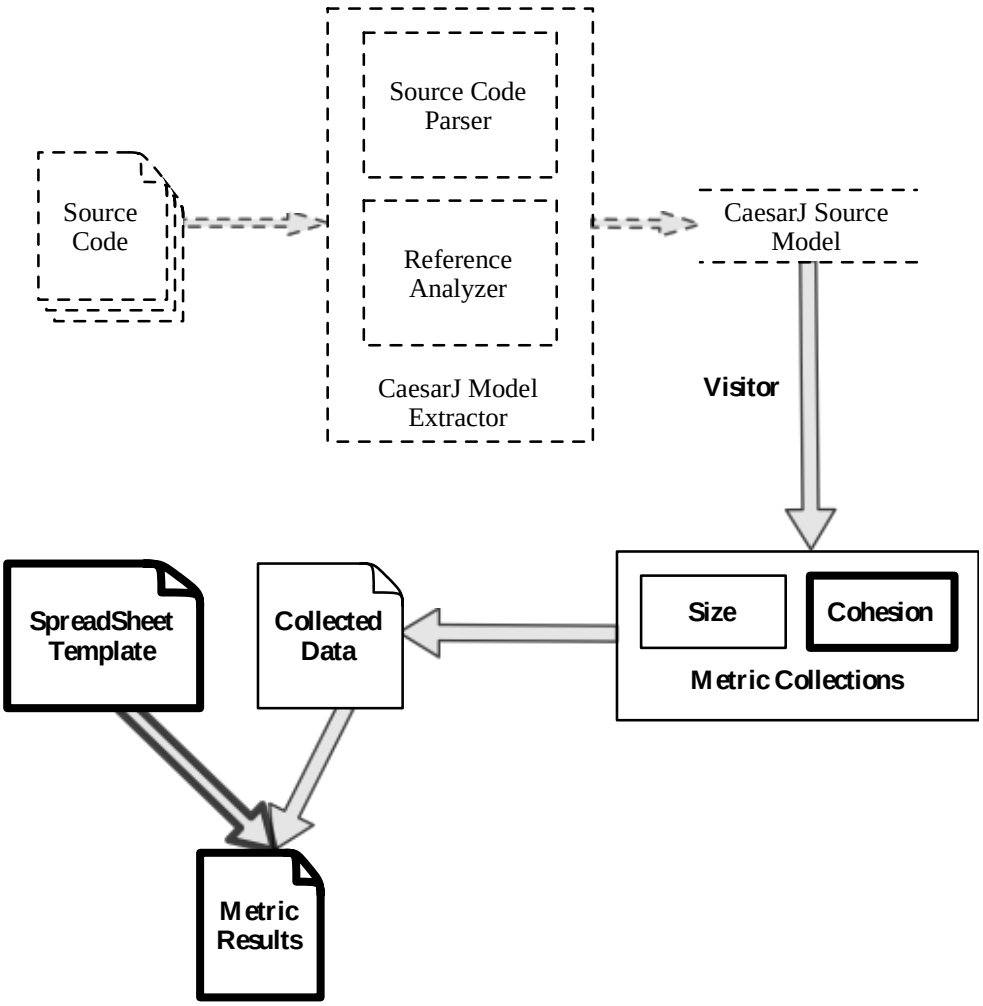

Figure 6. MuLATo metric collecting process

Figure 6 illustrates the metric collecting process after the developments made in the context of this dissertation. The forms with the perforated lines did not need to be altered. The shapes with the normal lines were altered and the ones with the thick lines are new and were not present in the previous versions of MuLATo.

The CaesarJ Model Extractor module takes as input all the source files (classes, interfaces and cclasses) and detects the structure of the analyzed files, in terms of their components (e.g. attributes, pointcuts, operations and statements). The Extractor module parses the CaesarJ code and builds a representative model of the system, called CaesarJ Source Model. This model is a suitable representation of the source code. Two sub-modules comprise the Model Extractor: the Source Code Parser and the References Analyzer. The first sub-module extracts information from the code while the second captures the existing relationships between syntactic elements (e.g. imports, inheritance, associations and method calls).

The Metric Collector module is responsible for computing the metrics.

To retrieve information from a system, it is necessary to navigate the parse tree built by MuLATo (the CaesarJ Source Model) from the subject's source code and collect all the needed values. Next, these are exported to a CSV file where it can be used in conjunction with a spreadsheet "template" file to compute the desired metrics. The use of an Excel template file also gives the freedom to choose which components are to be included (or excluded) from the calculations.

Besides confirming the results of LCOO-HS collected by the tool with manual calculations, another kind of validation was made.

To validate the implementation of the LCOO-HS metric, results were collected from various Java projects and compared with the results of LCOM-HS metric collected by the Metrics plugin. Both results are equal.

6. Evaluating CaesarJ against Java

This chapter presents the study of the size and cohesion software attributes for the Java and CaesarJ languages. Its organization is adapted from the "standard" experimental report structure for Software Engineering (Jedlitschka, Ciolkowski, & Pfahl, 2008).

6.1 Introduction

6.1.1 Research Objectives

The purpose of this study is to **analyse** the CaesarJ language, **for the purpose of** assessing the usefulness of its language constructs (using the Java language as a yardstick), **with respect to** software size and cohesion, **from the point of view of** developers who implemented analogous systems in both the CaesarJ and Java languages, **in the context of** a study on several repositories that includes several examples of functionally equivalent pattern implementations in both Java and CaesarJ.

6.1.2 Context

This study builds on previous work, namely the implementation of the well-known GoF design patterns in CaesarJ developed by Sousa et al (Sousa & Monteiro, 2008) and Braz (Braz, 2009). Six repositories of implementations of the GoF are used: Hannemann & Kiczales' (Hannemann & Kiczales, 2002), Cooper's (Cooper, 1998), Eckel's (Eckel, 2003)**,** Fluffycat's (Truett), Huston's (Huston, 2007), and Polanco's (Polanco, 2002).
We consider results from this study valid only in the context of the patterns' examples used, rather than as applying to software modules in general. Further research must be conducted to assess which conclusions are specific to the implementations used and which are generalizable.

6.2 Background

6.2.1 Related studies

Some studies related to this one have already been made. Garcia et al. (Garcia A. , Sant'Anna, Figueiredo, Kulesza, Lucena, & von Staa, 2006) presented a quantitative study

that compared aspect-based and OO solutions for the 23 Gang-of-Four patterns. They used AspectJ as the AOP langue and Java as the OOP one. They've found that the use of aspects reduced coupling between components and increased cohesion for most solutions. They also discovered, for the size attribute, the results were much better for the aspect-based solutions.
Other related case study is the HealthWatcher, in the context of the TAO research project (Lancaster University, 2007) funded by the Lancaster University for the assessment and comparison of AOSD techniques with existing ones. It was concluded that concerns aspectized upfront tend to show superior modularity stability in the AO designs. Results also showed better coupling and cohesion in AOP. Unfortunately the CaesarJ implementation of the study is very similar to the AspectJ implementation as they share the same class diagram and do not take into account the particular features of CaesarJ.

6.2.2 Relevance to practice

Before adopting a programming language for its alleged benefits, it is important to realize the extent to which this benefits are real and in what circumstances are observed or not. This work aims to do this with a comparative study that sheds some light on the claims of CaesarJ being better Java with respect of modularity and complexity.

6.3 Experimental planning

6.3.1 Goals

The study focuses on the comparison of complexity (which is measured indirectly via size) and cohesion.
For this purpose, this goal is broken down into 6 sub-goals, where the variation lies on the metric under assessment. The sub-goals definitions are depicted in **Table** 3.

Goals	Description
G1	Compare the Java and CaesarJ languages with respect to the *LOC* metric
G2	Compare the Java and CaesarJ languages with respect to the *VS* metric
G3	Compare the Java and CaesarJ languages with respect to the *NOA* metric
G4	Compare the Java and CaesarJ languages with respect to the *NOO* metric
G5	Compare the Java and CaesarJ languages with respect to the *WOC* metric
G6	Compare the Java and CaesarJ languages with respect to the *LCOO-HS* metric

Table 3. Research goals

Goals G1 to G5 contributes to complexity and goal G6 is related to cohesion.

6.3.2 Experimental Units

There are several existing Java repositories of the design patterns. These repositories are available online for free. Some of them are the following:

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 4. Gang-of-Four Design Patterns repositories implemented in Java

The repositories of the CaesarJ implementations are refactorings to CaesarJ of existing examples in originally coded in Java, developed by Sousa et al (Sousa & Monteiro, 2008) and Braz (Braz, 2009) and were developed taking into account the specific features of CaesarJ, including pointcuts, advice, virtual classes and family polymorphism.
Table 5 indicates the available implementations in both Java and CaesarJ. Each "X" represents an implementation of a scenario for a given pattern. There are a total of 51 examples available in both languages.

	Thinking in patterns (Eckel, 2003)	**DP Java companion** (Cooper, 1998)	**Fluffycat** (Truett)	**Hanneman n et al.** (Hanneman n & Kiczales, 2002)	**Huston** (Huston , 2007)	**Guidi Polanco** (Polanco , 2002)
Abstract Factory	X	X	X	X		
Adapter		X		X		
Bridge	X		X	X	X	
Builder	X			X		
Chain of Res-ponsibility		X	X	X		X
Command		X		X		
Composite	X	X		X	X	

Decorator	X		X	X		X
Facade				X		
Factory Method				X		X
Flyweight						
Interpreter				X		
Iterator		X		X		
Mediator		X	X	X		
Memento						
Observer	X	X	X	X		X
Prototype		X	X	X		
Proxy						
Singleton				X		
State						
Strategy						
Template Method		X		X		
Visitor	X	X	X	X	X	

Table 5. Design pattern implementations implemented in CaesarJ

6.3.3 Experimental Material

All 51 implementations from all the repositories are used as subject of this study. This will be done for greater statistical reliability.

6.3.4 Tasks

As noted on the previous sub-section, the subjects of this study are design pattern implementations. As such, the "tasks" item in the experimental design description is not applicable for this study.

6.3.5 Hypotheses

The goals lead to test six different basic hypotheses, in order to assess the effect of CaesarJ on each metric (when compared to Java). The hypotheses were identified as H1, H2, H3, H4, H5 and H6 (Table 6). For each hypothesis, both a null and an alternative hypothesis were formulated.

