Quantitative and qualitative evaluation of AspectJ, JBoss AOP and CaesarJ, using Gang-of-Four design patterns

Soumeya Debboub¹ and Djamel Meslati ²

Computer Science Department, LISCO Laboratory
Badji Mokhtar-Annaba University, Annaba, Algeria
¹ Debboub_soumeya@hotmail.com, ² meslati_djamel@yahoo_com

Abstract

Several languages are currently proposed to apply the basics of the aspect oriented programming and choosing one language or another is not obvious since they are sophisticated languages that deal with new sophisticated concepts of software engineering.

A comparison between these languages is therefore worthwhile, not only to help developers choosing the right language for their needs but also to improve these languages themselves. However, such a comparison requires important investigations to put into evidence strengths and weaknesses of each language and ultimately operate a synergy between these languages.

In this study we have selected three of well known advanced separation of concerns (ASOC) languages: AspectJ, JBoss AOP and CaesarJ, and we have conducted a comparative study using the GoF design patterns as hypothetical benchmarks. Our starting point was our confidence in the fact that design patterns are seamlessly powerful elicitation artifacts to test separation of concerns languages. Indeed design patterns cover most of the problems associated with the design and implementation of large and complex software systems.

This article reports on our quantitative and qualitative comparisons using eight GoF design patterns in each of the AspectJ, JBoss AOP and CaesarJ languages. The result of this work allowed us to highlight strengths and weakness of each language and showed that the use of design patterns as benchmarks is an effective way for comparing ASOC languages.

Keywords: Advanced Separation of Concerns, Aspect-oriented programming, Empirical assessment, AspectJ, JBoss AOP, CaesarJ

1. Introduction

The increasing complexity of current software systems shows up some limitations of the object oriented paradigm. Advanced Separation of Concerns (ASOC) is an approach that seems promising to meet these limitations. However, to cover the integration of the new concepts introduced by this approach, many extensions of existing languages such as C, C++, Java or Eiffel have been proposed. Each extension tries to provide the concepts of this new paradigm in a useful way.

Nevertheless, the development of several languages extensions in a relatively short period of time makes the choice of an adequate language for a given programming task, difficult, especially if we take into account the fact that the concepts dealt with are somewhat sophisticated. Our work is a contribution that aims to compare three ASOC languages using eight GoF (Gang of Four [1]) design patterns. Our contribution is two folds. First, up to now, most of the comparisons made on advanced separation of concerns approaches focus on the qualitative aspects. Therefore a quantitative comparison became necessary to show
quantitatively the principal attributes of a software system, such as coupling and cohesion. Second, nowadays several complex applications are essentially based on design patterns, and most developers are more familiar with these patterns. However, although the aspect approach provides more advantages for the implementation of design patterns it remains relatively less used, this fact is verified and validated by several studies [6, 14, 15]. So, such a comparison may helps developers choosing suitable ASOC languages for their needs and, also, evolving their object conceptions to other more profitable languages according to the multiple uses of design patterns.

In the same way such a comparison highlights relative strengths and weaknesses of languages, it can be used to improve them, however this aspect is out of the scope of this article.

Our comparison covers the three ASOC languages: AspectJ, JBoss AOP and CaesarJ. All three are based on the Java language and apply the concepts of aspect-oriented programming (AOP) as a subset of the ASOC programming paradigm. Since the base language is the same, this allows us to focus, in our comparison, on the language extension (i.e. what is added to support aspect oriented programming concepts), rather than on the differences coming up from multiple base languages.

The AspectJ language was the first proposed, it extends the Java language with new concepts like aspects, advices and inter-type member declarations. This language has proven effective in several experimental studies [14, 15]. Similar to AspectJ, JBoss AOP works with almost the same principles, the language belongs to what is called «AspectJ-like» languages. However, JBoss AOP advocates the idea of not extending Java with new constructs but using pure Java instructions to express concerns. Unlike AspectJ and JBoss AOP, CaesarJ supports specific concepts, such as virtual class, bindings and mixin composition.