Hypotheses		
H1	$H1_0$	CaesarJ provides no significant improvement on the patterns' LOC.
	$H1_1$	CaesarJ provides a significant improvement on the patterns' LOC
H2	$H2_0$	CaesarJ provides no significant improvement on the patterns' VS
	$H2_1$	CaesarJ provides a significant improvement on the patterns' VS
H3	$H3_0$	CaesarJ provides no significant improvement on the patterns' NOA
	$H3_1$	CaesarJ provides a significant improvement on the patterns' NOA
H4	$H4_0$	CaesarJ provides no significant improvement on the patterns' NOO
	$H4_1$	CaesarJ provides a significant improvement on the patterns' NOO
H5	$H5_0$	CaesarJ provides no significant improvement on the patterns' WOC
	$H5_1$	CaesarJ provides a significant improvement on the patterns' WOC
H6	$H6_0$	CaesarJ provides no significant improvement on the patterns' LCOO-HS
	$H6_1$	CaesarJ provides a significant improvement on the patterns' LCOO-HS

Table 6. Research hypotheses

6.3.6 Independent variables

The independent variable is the same for all the hypotheses. This variable, which we'll call "Is CaesarJ", assumes the value *true* for pattern instances implemented in CaesarJ and *false* otherwise.

6.3.7 Dependent variables

The variables used in this experiment represent the various metrics collected. These metrics can be applied to both Java and CaesarJ and have already been explained in the previous chapters. They are:

- Lines of Code (LOC)
- Vocabulary Size (VS)
- Number of Attributes (NOA)
- Number of Operations (NOO)
- Weighted operations per component (WOC)
- Lack of Cohesion in Operations (LCOO-HS)

6.3.8 Design

In this case study we have 51 scenarios, each implemented in two different programming languages. We have 2 observations for each subject (i.e. scenario). The first observation is

the metrics collection for the Java scenario. The second observation is the refactorings in CaesarJ.

6.3.9 Procedure

First each Java and CaesarJ repository project was organized to ensure that each pattern implementation of the standards used the same data structures (some CaesarJ examples had multiple implementations of the same patterns using different data structures, e.g. *ArrayLists* and *WeakHashMaps*). Then Metrics[6] tool was used in each class of each of the 102 examples to collect LOC. Afterwards we used the MuLATo tool to collect the remaining metrics. An illustration of this process is in depicted in Figure 6.
This tool returns a CSV file with the metric results for each class. We grouped these results by pattern. After all the values of the pretended metrics for every implementations of each repository are collected, they are converted to the format of SPSS so that we can run the appropriate statistical tests. For each metric, we performed statistical and normality tests to verify data normality. After verifying that non-parametric tests were the most suitable for our samples, we executed the Wilcoxon signed-rank test for each pair of metrics.

6.3.10 Analysis procedure

The following steps were taken:

- **Compute descriptive statistics**: For all the independent and dependent variables, a set of descriptive statistics, (specifically the mean value within the sample, standard deviation, minimum value, maximum value, range, skewness and kurtosis) was collected. These descriptive statistics provide a first overview of the data, which is further detailed in subsequent analyses.
- **Normality tests**: Data is checked for normality, so that the statistics tests that are suitable for the data and our experimental design
- **Analysis of differences between groups**: Finally, a test to detect whether there are significant differences between groups is executed. This allows the test of the hypotheses stated in sub-section 6.3.5.

The first two steps are depicted in section 6.6. Then the analysis of differences between groups are showed in section 6.8

6.4 Execution

6.4.1 Preparation

[6] http://metrics2.sourceforge.net/

The implementations of CaesarJ were divided into different projects, organized by repository and each implementation was compared to the its Java equivalent because some CaesarJ implementations had multiple alternatives (for instance, one example had various alternatives for a list implementation like an *ArrayList*, *HashMap* or a custom class).
No other special preparations were required, other than installing the version of the MuLATo plugin developed in this dissertation context and the Metrics plugin.

6.4.2 Deviations

Data collection followed the plan outlined in sub-section 6.3.9.

6.5 Analysis

The results of the data collection are depicted in Table 7. The design patterns are grouped in the two rightmost columns by repository. The collected metrics are the ones listed in section 6.3.7. The first 5 metrics have already been explained in chapter 3.2.1 and the LCOO-HS in chapter 4.

		LOC		VS		NOA		NOO		WOC		LCOO-HS	
Repository	**Design Pattern**	**Java**	**CaesarJ**	**Java**	**CaesarJ**	**Java**	**CaesarJ**	**Java**	**CaesarJ**	**Java**	**CaesarJ**	**Java**	**CaesarJ**
HK	**Abstract Factory**	145	139	7	7	5	5	16	14	3,571	3,140	0,000	0,000
	Adapter	35	34	4	4	3	1	5	5	2,500	2,500	0,000	0,000
	Bridge	98	125	7	35	1	1	17	19	3,714	0,770	0,000	0,000
	Builder	72	87	4	9	3	4	13	16	6,000	3,000	0,125	0,060
	Chain of Responsibility	108	155	6	26	4	2	12	25	3,667	1,880	0,000	0,000
	Command	68	82	5	16	1	2	7	8	2,000	0,690	0,000	0,040
	Composite	92	135	4	24	5	5	22	22	9,000	1,375	0,213	0,010
	Decorator	48	120	6	4	1	0	9	11	3,000	5,250	0,000	0,000
	Facade	55	24	5	3	0	0	9	5	3,000	2,670	0,000	0,000
	Factory Method	65	85	4	12	1	3	8	10	2,250	1,080	0,000	0,000
	Interpreter	152	152	9	13	8	9	29	29	7,222	4,540	0,000	0,000
	Iterator	74	79	4	7	3	3	15	15	6,000	3,140	0,063	0,040
	Mediator	69	110	5	19	5	5	8	17	3,000	1,320	0,000	0,000
	Observer	116	174	5	27	6	5	21	36	7,600	2,296	0,250	0,025
	Prototype	66	105	3	15	2	2	9	13	4,667	1,200	0,000	0,000
	Singleton	76	64	3	6	6	3	8	8	3,000	2,170	0,278	0,000
	Template Method	55	55	4	4	0	0	11	11	5,500	5,500	0,000	0,000
	Visitor	104	118	7	19	5	5	18	18	4,429	1,680	0,071	0,030
JCooper	**Abstract Factory**	238	247	11	15	14	15	33	33	4,273	3,200	0,222	0,106
	Adapter	134	140	3	4	9	9	13	13	6,000	4,500	0,488	0,366
	Chain of Responsibility	313	361	9	34	23	14	34	37	6,444	1,971	0,441	0,034
	Command	86	97	8	11	15	8	14	9	2,875	1,364	0,000	0,000
	Composite	198	273	4	23	19	26	20	34	9,000	2,174	0,380	0,062
	Iterator	200	204	6	10	11	12	19	21	5,000	3,200	0,222	0,136
	Mediator	399	480	13	33	38	35	48	71	5,769	3,303	0,522	0,156
	Observer	166	260	8	24	17	18	16	33	3,375	2,208	0,358	0,118
	Prototype	208	238	4	15	17	18	19	22	6,250	1,933	0,585	0,108
	Template Method	136	136	6	6	13	13	16	16	7,500	7,500	0,306	0,306
	Visitor	184	193	7	19	13	12	26	25	6,714	2,421	0,202	0,071
Beckel	**Abstract Factory**	77	64	11	14	7	0	17	14	2,000	1,429	0,121	0,000
	Bridge	143	168	9	37	3	3	34	31	4,222	0,919	0,000	0,000
	Builder	110	130	14	27	6	6	23	23	2,429	1,259	0,000	0,000
	Composite	42	106	4	23	2	3	9	19	3,000	1,217	0,000	0,000
	Decorator	124	128	10	16	12	12	27	28	3,500	1,813	0,500	0,313
	Observer	204	201	14	21	23	10	30	33	3,714	2,714	0,241	0,036
	Visitor	97	104	9	23	3	2	18	12	3,556	0,783	0,000	0,000

		LOC		VS		NOA		NOO		WOC		LCOO-HS	
Repository	**Design Pattern**	**Java**	**CaesarJ**	**Java**	**CaesarJ**	**Java**	**CaesarJ**	**Java**	**CaesarJ**	**Java**	**CaesarJ**	**Java**	**CaesarJ**
FluffyCat	**Abstract Factory**	139	216	16	35	3	2	36	24	2,875	0,971	0,031	0,014
	Bridge	113	117	9	37	2	2	20	15	2,444	0,486	0,000	0,000
	Chain of Responsibility	177	241	5	22	8	8	36	47	10,800	3,409	0,417	0,084
	Decorator	69	69	4	8	3	3	9	9	3,000	1,250	0,167	0,000
	Mediator	149	185	6	23	10	9	24	41	7,167	2,870	0,444	0,126
	Observer	239	323	4	24	9	11	26	45	14,250	3,833	0,413	0,074
	Prototype	90	132	7	19	4	4	13	15	2,571	1,053	0,071	0,026
	Visitor	129	167	8	29	5	8	27	27	6,500	1,690	0,083	0,057
Polanco	**Chain of Responsibility**	74	154	7	28	1	1	8	24	2,286	1,750	0,000	0,000
	Decorator	85	84	6	9	4	3	17	18	4,000	2,444	0,083	0,056
	Factory Method	83	87	7	12	4	4	14	11	3,429	1,750	0,071	0,042
	Observer	69	148	4	22	4	5	7	24	3,250	1,955	0,000	0,000
Vhuston	**Bridge**	120	200	6	36	8	8	16	33	3,500	1,167	0,042	0,007
	Composite	157	168	14	30	13	4	28	28	3,286	1,400	0,107	0,000
	Visitor	260	105	25	18	5	6	56	19	4,120	1,667	0,027	0,074

Table 7. Values of collected metrics

6.6 Descriptive statistics

Descriptive statistics are used to describe the basic features of the data in the study. They provide simple summaries about the sample and the measures.
For each variable, we present in Table 8 the following statistics:

- **Number of cases:** count of the total of observations in each series.
- **Mean value within the sample**: sum of observations divided by the number of observations in the series. It is commonly used to describe the central tendency of variables.
- **Standard deviation**: measure of dispersion that is calculated based on the values of the data. It allows us to see how widely the data are dispersed around the mean.
- **Minimum value**: smaller value of the observations.
- **Maximum value**: higher value of the observations.
- **Range**: it is calculated by subtracting the smallest observation (minimum value) from the greatest (maximum value) and provides an indication of statistical dispersion
- **Skewness**: measure of whether the peak is centred in the middle of the distribution. A positive value means that the peak is off to the left, and a negative value suggests that it is off to the right.
- **Kurtosis**: measure of the extent to which data are concentrated in the peak versus the tail. A positive value indicates that data are concentrated in the peak; a negative value indicates that data are concentrated in the tail.