We think that the GoF design patterns can be used as benchmarks in our comparison. These design patterns form complex structures whose implementation is not trivial for programming languages. Indeed, simplifying the implementation of some design patterns may be a very challenging task.

In the quantitative comparison part of this work, we made use of structural object oriented metrics, but we also took into account the recent influence of the aspect programming approach on these metrics. One of the metric used in this quantitative comparison is the performance metric which consists in the calculation of the execution time.

The remainder of this article is organized as follows: Section 2 presents an overview of our comparison approach, Section 3, 4 and 5 present, respectively, the ASOC approaches, design patterns and metrics selected for the comparison. Section 6 presents the results of the comparison and discussion of these results. Section 7 summarizes some related work. The conclusion and some perspectives are given in section 8.

2. Overview of our Comparison

The approach we have chosen to perform our comparison is based on the implementation of eight GoF design patterns with the three languages, AspectJ, JBoss AOP and CaesarJ.

In order to be fair with the three languages we asked a group of twelve Master students to achieve the implementation of eight patterns with the three languages. This was achieved in the context of a course on advanced separation of concerns. The examples of programs chosen in these implementations are the same for all; and we tried to use the minimum number of components (class, aspect) and members (method, advice) and, also, work as much as possible with highly reusable components (interfaces, abstract aspects, and abstract classes).
The use of these implementations as hypothetical benchmarks allowed us to achieve the two kinds of comparison: the quantitative one and the qualitative one.

For the first comparison we used structural and performance metrics. The lack of full-blown tools to collect such metrics forced us, in some cases, to collect them manually.

Following these metrics we have done three sub-comparisons:

— A comparison using object paradigm extended metrics; the results of this comparison reflect three important criteria that are: the coupling, the size and the cohesion of programs.

— A comparison using metrics specific to the aspect oriented programming paradigm: the results of this comparison reflect the use of new AOP concepts by AspectJ, JBoss AOP and CaesarJ

— A comparison using performance metric: the results of this comparison give an overall idea about support tools of these languages.

The qualitative comparison aims to give remarks about some aspects observed during the learning of the three languages and the implementation of the eight design patterns using these languages, and, for which, it doesn’t exist any quantitative metric.

The qualitative comparison was conducted in parallel with the quantitative study using questions that are divided into four categories: the language understandability, the language concepts, the maturity of the tools supporting the languages and the difficulties faced by the Master students when implementing the different design patterns.

Notice that the results presented in this report concern the design patterns best implementations that we have selected among those writing by Master students or already existing in literature. (The UML representations and the source code for all of our examples are available at: http://www.debboub-soumeya.sitew.org/#Research_Projects.C)

The overall steps of comparison are depicted in Figure 1.

3. The AOP Approaches Selected for Comparison

AspectJ was originally developed by Gregor Kiczales and his team at Xerox PARC. AspectJ provides explicit language support for modularizing application concerns that crosscut the application base code. By separating the base code from crosscutting concerns (called aspects), the application source code becomes untangled and consequently becomes easy to understand, maintain and reuse. To obtain the executable code, a special tool called weaver is used to combine the application base code and its specific aspects [10].

AspectJ is a general-purpose aspect oriented programming (AOP) extension to JAVA that introduces four concepts: Aspect, Pointcut, Advice and static crosscutting. An aspect is an entity that looks like a class but models a concern that crosses object classes. Pointcuts are declarations used in an aspect to identify principled points in the program execution and source code locations where it can be involved. Principled points such as an access or change of a field value, a method call or a method execution are called Join points. Pointcuts are particular forms of predicates that use Boolean operators and specific primitives to capture join points and dynamic contextual information such as parameters of a call statement.

AspectJ supports eleven different kinds of join points: method call, method execution, constructor call, constructor execution, field get, field set, pre-initialization, initialization, static initialization, handler, and advice execution join points. There are also nine kinds of pointcut designators that match join points according to their kind: call, execution, get, set, handler, static initialization, preinitialization, initialization, and advice execution.
The aspect code is divided into blocks called advices. They are method-like mechanisms used to declare that a certain code should execute before, after or around the code corresponding to the join points captured by pointcuts. Therefore, there are three possible relationships that bind an advice to pointcuts: before, after and around. AspectJ provides a rich set of primitive pointcuts to specify join points within an aspect.