	N	Range	Minimum	Maximum	Mean		Std. Deviation	Variance	Skewness		Kurtosis	
	Statistic	Statistic	Statistic	Statistic	Statistic	Std. Error	Statistic	Statistic	Statistic	Std. Error	Statistic	Std. Error
LOC Java	51	364	35	399	127,6471	10,14715	72,46512	5251,193	1,55	0,333	3,121	0,656
LOC CJ	51	456	24	480	152,3333	11,82615	84,45559	7132,747	1,58	0,333	3,793	0,656
VS Java	51	22	3	25	7,2549	0,5711	4,07845	16,634	2,088	0,333	6,207	0,656
VS CJ	51	34	3	37	18,7647	1,39764	9,98116	99,624	0,172	0,333	-0,953	0,656
NOA Java	51	38	0	38	7,5882	1,01331	7,23651	52,367	1,98	0,333	5,297	0,656
NOA CJ	51	35	0	35	6,8431	0,94532	6,75092	45,575	2,079	0,333	5,744	0,656
NOO Java	51	51	5	56	19,4118	1,51001	10,78365	116,287	1,223	0,333	1,825	0,656
NOO CJ	51	66	5	71	22,2745	1,74088	12,43234	154,563	1,409	0,333	3,427	0,656
WOC Java	51	12,25	2	14,25	4,6906	0,3436	2,4538	6,021	1,692	0,333	3,693	0,656
WOC CJ	51	7,01	0,49	7,5	2,2707	0,19255	1,37511	1,891	1,609	0,333	3,403	0,656
LCO-HS Java	51	0,59	0	0,59	0,1479	0,02487	0,17764	0,032	0,992	0,333	-0,326	0,656
LCO-HS CJ	51	0,37	0	0,37	0,0505	0,01151	0,08218	0,007	2,471	0,333	6,362	0,656

Table 8. Descriptive statistics of the metrics

To decide whether it is appropriate to use parametric tests, we need to check if the variables have a normal distribution.

An important detail to consider is that the variables are being compared *in pairs*. Therefore, if one element of the pair is not normal, it is safest to assume that the data from the pair is not normal.

Positive skewness indicates an asymmetric distribution, with a higher frequency of the variable's lower values. In other words, the distribution is right-skewed. This contrasts with the normal distribution, which is symmetric and should therefore exhibit a skewness of 0, providing a hint on the non-normality of the data. Further tests are used to confirm the non-normality of this variable.

Table 9 presents results of two such tests: the Kolmogorov- Smirnov with the Lilliefors correction and the Shapiro-Wilk's normality tests. The former is the most widely used test and adequate for this sample size. The latter is often used with smaller samples, and used here for confirmation purposes only. The null hypothesis, for each of the tests, is that the sample comes from a normal distribution. The alternative is that the sample comes from a non-normal distribution.

		Kolmogorov-Smirnov[a]			**Shapiro-Wilk**		
Metric	**Language**	Statistic	df	Sig.	Statistic	df	Sig.
LOC	**Java**	0,114	51	0,098	0,875	51	0
	CaesarJ	0,16	51	0,002	0,897	51	0
VS	**Java**	0,192	51	0	0,804	51	0
	CaesarJ	0,075	51	,200*	0,95	51	0,031
NOA	**Java**	0,195	51	0	0,813	51	0
	CaesarJ	0,165	51	0,001	0,81	51	0
NOO	**Java**	0,125	51	0,044	0,906	51	0,001
	CaesarJ	0,123	51	0,053	0,905	51	0,001
WOC	**Java**	0,184	51	0	0,842	51	0
	CaesarJ	0,151	51	0,005	0,867	51	0
LCOO	**Java**	0,211	51	0	0,808	51	0
	CaesarJ	0,276	51	0	0,633	51	0

a. Lilliefors Significance Correction

*. This is a lower bound of the true significance.

Table 9. Normality tests

For values of significance less than 0.05, the data is not considered normal. These values confirm the non-normality of the sample, as there is no pair in that both implementations

have a value above 0.05. As such, non-parametric procedures will be used for testing the hypotheses.

6.7 Data set reduction

No experimental units were removed from the sample.

6.8 Hypotheses testing

The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test for the case of two-sample designs involving repeated measures or "before" and "after" measures. In this case, the "before" case is the implementation of the design patterns in Java and the "after" is the implementations in CaesarJ.

		N	Mean Rank	Sum of Ranks	
LOC (CaesarJ) – LOC (Java)	Negative Ranks	8[a]	15,19	121,5	a. LOCCJ < LOCJ
	Positive Ranks	39[b]	25,81	1006,5	b. LOCCJ > LOCJ
	Ties	4[c]			c. LOCCJ = LOCJ
	Total	51			
VS (CaesarJ) - VS (Java)	Negative Ranks	3[d]	7,17	21,5	d. VSCJ < VSJ
	Positive Ranks	44[e]	25,15	1106,5	e. VSCJ > VSJ
	Ties	4[f]			f. VSCJ = VSJ
	Total	51			
NOA (CaesarJ) - NOA (Java)	Negative Ranks	16[g]	17,94	287	g. NOACJ < NOAJ
	Positive Ranks	14[h]	12,71	178	h. NOACJ > NOAJ
	Ties	21[i]			i. NOACJ = NoAJ
	Total	51			
NOO (CaesarJ) - NOO (Java)	Negative Ranks	11[j]	17,86	196,5	j. NOOCJ < NOOJ
	Positive Ranks	26[k]	19,48	506,5	k. NoOCJ > NoOJ
	Ties	14[l]			l. NOOCJ = NOOJ
	Total	51			
WOC (CaesarJ) -	Negative Ranks	47[m]	24,45	1149	m. WOCCJ < WOCJ

WOC (Java)	Positive Ranks	1^{n}	27	27	n. WOCCJ > WOCJ
	Ties	3^{o}			o. WOCCJ = WOCJ
	Total	51			
LCOCJ - LCOJ	Negative Ranks	29^{p}	16,52	479	p. LCOCJ < LCOJ
	Positive Ranks	2^{q}	8,5	17	q. LCOCJ > LCOJ
	Ties	20^{r}			r. LCOCJ = LCOJ
	Total	51			

Table 10. Wilcoxon signed-rank test ranks

	H1	H2	H3	H4	H5	H6
	LOCCJ - LOCJ	VSCJ - VSJ	NOACJ - NOAJ	NOOCJ - NOOJ	WOCCJ - WOCJ	LCOCJ - LCOJ
Z	$-4,683^{a}$	$-5,743^{a}$	$-1,147^{b}$	$-2,342^{a}$	$-4,683^{a}$	$-5,743^{a}$
Asymp. Sig. (2-tailed)	0	0	0,252	0,019	0	0

a. Based on negative ranks.
b. Based on positive ranks.
c. Wilcoxon Signed Ranks Test

Table 11. Wilcoxon signed-rank test statistics

The Wilcoxon signed-rank test rank results are summarized in Table 10 and the test statistics are depicted in Table 11. The asymptotic significance is the probability that the differences between Java and CaesarJ are by chance (the smaller the value, the more this difference is unlikely to be casual).

It can be observed that for hypotheses H1, H2, H5 and H6, the null hypothesis can be rejected with $p < 0,01$. Hypothesis H4 can also be rejected with $p < 0,05$. For hypothesis H3, no significant differences were found, so it can be rejected. In other words, for all hypotheses except H3, significant differences in metrics values were found for CaesarJ instances when comparing with their Java counterparts.

6.9 Discussion

6.9.1 Interpretation of results

The *Lines of Code* (LOC) metric has a significant difference between Java and CaesarJ. 39 of the 51 implementations in CaesarJ have a higher LOC than Java. *Vocabulary Size* (VS) also has a noteworthy noticeable between Java and CaesarJ with 44 implementations in CaesarJ with greater VS than Java. With respect of *Number of Attributes* (NOA), no significant differences were found. The *Number of operations* (NOO) metric also had a significant difference. In the 51 implementations, 26 have lower value in Java and 14 have the same value. By themselves, one cannot take many conclusions about theses metrics. But studying them together can yield some interesting conclusions.
Figure 7 illustrates with a graphic the measurement values of LOC for CaesarJ and Java (lower values are better).