The last concept of ASPECTJ is the static crosscutting which modifies a program at compile time by specifying new members of a class (called introduction) or specifying what a class extends or implements (called inter-type member declaration). For more details please see [19, 9, 24].

JBoss AOP. was designed and developed by Bill Burke [17]. It can be used independently or in conjunction with J2EE application server JBoss; in the first case it is called Standalone. From version 4.0, the JBoss Application Server includes as standard the JBoss AOP framework.

If AspectJ defines pointcuts using keywords, the pointcuts declarations in JBoss AOP can be done in two ways: in a dedicated XML document (usually called jboss-aop.xml), or in the class that implements the aspect, as annotation. JBoss AOP allows defining five types of pointcuts: method execution, constructor, attribute, class and method call.

In JBoss AOP, the aspect is a java class. The advices are methods (i.e., code that must be executed). An interceptor in JBoss AOP is a particular type of aspect that has only one advice. The mix-in mechanism provided by JBoss AOP allows extending the behavior of an application. This mechanism is similar to the introduction mechanism applied in AspectJ. Specifically with the mix-in mechanism we can introduce interfaces, fields and methods to the existing classes of an application. For more details please see [11].

CaesarJ. is an aspect-oriented language with a strong support for reusability. It combines the aspect-oriented constructs, pointcut and advice, with advanced object-oriented modularization mechanisms [8], such as family class, wrappers and mixing composition. A family class is a special CaesarJ class which can contain inner classes called “virtual classes”.

A CaesarJ program may be seen as a component in which there is a general part that represents the collaboration between classes to achieve a concern; this part is called collaboration interface and it is independent of all specific application. Two others primary parts in the structure of the CaesarJ component are: CaesarJ implementation and CaesarJ binding. CaesarJ implementation class encloses all elements of the concrete implementation of the component that are potentially reusable across multiple cases and scenarios. CaesarJ supports polymorphism in such a way that it is possible to switch from one implementation to another without the need to perform invasive changes on the remaining parts of the system. CaesarJ binding is the glue that binds the CaesarJ component to a specific application.

Implementations and bindings are composed together by declaring a concrete CaesarJ class, called weavelet, that uses mixin composition to integrate all modules into a component. For more details please see [3, 8,12].

Figure 2 illustrates a simplified overall programming process of the three languages. Notice that according to the supporting IDE, the weaving step can take place at different moments: at compile time, during /after the class-loading or at execution time.
4. The Design Patterns used for Comparison

A design pattern refers to a general solution to a design problem that has proven effective and deserves to be preserved for reuse in different situations. The gang of four catalog (i.e. frequently called GoF catalog [1]) contains 23 design patterns.

The GoF design patterns are the most popular; they have been used in many research works in various fields of software engineering. In the context of the aspect oriented programming, design patterns can be used to study the characteristics of a given language because the implementation of each one has a non trivial structure.

According to the GoF catalog, these patterns belong to three broad classes: creational patterns, structural patterns and behavioral patterns. Where each class covers respectively: the instantiation of objects, their composition to make larger structures and the allocation of responsibilities between objects and the description of the communications between them.

In this study, we selected eight design patterns, we have taken at least two patterns from each class. Having a diverse set of classes in our study allowed us to interact with problems of different nature, which then allowed us to test the use of new programming aspects concepts with the three languages, such as introduction mechanism and dynamic deployment of advices.

These patterns were chosen because we found them the most interesting to cover and also due to the existence of other work using these patterns. Especially for CaesarJ [3], which allowed us to compare our solutions to those existing and select the best one for the
comparison. Hereafter are some criteria mostly cited in design patterns research work, for the designs patterns we have selected.

*The Observer design pattern* defines a one-to-many dependency between objects so that when one object state changes, all dependent objects are notified and updated automatically [1]. This pattern is often used as the starting point for learning and introducing the separation of concern approaches, because it illustrates well how an approach menages direct dependencies.

*The Chain Of Responsibility design pattern* avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. It chains the receiving objects and passes the request along the chain until an object handles it [1]. This pattern allows separating the different processing steps.

*The Singleton design pattern* is among the most important GoF design patterns, it illustrates clearly how instantiation of objects can be controlled which is interesting in the security context and critical resources handling.

*The Adapter design pattern* lets classes with incompatible interfaces work together. This pattern is very useful in the maintenance of systems, because it allows the adaptation of sub-systems without rewriting all the code.

*The Visitor design pattern* defines a new operation to be performed on the elements of an object structure without changing the classes of the elements on which it operates. This pattern is useful when changing the object structure classes requires redefining the interfaces, which is potentially costly.

*The Decorator design pattern* attaches additional responsibilities to an object dynamically [1]. This pattern avoids sub-classing the class to add these responsibilities, which reduces the program structure, and the coupling between the components of a program.

*The Bridge design pattern* decouples an abstraction from its implementation so that the two can vary independently [1]. Consequently the implementation of an abstraction can be configured at run-time. It is also possible for an object to change its implementation at run-time.

*Abstract Factory pattern* provide an interface for creating families of related or dependent objects without specifying their concrete classes [1]. The purpose of this design pattern is to isolate the creation of objects from their use.

Notice that; the two patterns: Singleton and Abstract Factory belong to the creational patterns class; Decorator, Bridge and Adapter patterns belong to the structural class, and finally Observer, Chain of responsibility and Visitor patterns belong to the behavioral class.

5. The Metrics Selected for the Comparison

The metrics are important to estimate a software development process and software maintenance and, consequently, to compare software programs. The existing metrics for aspect-oriented systems are a set of mixing between extended object oriented metrics and new metrics devoted specifically to aspect oriented programming [5].

The majority of structural metrics selected in this comparison, are defined, extended and validated in previous research works like those in [2, 5, 13, 21].
We divide the metrics chosen into three groups: object paradigm extended metrics, metrics specific to the aspect oriented programming paradigm and performance metric. We describe them in the following.

5.1. Object Paradigm Extended Metrics

This first group contains the metrics that quantify the coupling, cohesion and size. These metrics have been used in different studies [7, 15, 18]. We give their descriptions in Table 1.

### Table 1. Object Paradigm Extended Metrics

<table>
<thead>
<tr>
<th>Size</th>
<th>LOC</th>
<th>Counts the lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>WOM</td>
<td>Counts the number of methods and advices of each class or aspect and the number of their parameters.</td>
<td></td>
</tr>
<tr>
<td>NOC</td>
<td>Counts the number of immediate subclasses or sub-aspects of a given module</td>
<td></td>
</tr>
<tr>
<td>NOO</td>
<td>Counts the number of operations of each class or aspect.</td>
<td></td>
</tr>
<tr>
<td>NOA</td>
<td>Counts the number of attributes of each class or aspect.</td>
<td></td>
</tr>
<tr>
<td>Coupling</td>
<td>CBC</td>
<td>Counts the number of other classes and aspects to which a class or an aspect is coupled.</td>
</tr>
<tr>
<td>DIT</td>
<td>Counts how far down in the inheritance hierarchy a class or aspect is declared.</td>
<td></td>
</tr>
<tr>
<td>Cohesion</td>
<td>LCOO</td>
<td>Measures the lack of cohesion of a class or an aspect in terms of the amount of method and advice pairs that do not access the same instance variable.</td>
</tr>
</tbody>
</table>

According to the model of quality proposed by Sant'Anna et al., in [4], the three attributes: size, coupling and cohesion reflect the two most important assessment criteria: maintainability and reusability of software systems.

5.2. Metrics Specific to the Aspect Oriented Programming Paradigm

The second group contains metrics that reflect the new concepts introduced by the aspect oriented programming paradigm. Table 2 gives the meaning of these metrics.