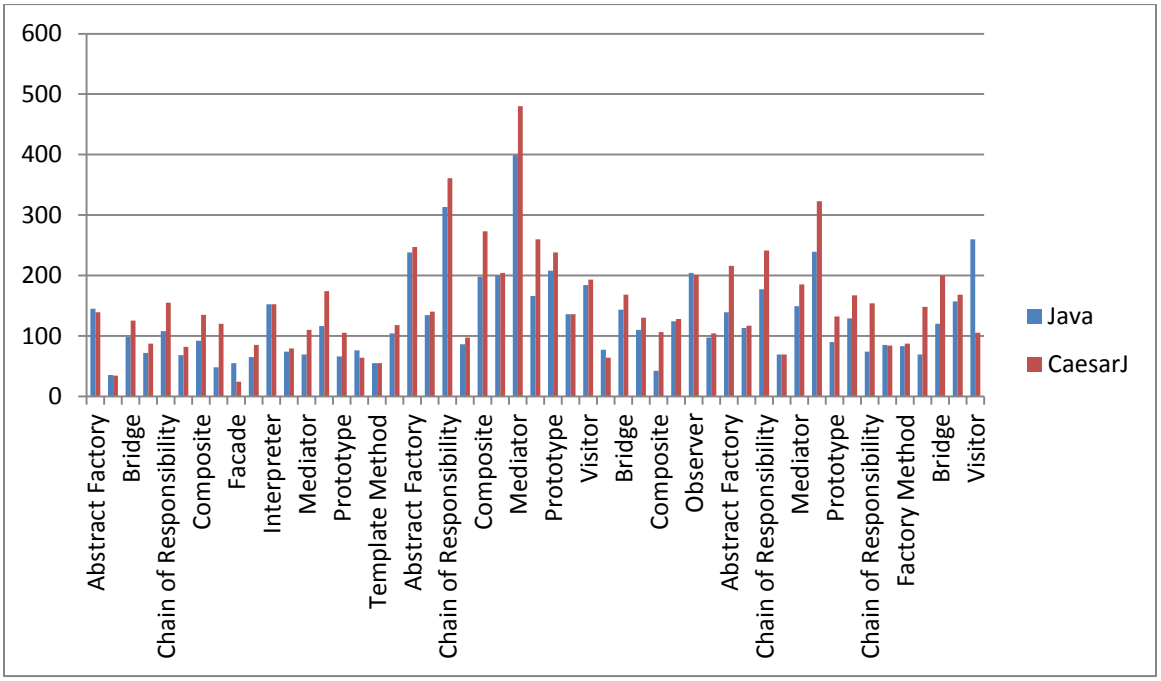

Figure 7. Measurement values of Lines of Code for Java and CaesarJ

From 51 design patterns:

- 8 (16%) Java implementations have a higher value of LOC.
- 39 (76%) CaesarJ implementations have a higher value of LOC.
- 4 (8%) cases have the same value of LOC.

In Figure 8 are the measurement values for the VS metric (lower values are better).

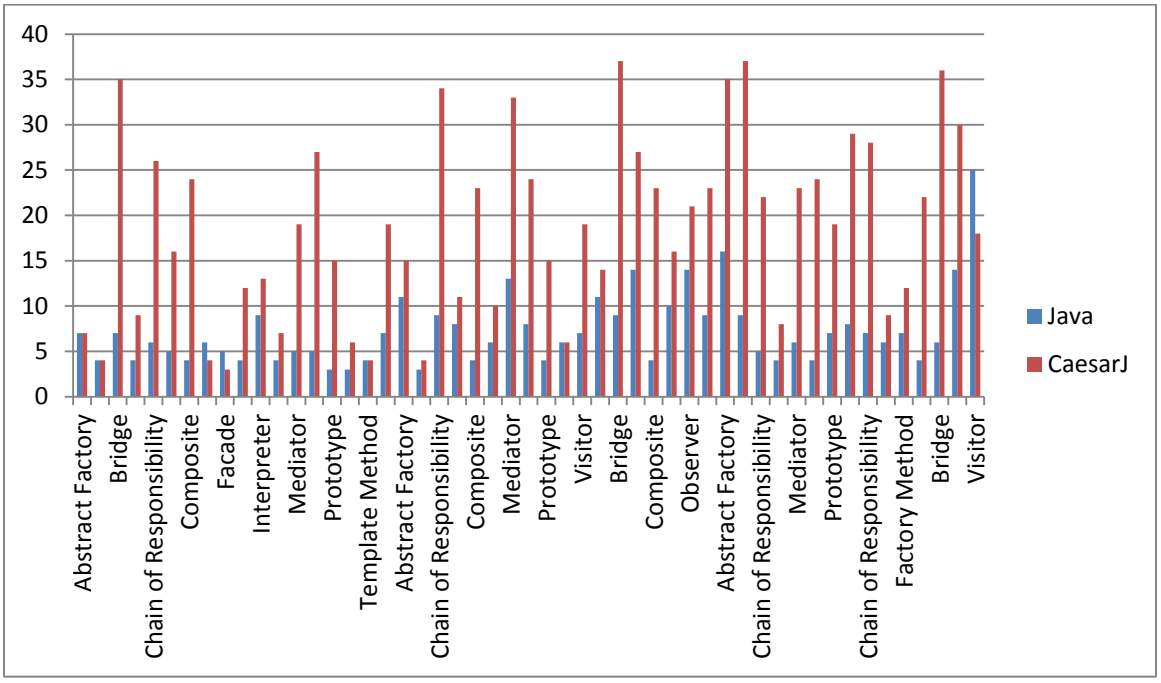

Figure 8. Measurement values of Vocabulary Size for Java and CaesarJ

Of the 51 design patterns:

- 3 (6%) Java implementations have a higher value of VC.
- 44 (86%) CaesarJ implementations have a higher value of VC.
- 4 (8%) cases have the same value of VC.

Besides the majority of CaesarJ implementations have a larger value than the ones in Java, these implementations have an increase of 322%.

Figure 9 illustrates the measurement values of NOA for Java and CaesarJ (lower values are better).

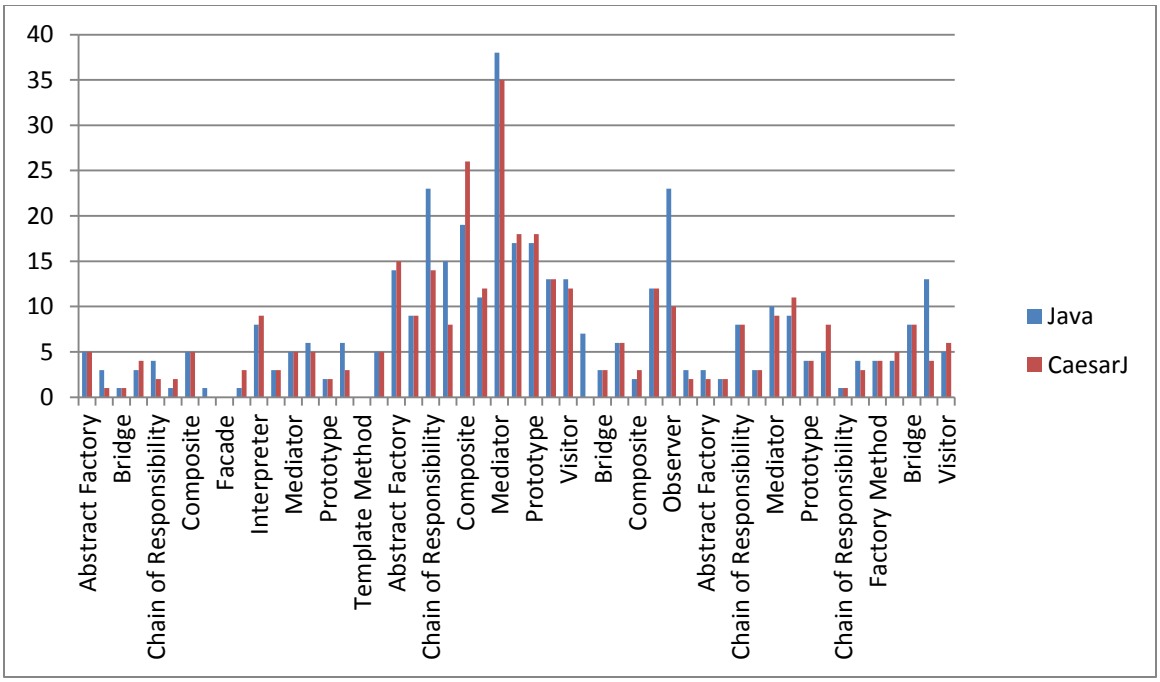

Figure 9. Measurement values of Number of Attributes for Java and CaesarJ

From the 51 design patterns:

- 16 (31%) Java implementations have a higher value of NOA.
- 14 (27%) CaesarJ implementations have a higher value of NOA.
- 21 (41%) cases have the same value of NOA.

In Figure 10 are the measurement values for the NOO metric (lower values are better).

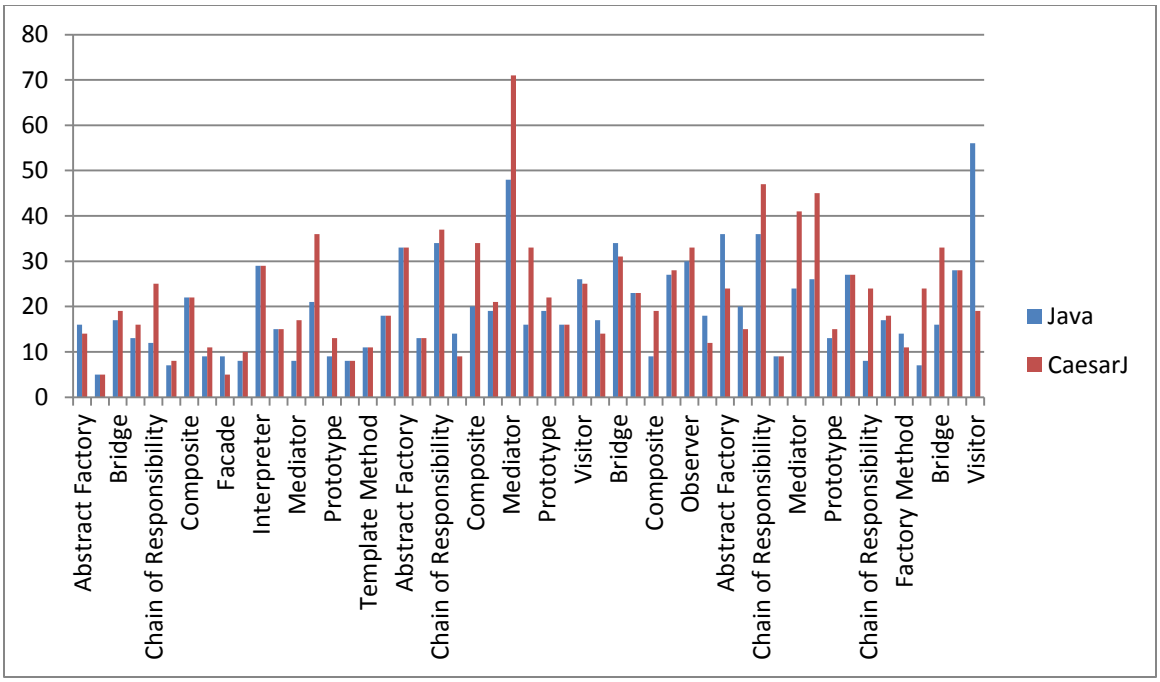

Figure 10. Measurement values of Number of Operations for Java and CaesarJ

From 51 design patterns:

- 11 (22%) Java implementations have a higher value of NOO.
- 26 (51%) CaesarJ implementations have a higher value of NOO-
- 14 (27%) cases have the same value of NOO.

In Figure 11 are the measurement values for the WOC metric (lower values are better).

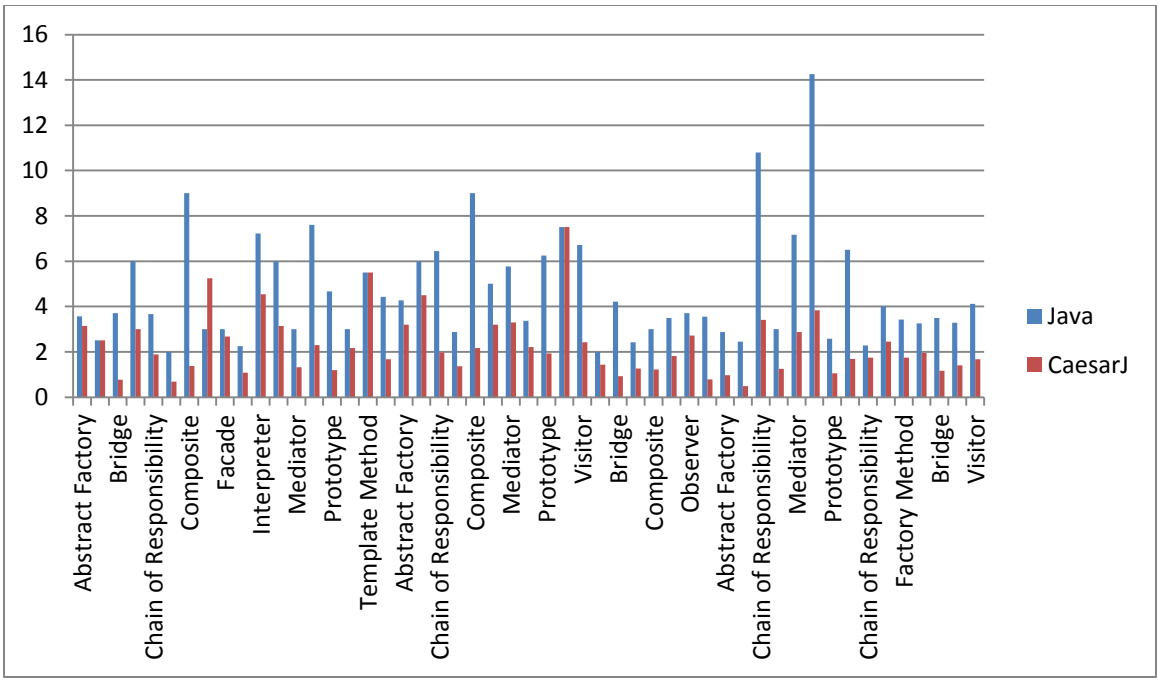

Figure 11. Measurement values of Weighted Operations per Component for Java and CaesarJ

From 51 design patterns:

- 47 (92%) Java implementations have a higher value of WOC
- 1 (2%) CaesarJ implementations have a higher value of WOC
- 3 (6%) cases have the same value of WOC

The Java implementations that have a higher WOC show an average increase of 258% in this value.

Finally Figure 12 depicts the measurement values for LOO-HS (lower values are better).

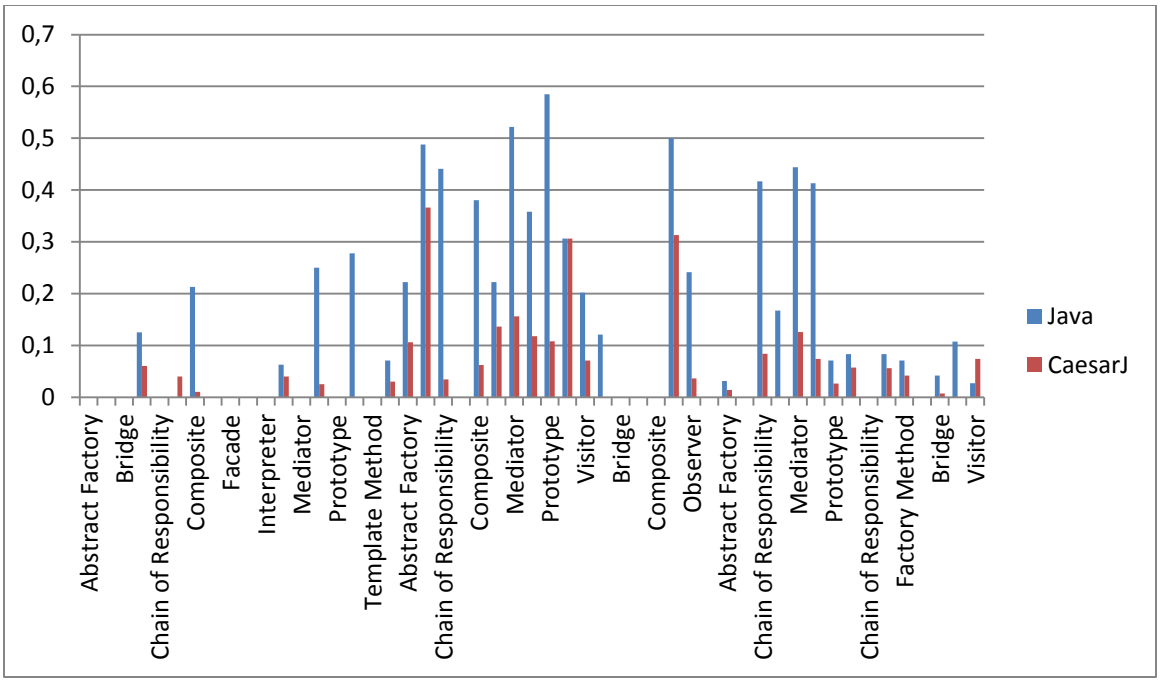

Figure 12. Measurement values of LCOO-HS for Java and CaesarJ

From 51 design patterns:

- 29 (57%) Java implementations have a higher value of LCOO-HS
- 2 (4%) CaesarJ implementations have a higher value of LCOO-HS
- 20 (39%) cases have the same value of LCOO-HS

Most of CaesarJ implementations have higher LOC, VS and NOO. This means that the size of the CaesarJ implementations is consistently bigger than the Java ones.

This can be justified by the usage of the "aspect component" in CaesarJ. CaesarJ classes are designed to be used via inheritance and CaesarJ promotes the use of CaesarJ classes with various inner classes. This increases the VS, as many CaesarJ classes do not have any operations or attributes in them, having the purpose of serving as a container for one or more inner classes.

The fact that the examples used are of small dimension may bias the results against CaesarJ, since the "aspect component" is aimed to enhance reuse. Many CaesarJ implementations generalize the overall pattern behaviour so that the "aspect component" can be reuse and shared among multiple pattern instances. With larger examples, these differences in size should decrease or even be reversed.

Interestingly, for the *Weighted Operations per Component* (WOC) metric, 49 implementations in CaesarJ have a superior value to the ones in Java. This indicates a much lower complexity in CaesarJ when compared with Java. This makes sense, since crosscutting concerns became encapsulated in the "aspect component".

Lack of cohesion in operations (LCOO-HS) measures the degree to which operations within a component are related to one another in terms of shared variables. In the 51 implementations of the design patterns, 29 have better cohesion in CaesarJ and 2 have better cohesion in Java. The remaining 20 implementations remained with the same value. The statistical tests confirmed a significant difference between the implementations. Since high cohesion is a desirable feature of a modular design, this could indicate a advantage on the part of CaesarJ.

6.9.2 Limitations and threats to validity

The limited size and complexity of the examples used in the implementations may restrict the extrapolation of our results. In addition, this assessment is restricted to the specific pattern instances at hand. Although this study involves multiple repositories, not all 23 Gang-of-Four patterns are implemented. Some patterns have more implementations than others, which could benefit (or impair) the results of one language over another (Table 5).

The cohesion metric also has some limitations. Since LCOO-HS is an extension of LCOM-HS, it inherits its problems. This metric doesn't count indirect connections between operations and attributes (Briand, Daly, & Wust, 1998). For instance, an access method provides read or write access to an attribute of the class. Access methods generally reference just the attribute they provide access to. Thus, other operations of the same class that use this access method also access the attribute. This situation is not covered by the metric and actually can artificially yield a lower cohesion value. It also doesn't take into account the direct connections between methods.
Other limitations in LCOO-HS (and LCOM-HS) are their inability to differentiate the cohesion degree in some components (Chae, Kwon, & Bae, 2000). For example, Figure 13 represents two components A and B. The rectangle and oval shapes represent an operation and attribute respectively and an edge symbolizes an interaction between them. Intuitively, component B should be more cohesive than component A but the LCOO-HS value for both of them is the same.

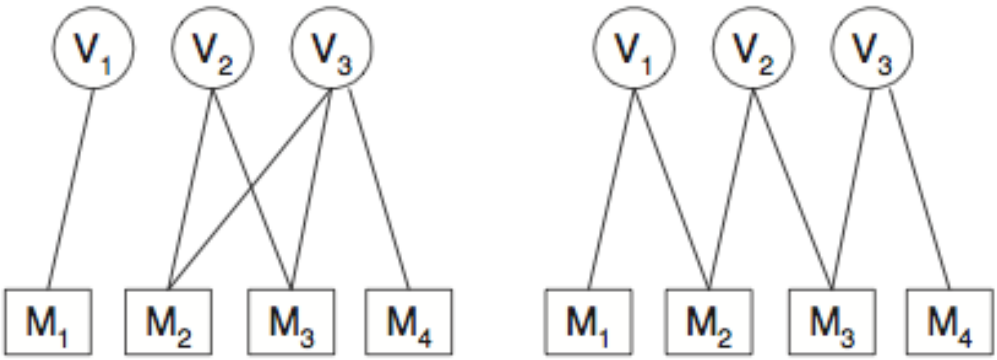

Figure 13. Example of cohesion degree by Chae et al.