### Table 2. Metrics Specific to the Aspect Oriented Programming Paradigm

<table>
<thead>
<tr>
<th>CIM</th>
<th>Counts the number of modules or interfaces explicitly named in the pointcuts belonging to a given aspect.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDA</td>
<td>Counts the number of modules affected by the pointcuts and by the introductions in a given aspect.</td>
</tr>
<tr>
<td>NAP</td>
<td>Counts the number of Abstract pointcuts.</td>
</tr>
<tr>
<td>NCP</td>
<td>Counts the number of concrete pointcuts.</td>
</tr>
<tr>
<td>NP</td>
<td>Counts the number of pointcuts</td>
</tr>
</tbody>
</table>

In addition to the metrics given in Table 2, we propose a new metric called NCI that counts the number of components (classes, aspects, XML files and others artifacts) introduced by a given approach in the implementing of a design pattern. This gives us an overall count of the components added to support a design pattern.

5.3. Performance Metric

For the performance, we have computed the execution time of each program embodying design patterns. In order to get accurate results, we repeated the execution of these programs for several thousand times using repeated loops.

Notice that, the performance metric evaluates mainly the tools that support a language.
6. Assessment of AspectJ, JBoss AOP and CaesarJ

6.1. Quantitative Comparison

After implementing the eight design patterns in AspectJ, JBoss AOP and CaesarJ, we obtained the results given in Figures 3, 4 and 5. According to the nature of the metrics used, we divided the results into three groups.

6.1.1. Results of Object Paradigm Extended Metrics

We present here the results obtained using the object paradigm extended metrics, as we have already mentioned, these metrics reflect three criteria that are: coupling, size and cohesion.

Concerning the coupling, AspectJ scored the best results for the coupling attribute for the five patterns (Bridge / Chain of responsibility / Observer / Visitor / Adapter), if we take the example of Chain of responsibility pattern we found that AspectJ uses only two aspect, the first, which is abstract, encapsulates the generic protocol of the design pattern, and the second, which is concrete, connects this protocol to a particular application. In contrast with CaesarJ we have been forced to use four elements to ensure such a relationship, which increases the coupling between the elements making this pattern, and consequently the coupling in the final implementation.

Decorator gave good results with CaesarJ. The dynamic deployment of aspects offered by CaesarJ has been very useful in the implementation of the decorator design pattern, where the intention of the latter attach additional responsibilities to an object dynamically, providing a flexible alternative to subclassing for extending functionality.

Notice that CaesarJ presents a new way of dealing with modularization and offers new conceptual language modules. It also makes use of mechanisms like virtual classes that provide classes with the ability to process inner classes as class attributes, in the same way as methods and fields. The inner classes are also virtual classes and have different properties than Java’s inner classes. Conceptually, the inner classes affect in some ways the coupling attribute.

In our study we took into account the two sort of coupling, thus we have two results for the coupling attribute in CaesarJ language in Figure 3 (i.e., taking into account the coupling between inner classes in the first calculation, and ignoring this kind of coupling in the second).

In JBoss AOP the coupling between aspects and base classes is located in XML files, because this is where we find the different relationships building the cross-cutting concerns. In the implementation of the Singleton design pattern we used a single interceptor that manages the instantiation of the singleton class. This reduces the coupling attribute in this design pattern.

Figure 3 illustrates with a graphic the measurement values of the coupling attribute for AspectJ, CaesarJ and JBoss AOP (lower values are better).
Concerning the Size, AspectJ, along with patterns (Bridge / Chain of responsibility / Observer / Visitor / Singleton / Adapter) showed better results for the size attribute, this is due to the power of its introduction mechanism. For example, the Visitor design pattern relies mainly on the introduction mechanism provided by AspectJ; this allows us to implement this pattern with only one aspect.

The realization of the three design patterns Bridge, Decorator, and Adapter, showed good results for the Size attribute with CaesarJ. In the case of the Bridge design pattern, it shows clearly the method followed by CaesarJ, where the various elements of the component CaesarJ are used in the implementation of this pattern. The only drawback of this implementation is that it is essentially based on the use of virtual classes which limits the reuse of these classes.