6.9.3 Inferences

The analysis performed in this observational study should hold for implementations of similar characteristics (in particular their complexity). Extrapolating these results to larger implementations should be performed with caution, as discussed in the previous section.

7. Related work

7.1 Quantitative study of Design Patterns in Java and AspectJ by Garcia et al.

Garcia et al. (Garcia A. , Sant'Anna, Figueiredo, Kulesza, Lucena, & von Staa, 2006) presented a quantitative study that compared aspect-based and OO solutions for the 23 Gang-of-Four patterns implemented by Hannemann and Kiczales in Java and AspectJ (Hannemann & Kiczales, 2002).
This study was based on popular attributes used in software engineering: Separation of Concerns, Coupling (3.1.2), Cohesion (3.2.2) and Size. The metrics used for measuring these attributes are the same from (Sant'Anna, Garcia, Chavez, Pereira de Lucena, & von Staa, 2003).
Design Patterns usually assign roles to their participants. Hannemann and Kiczales's study identified two roles called defining and superimposed. *Defining roles* are roles in which the participant class has no functionality outside the pattern. A *superimposed role* can be assigned to participant classes that can have functionality outside of the pattern.

The results of the study for each attribute are depicted in Table 12, Table 13 and Table 14. For each attribute there is a metric result for each implementation and in the rightmost column, the superior solution according to the authors. The last line of each table also counts how many patterns in each implementation was superior with respect to each metric (3 first cells), and in general terms (last cell).

Separation of Concerns: as showed in Table 12, 14 of the 23 patterns implementations confirmed superior results in the metrics of SoC (lower values are better). 3 patterns showed similar results in both implementations. To evaluate scalability in this system the authors changed some functionality in both implementations and used the CDLOC metric as a main mechanism to assess scalability. If the CDLOC after the change increases, the authors consider the system not scalable; if it stays the same, the system is scalable.

	CDC		CDO		CDLOC		Scalability		Superior
Design Pattern	**OO**	**AO**	**OO**	**AO**	**OO**	**AO**	**OO**	**AO**	**Solution**
Abstract Factory	14	16	35	35	34	34	no	no	OO
Adapter#	8	7	30	22	32	16	no	yes	AO+
Bridge	12	13	24	26	16	16	no	no	OO
Builder	9	10	29	30	8	8	yes	yes	OO
CoR#	9	3	15	21	50	4	no	yes	AO
Command#	17	11	23	16	38	21	no	yes	AO
Composite#	18	9	149	28	70	48	no	no	AO
Decorator#	18	8	31	8	38	6	no	yes	AO+
Façade	Same implementations for Java and AspectJ								
Factory Method	14	16	23	23	18	18	no	no	OO
Flyweight#	10	13	10	12	20	26	no	no	OO
Interpreter	13	13	26	26	38	38	no	no	=
Iterator#	10	6	20	20	18	14	no	no	AO
Mediator#	13	5	18	6	36	10	no	yes	AO
Memento#	11	10	23	24	44	40	no	no	AO
Observer#	14	9	49	9	92	20	no	yes	AO
Prototype#	7	3	7	2	30	8	no	yes	AO+
Proxy#	11	11	38	19	8	2	no	yes	AO+
Singleton#	6	6	6	1	6	2	yes	yes	AO+
State#	10	10	78	78	30	30	no	no	=
Strategy#	14	12	20	17	18	16	no	no	AO
Template Method#	15	16	24	24	20	20	no	no	OO
Visitor#	20	9	50	23	34	14	no	yes	AO+
Success Total	**6 vs. 12**		**5 vs. 11**		**1 vs. 14**		**2 vs. 11**		**6 vs. 14**

(#) indicates that the design pattern contains one or two superimposed roles
(+) indicates the AO solutions that achieved the best results

Table 12. Overall Results for Separation of Concerns by Garcia et al.

Coupling and Cohesion: The use of aspects reduced coupling between components and increased cohesion for most solutions as can be seen in Table 13 (low values are desired in this table). The only cases where this did not occur with the use of aspects were when the implemented patterns had roles that were not very interactive.

	CBC		DIT		LCOO		Superior
Design Pattern	**OO**	**AO**	**OO**	**AO**	**OO**	**AO**	**Solution**
Abstract Factory	37	44	7	7	1	1	OO
Adapter#	5	5	2	1	-	-	AO
Bridge	17	18	2	2	0	0	OO
Builder	2	3	2	2	12	6	OO
CoR#	29	28	2	2	1	13	OO
Command#	21	34	7	7	3	4	OO
Composite#	47	23	2	2	463	82	AO
Decorator#	3	14	3	1	0	0	AO
Façade	Same implementations for Java and AspectJ						
Factory Method	22	24	2	2	3	0	OO
Flyweight#	11	17	2	2	0	1	OO
Interpreter	17	23	5	5	0	0	OO
Iterator#	12	13	2	2	0	0	=
Mediator#	41	34	2	2	5	1	AO
Memento#	13	18	1	2	0	0	OO
Observer#	45	40	2	2	80	30	AO
Prototype#	7	13	2	2	0	0	OO
Proxy#	11	39	2	2	0	0	AO
Singleton#	11	22	2	2	5	0	AO
State#	17	10	2	2	106	93	AO
Strategy#	18	32	2	2	-	-	OO
Template Method#	2	3	2	2	-	-	OO
Visitor#	41	28	2	2	27	2	AO
Success Total	**15 vs. 6**		**1 vs. 2**		**3 vs. 8**		**12 vs. 9**

(#) indicates that the design pattern contains one or two superimposed roles

Table 13. Overall Results for Coupling and Cohesion by Garcia et al.

Size: for this attribute the results were much better for the aspect-based solutions. As Table 14 illustrates, 12 patterns had less number of operations and respective parameters that their OO counterparts. The number of attributes also reduced with the use of aspects in 10 patterns.

	NOA		WOC		LOC		Superior
Design Pattern	**OO**	**AO**	**OO**	**AO**	**OO**	**AO**	**Solution**
Abstract Factory	9	9	37	41	231	265	OO
Adapter#	3	1	34	32	67	61	AO
Bridge	1	1	40	44	156	161	OO
Builder	7	7	50	51	168	177	=
CoR#	8	2	40	64	213	234	=
Command#	6	4	26	29	198	206	=
Composite#	19	12	169	63	501	283	AO
Decorator#	1	0	34	16	88	69	AO
Façade	Same implementations for Java and AspectJ						
Factory Method	1	1	17	17	135	146	=
Flyweight#	7	7	30	36	119	132	OO
Interpreter	14	14	99	99	216	219	=
Iterator#	9	9	50	53	164	163	=
Mediator#	21	17	51	40	253	253	AO
Memento#	6	6	32	31	128	179	OO
Observer#	26	21	134	117	363	265	AO
Prototype#	6	6	38	33	142	147	AO
Proxy#	9	3	105	38	248	190	AO
Singleton#	30	26	25	21	238	251	AO
State#	13	20	164	110	367	374	=
Strategy#	5	1	62	58	251	264	AO
Template Method#	0	0	46	46	125	128	=
Visitor#	13	13	105	57	289	222	AO
Success Total	**1 vs. 10**		**7 vs. 12**		**14 vs. 7**		**4 vs. 10**

(#) indicates that the design pattern contains one or two superimposed roles

Table 14. Overall Results for Size Measures by Garcia et al.

Garcia et al. noticed that several patterns with superimposed roles were better modularized in the AOP solution and had better results with separation of concerns over operations and lines of code. However, this could not be supported (or refuted) with this empirical study because the data collected was not conclusive enough. Still, the authors concluded, as can be seen in the TemplateMethod analysis in Table 12 that the OO solution is better than the AOP, even though only the CDC is different between them and only by one unit (CDC is 15 for OO and 16 for the AO) and the rest of the metrics have the same values.

Also, the authors of this study evaluated the columns without regard of the weight they have (some metrics can have a bigger weight than others) and considering that these metrics are completely independent from each other, which has not been proved.

7.2 Analysis of modularity in aspect oriented design by Lopes et al.

Another study of aspects that has a markedly different approach from the previous one was made by Lopes et al. (Lopes & Bajracharya, 2005). This study analysed the modularity of an aspect-oriented design in comparison to an object-oriented one. The subject of the study was a web application called *WineryLocator* that uses mostly web services to locate wineries in California, given a street addresses or a city or zip code. It also takes preferences for the wineries and calculates a route for a tour that matches the user's criteria.

The Design Structure Matrix (DSM) was used as the analysis and modelling tool that represents the design structures of the system. This matrix takes the design parameters of the system and represents the interdependences between them. Design parameters are the attributes of the artefact that govern the variation in design (in this case classes, and interfaces).

Then, six modular operations were used to make design changes. These operations are Splitting, Substitution, Augmentation, Exclusion, Inversion, and Porting.

After the new matrix is completed, the authors defined Net Option Values (NOV) expressions to evaluate and compare them. The NOV model is a mathematical model that quantifies the value of a modular design. The NOV had a high increase after the aspect-oriented modularization.

Lopes et al. observed that the use of aspects increased the modularity of the system, even if this system already has a modularized design. They concluded that DSMs were capable of modelling dependencies in an aspect-based system's design without any change of the DSM's basic model. The authors also concluded that design changes can be expressed in terms of the modular operators and the NOV analysis can be used to compare the system's design with other alternatives.

One important limitation of this study is that some of the assumptions the authors did with the NOV expressions lack empirical validation. The NOV gives a quantitative dimension to the study

7.3 TAO – A Testbed for Aspect Oriented Software Development Project

The TAO[7] project, a testbed for Aspect Oriented Software Development, is research project funded by the Lancaster University for the assessment and comparison of AOSD techniques with existing ones in terms of rigorous qualities, such as modularity, reusability, and maintainability. The specific aims of this project are:

1) Design of the testbed by identifying assessment issues to be explored in the software development phases, such as requirements engineering, architecture design, implementation, and quality assurance.

[7] http://www.comp.lancs.ac.uk/research/projects/project.php?pid=215

2) End-to-end realization of a major case study, such as a context-sensitive tourist guide system, to form part of the testbed suite of studies.
3) Exploitation of the case study to identify candidate points of integration between our AOSD techniques.
4) Evaluation of the testbed using the AOSD techniques developed at Lancaster and gathering of empirical data based on the end-to-end case study.

The case study for this project is the HealthWatcher, a real-life system aimed at improving the quality of health care services. The system allows members of the public to register complaints against restaurants or pets, these complaints can then be investigated by health care personnel and appropriate action taken. The complaints are registered via a web-based front-end and Remote Method Invocation is then used to allow the web-server to interact with the application server.
The HealthWatcher is a non-trivial, real-life system that has been initially implemented in Java and AspectJ and then, re-implemented in CaesarJ. Each of these implementations have been evolved nine times, so the various changes that occurred in each of the of the versions of HealthWatcher could be assessed. These changes were designed so to represent common activities performed during software maintenance, refactoring, introduction of design patterns, introducing new behaviour etc.
Modularity metrics have been collected in all implemented versions (Greenwood, et al., 2007). The collected metrics are depicted in Table 15.
Results of the study showed that:

- Concerns aspectized upfront tend to show superior modularity stability in the AO designs.
- AOP solutions required less intrusive modification.
- AO modifications tended to propagate to seemingly unrelated modules.
- Invasive modification is more frequent in OO solutions but AO modifications tend to propagate to seemingly unrelated modules.

Software Attribute	Metric
SoC	CDC
	CDO
	CDLOC
Coupling	CBC
	NOC
Cohesion	LCOO
Size	LOC
	NOA
	WOC

Table 15. Metrics collected in the TAO study

Results also showed better coupling and cohesion in AOP as shown in Figure 14 and Figure 15 respectively. Lower values are better.

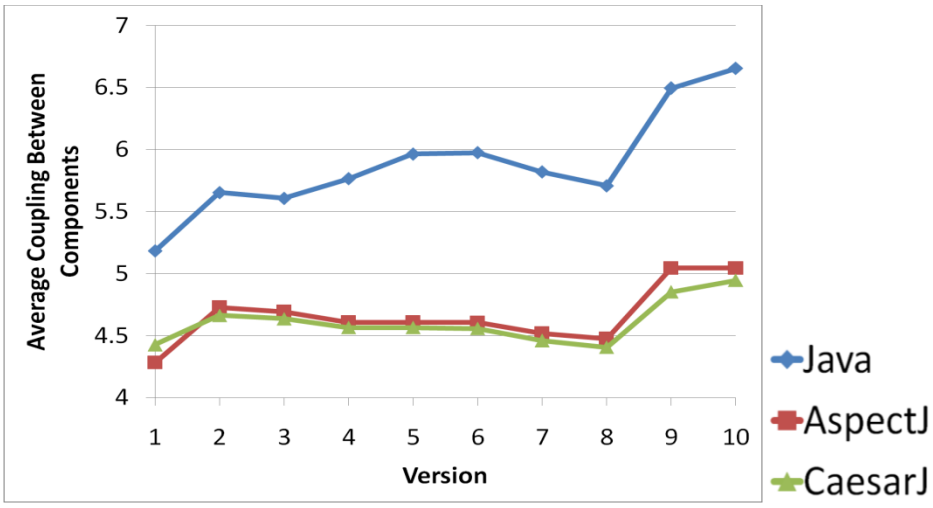

Figure 14. Coupling chart from TAO study

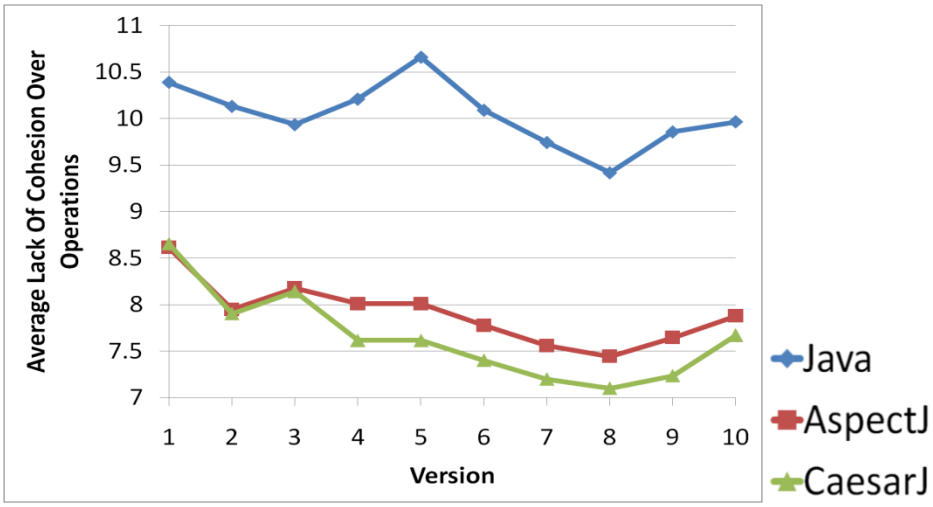

Figure 15. Lack of Cohesion chart from TAO study

By having two implementations from different aspect oriented languages, most conclusions drawn refer to AOP in general.

Unfortunately the CaesarJ implementation of the study is very similar to the AspectJ implementation as they share the same class diagram and do not take into account the particular features of CaesarJ.

8. Conclusions and future work

This chapter presents the final conclusions of this dissertation (8.1) and points some research directions for the future (8.2).

8.1 Conclusions

Building high quality systems is a driving goal in software engineering. Since AOP is a recent programming paradigm, it is still subject to research and maturation. The lack of design and implementation guidance can lead to the misuse of the new abstractions present in AOP, worsening the overall quality of the system. In this way, as AOSD moves forward, a significant research effort is required to define the quality measures that affect important quality requirements, such as modularity and complexity. Measuring the structural design properties of software artifacts, such as cohesion, and size, is a promising approach towards quality assessments. Some empirical studies have been undertaken in the context of AOSD. However, the assessment in these studies generally only applicable to AspectJ.

In this dissertation, various AOP size metrics have been formalized to support CaesarJ and a new cohesion metric was proposed. The novel cohesion metric is based on a well-known OOP metric. The LCOO-HS extends LCOM-HS to support the new features and language mechanisms of CaesarJ. Also, this metric is formalized in and fully operational manner unambiguous manner so that it no additional interpretation from the user is required, a problem that is frequent with the current metrics.

The NIC and LCOO-BDW metrics were also implemented in the MuLATo tool but were not used in the quantitative study. NIC did not fit in the context of this study. LCOO-BDW measures the same cohesion relations as LCOO-HS, so it was not included.

The size and cohesion metrics formed the basis for a comparative study between Java and CaesarJ with a focus on modularity and complexity.

Analysis of the metrics derived via the MuLATo tool lead to a few interesting insights:

Firstly, the CaesarJ implementations tend to have a significantly better cohesion than Java. This leads to better understanding of its modules as well as easier maintenance of the system. Better cohesion also increases the likelihood of reuse, while complexity is kept manageable.

Secondly, CaesarJ modules displayed an increase in size, even if its constituent parts tend to be simpler. This increase in size is possibly explained by the small size of the subjects of this study (the design patterns). CaesarJ components aim to offer better reuse and having as case study individual examples of design patters (that do not take into account reuse) can bias the

results. With bigger examples, these differences in size should decline or even be reversed, but to confirm this further studying is needed.
While the results may not be directly generalized to professional developers and real-life applications, these representative examples allow us to make useful initial assessments of the use of CaesarJ or CaesarJ-like programming languages for the modularization of classical design patterns and we consider that this subject is worth of being studied further. In spite of its limitations, the study constitutes an important initial empirical work as it proves.

8.2 Future work

Some design pattern examples are missing from the repositories used in this study so, in the immediate future it is planned to finish the development of the remaining implementations and complete the study.
In this study, experimental case studies are of small/medium size projects. It is very difficult to get large industrial projects of this domain for experiment. However, results obtained from the present study are quite instructive. Additional research is needed to repeat this study with larger, more complex, systems and assess whether some of the conclusions are specific to the examples or whether they are generalizable.
Some metrics were implemented in MuLATo but were not used. It is planned in the immediate future to do a study that compares the design patterns examples with both cohesion metrics that are already developed in MuLATo.
To fully access modularity, one has to take into account coupling, also. Coupling metrics, like *Coupling between Components* and *Depth of Inheritance Tree* for instance, should be formalized for CaesarJ and implemented in MuLATo. The frontiers between cohesion and coupling should be studied further, specifically within top-level components and the relations between its inner components. A metric *Number of Inner Components* was already developed in MuLATo and can be a good start for this study.
The new metrics implemented in MuLATo also pave the way for new opportunities of future work. The TAO project was initially dismissed as the case study for this dissertation because its CaesarJ implementation was “AspectJ-like” but it is interesting to know if this style has an impact "statistically distinct" in the metrics supported by MuLATo, compared with examples of patterns. Another front is to do a comparative study between LCOO-HS and LCOO-BDW using Java system of realistic size.
This work had only in focus Java and CaesarJ. In the future it could also be extended to other AOP languages such as AspectJ, HyperJ or Object Teams.

9. References

Alexander, R. (2003). The Real Costs of Aspect-Oriented Programming. *IEE Software, 20*(6), 92-93.

Aracic, I., Gasiunas, V., Mezini, M., & Ostermann, K. (2006). An Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I. LNCS. 3880*, pp. 135-173. Darmstadt: Springer.

Baldwin, C. Y., & Clark, K. B. (1999). *Design Rules. Vol. 1, The Power of Modularity.* London: MIT Press.

Barnes, Jr., N., Hale, J., Hale, D., & Smith, R. (2006). The Cohesion-Based Requirements Set Model for Improved Information System Maintainability. *AMCIS 2006 Proceedings.*

Baroni, A. L., Braz, S., & Brito e Abreu, F. (2002). Using OCL to Formalize Object-Oriented Design Metrics Definitions. *Proceedings of ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering.* Málaga.

Bartolomei, T. T. (2007, 04 24). *MuLATo - Multi-Language Assessment Tool*. Retrieved 2010, from SourceForge: http://sourceforge.net/projects/mulato/

Bartolomei, T. T., Garcia, A., Sant'Anna, C., & Figueiredo, C. (2006). Towards a Unified Framework for Measuring Aspect-Oriented Programs. *Third International Workshop on Software Quality Assurance (SOQUA'06)*, (pp. 46-53).

Basili, V. R., Caldiera, G., & Rombach, H. D. (1994 йил September). The Goal Question Metric Approach. *Encyclopedia of Soft. Eng., 2*, 528-532.

Bergmans, L., & Aksits, M. (2001, October). Composing crosscutting Concerns Using Composition Filters. *Communications of the ACM, 44*(10), 51-57.

Bracha, G., & Cook, W. (1990). Mixin-based inheritance. *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications* (pp. 303-311). Ottawa,: ACM.

Braz, S. (2009). *A Qualitative Assessment of Modularity in CaesarJ components based on Implementations of Design Patterns.* MsC Thesis, Faculdade de Ciências e Tecnologia, Departamento de Informática.

Briand, L. C., Daly, W. J., & Wust, J. (1998). A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering, 25*(1), 65-117.

Brito e Abreu, F., & Carapuça, R. (1994). Object-Oriented Software Engineering: Measuring and Controlling the Development Process. *4th Int. Conf. on Software Quality.* McLean.

Bryton, S., & Brito e Abreu, F. (2007). Towards Paradigm-Independent Software Assessment. *6th International Conference on the Quality of Information and Communications Technology (QUATIC)* (pp. 40-54). Lisboa: IEEE Computer Society.

Cacho, N., Sant'Anna, C., Figueiredo, E., Garcia, A., Batista, T., & Lucena, C. (2006). Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. *Proceedings of the 5th international conference on Aspect-oriented software development* (pp. 109-121). Bonn: ACM.

Chae, H. S., Kwon, Y. R., & Bae, D. H. (2000). A cohesion measure for object-oriented classes. *Journal Software—Practice & Experience, 30*(12).

Chidamber, ,. S., & Kemerer, C. F. (1994 йил June). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering, 20*(6), 476-493.

Cooper, J. W. (1998). *The Design Patterns Java Companion.* Addison-Wesley.

Dijkstra, E. W. (1976). *A Discipline of Programming.* Prentice-Hall.

Eckel, B. (2003). *Thinking in Patterns* (Revision 0.9 ed.).

Ernst, E. (2001). Family polymorphism. *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01). 2072,* pp. 303–326. Springer.

Ernst, E., Ostermann, K., & Cook, W. R. (2006). A Virtual Class Calculus. *33rd ACM Symposium on Principles of Programming Languages (POPL'06).* ACM SIGPLAN-SIGACT.

Fenton, N. (1994, March). Software Measurement: A Necessary Scientific Basis. *IEEE Transactions on Software Engineering, 20*(3), 199-206.

Fenton, N. E., & Pfleeger, S. L. (1998). *Software Metrics: A Rigorous and Practical Approach 2nd Edition.* PWS Publishing Co.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison–Wesley.

Garcia, A. F., Sant'Anna, C. N., Chavez, C. v., Torres da Silva, V., Pereira de Lucena, C. J., & Staa, A. v. (2003). *Agents and Objects: An Empirical Study on Software Engineering.* Technical Report 06-03, Computer Science Department, PUC-Rio.

Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., & von Staa, A. (2006). Modularizing Design Patterns with Aspects: A Quantitative Study. *Transactions on Aspect-Oriented Software Development I. 3880*, pp. 36-74. Springer.

Gasiunas, V., Mezini, M., & Ostermann, K. (2007). Depedent classes. *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications.* ACM.

Gélinas, J.-F., Badri, M., & Badri, L. (2006). A Cohesion Measure for Aspects. *Journal of object technology, 5*(7), 97-114.

Gogolla, M., Büttner, F., & Richters, M. (2007). USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*(69), 27-34.

Greenwood, P., Bartolomei, T., Figueiredo, E., Dosea, M., Garcia, A., Cacho, N., et al. (2007). On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP).* Berlin: Springer.

Hannemann, J., & Kiczales, G. (2002). Design Pattern Implementation in Java and AspectJ. *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'02). 37*, pp. 161-173. Seattle: ACM.

Henderson-Sellers, B. (1996). *Object-Oriented Metrics: Measures of Complexity.* Prentice-Hall.

Hitz, M., & Montazeri, B. (1995). Measuring Coupling and Cohesion in Object-Oriented systems. *Int. Symposium on Applied Corporate Computing.* Monterrey.

Huston, V. (2007, January 07). Retrieved 2010, from Design Patterns: http://www.vincehuston.org/dp

IEE Computer Society. (2000). IEEE Standard 1471 . *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*.

Jedlitschka, A., Ciolkowski, M., & Pfahl, D. (2008). Reporting Experiments in Software Engineering. In *Guide to Advanced Empirical Software Engineering* (pp. 201-228). Springer London.

Kaner, C., & Bond, W. P. (2004). Software Engineering Metrics: What do they Measure and how do we know? *Proceedings of 10th International Software Metrics Symposium (METRICS '04).* Chicago.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. G. (2001). An Overview of AspectJ. *European Conference on Object-oriented Programming. 2072*, pp. 327–353. Springer.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., et al. (1997). Aspect-Oriented Programming. *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97). 1241*, pp. 220-242. Jyväskylä: Springer-Verlag.

Kumar, A., Kumar, R., & Grover, P. (2008). Towards a Unified Framework for Complexity Measurement in Aspect- Oriented Systems. *International Conference on Computer Science and Software Engineering, 2*, pp. 98-103.

Lancaster University. (2007). *TAO Project*. Retrieved 2010, from A Testbed for Aspect Oriented Software Development: http://www.comp.lancs.ac.uk/research/projects/project.php?pid=215

Lincke, R., Lundberg, J., & Löwe, W. (2008). Comparing Software Metrics Tools. *Proceedings of the 2008 international symposium on Software testing and analysis* (pp. 131-142). Seattle: ACM.

Lopes, C. V., & Bajracharya, S. K. (2005). An Analysis Of Modularity In Aspect Oriented Design. *Proceedings of the 4th international conference on Aspect-oriented software development* (pp. 15-26). Chicago: ACM.

Madsen, O. L., & Møller-Pedersen, B. (1989). Virtual Classes: A powerful mechanism in object-oriented programming. *Proceedings of the Object-Oriented Programming Systems, Languages and Applications 1989 (OOPSLA'89)* (pp. 397–406). New Orleans: ACM.

Mezini, M., & Ostermann, K. (2002). Integrating independent components with on-demand remodularization. *Proceedings of Object-oriented programming, systems, languages, and applications 2002 (OOPSLA '02)* (pp. 52-67). Seattle: ACM.

Mezini, M., & Ostermann, K. (2003). Conquering Aspects with Caesar. *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD '03).* Boston.

Mezini, M., & Ostermann, K. (2004). Untangling Crosscutting Models with Caesar. In R. E. Filman, T. Elrad, S. Clarke, & M. Aksit, *Aspect- Oriented Software Development* (pp. 165-199). Boston: Addison- Wesley.

Myers, G. J. (1978). *Composite Structured Design.* New York, USA: John Wiley & Sons, Inc.

Norvig, P. (1996). Design Patterns in Dynamic Programming. *Object World 96.* Boston.

Nystrom, N., Clarkson, M. R., & Myers, A. C. (2003). Polyglot: an extensible compiler framework for Java. *Proceeding CC'03 Proceedings of the 12th international conference on Compiler construction.* Berlin: Springer-Verlag.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM, 15*(12), 1053-1058.

Polanco, G. (2002). *GoF's Design Patterns in Java.* Politecnico di Torino.

Polya, G. (1957). *How to solve it.* Princeton University Press.

Pressman, R. S. (2000). *Software Engineering.* Mcgraw-Hill.

Rajan, H., & Sullivan, K. J. (2005). Classpects: Unifying Aspect- and Object-Oriented Language Design. *27th international conference on Software engineering (ICSE '05)* (pp. 59-68). St. Louis: ACM.

Ramnivas, L. (2003). *AspectJ in Action.* Greenwich: Manning.

Rashid, A., & Moreira, A. (2006). Domain Models are NOT Aspect Free. *MoDELS 2006. 4199*, pp. 155-169. Genoa: Springer LNCS .

Rosenberg, L., & Hyatt, L. (1997). Software Quality Metrics for Object-Oriented Environments. *Crosstalk Journal*.

Sant'Anna, C. N., Garcia, A. F., Chavez, C. v., Pereira de Lucena, C. J., & von Staa, A. (2003). On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. *Proceedings of Brazilian Symposium on Software Engineering (SBES'03)*, (pp. 19-34). Manaus.

Shull, F., Singer, J., & Sjøberg, D. I. (2008). *Guide to Advanced Empirical Software Engineering.* London: Springer-Verlag.

Sjoeberg, D. I., Hannay, J., & Hansen, O. (2005). A survey of controlled experiments in software engineering. *Transactions on Software Engineering, IEEE, 31*(9), 733-753.

Sousa, E., & Monteiro, M. P. (2008). *An Exploratory Study of CaesarJ Based on Implementations of the Gang-of-Four patterns.* Faculdade de Ciências e Tecnologia, Departamento de Informática.

Truett, L. (n.d.). *Java Design Patterns Reference and Examples*. Retrieved 2010, from FluffyCat: http://www.fluffycat.com/Java-Design-Patterns/

Tsang, S. L., Clarke, S., & Baniassad, E. (2004). *Object Metrics for Aspect Systems: Limiting Empirical Inference Based on Modularity.* Technical Report, Trinity College Dublin, Dublin.

Zakaria, A. A., & Hosny, H. (2003). Metrics for Aspect-Oriented Software Design. *Proceedings of the Third International Workshop on Aspect-Oriented Modeling (AOSD'03).*

Zhao, J. (2002). *Towards A Metrics Suite for Aspect-Oriented Software.* Information Processing Society of Japan (IPSJ).

Zhao, J. (2004). Measuring Coupling in Aspect-Oriented Systems. *Proceedings of METRICS'2004.*

Zakaria, A. A., & Hosny, H. (2003). Metrics for Aspect-Oriented Software Design. *Proceedings of the Third International Workshop on Aspect-Oriented Modeling (AOSD'03).*

Zhao, J. (2004). Measuring Coupling in Aspect-Oriented Systems. *Proceedings of METRICS'2004.*

Zhao, J. (2002). *Towards A Metrics Suite for Aspect-Oriented Software.* Information Processing Society of Japan (IPSJ).