For the size attribute, JBoss AOP did not give good results; this is due to his introduction mechanism that uses three different elements and also the use of castings, which increases the size of a program.

Figure 4 illustrates with a graphic the measurement values of the size attribute for AspectJ, CaesarJ and JBoss AOP (lower values are better).

Concerning the Cohesion, AspectJ has shown good results for five design patterns (Bridge, Visitor, Decorator, Singleton and Adapter). However, in the Observer design pattern we found that the cohesion attribute is very high due to the lack of cohesion in the abstract aspect.
ObserverProtocol that manages itself the observers. This design is conceptually questionable leading to a poor separation of concerns inside the aspect. Mezini and Jan Grék discussed this problem in [20] and [22].

The CaesarJ approach which is based on the assembly of components that work together in a single element, gives clear results in favor of CaesarJ which took the first place in the realization of different design patterns when considering the cohesion attribute.

Most of the JBoss AOP implementations (Bridge / Observer / Visitor / decorator / adapter), showed positive results with regard to the cohesion attribute.

Figure 5 illustrates with a graphic the measurement values of the cohesion attribute for AspectJ, CaesarJ and JBoss AOP (lower values are better).

![Figure 5. The Cohesion Attribute Results](image)

In summary, the analysis of the three attributes (coupling, size and cohesion) based on the eight design patterns gave the following observations.

Coupling: AspectJ provided better results for the coupling attribute, and according to the quality model proposed by Sant'Anna et al., in [4], this attribute influences directly the understanding and flexibility of programs, and subsequently the maintainability and reusability. The simplicity and clarity of programs produced by AspectJ confirmed these results.

CaesarJ didn’t give good results for this attribute, this can be justified by the fact that elements of the CaesarJ component form a kind of additional coupling, besides the coupling produced between internal classes of these elements. This led to more difficult understanding compared to AspectJ programs.

JBoss AOP uses XML files as a means of communication between classes and aspects, which need to carefully read this file to understand these relationships, and the coupling attribute.

Size: According to the quality model, the size attribute affects the programs understanding, the latter, in turn, affects the maintainability and reusability.

Since the most aspect solutions of design pattern are often based on the introduction mechanism, AspectJ gone beyond the other approaches in the size attribute. That is, due to the simplicity of its introduction mechanism, it does not require a lot of elements.

Cohesion: the cohesion was good in the achievement of most of the patterns, and for the different approaches, because the new programming concepts improve this attribute, thus leading, to the improvement of the maintainability and the reuse.
CaesarJ exhibits clearly its power for this attribute; it was the leader in the implementation of various design patterns. This amounts to the basic idea of this approach, which involves gathering the classes that work together in wrapper units.

The following table summarizes the different results. The best approach is shown for each pattern and for each metric.

### Table 3. Results of the Object Paradigm Extended Metrics

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Cohesion</th>
<th>Coupling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visitor</td>
<td>AspectJ</td>
<td>All</td>
<td>AspectJ</td>
</tr>
<tr>
<td>Observer</td>
<td>AspectJ</td>
<td>CaesarJ/JBoss</td>
<td>AspectJ</td>
</tr>
<tr>
<td>Singleton</td>
<td>AspectJ</td>
<td>AspectJ/CaesarJ</td>
<td>JBoss</td>
</tr>
<tr>
<td>Decorator</td>
<td>CaesarJ</td>
<td>All</td>
<td>CaesarJ</td>
</tr>
<tr>
<td>Chain of responsibility</td>
<td>AspectJ</td>
<td>CaesarJ</td>
<td>AspectJ</td>
</tr>
<tr>
<td>Adapter</td>
<td>AspectJ/CaesarJ</td>
<td>All</td>
<td>AspectJ</td>
</tr>
<tr>
<td>Bridge</td>
<td>AspectJ/CaesarJ</td>
<td>All</td>
<td>AspectJ</td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>CaesarJ</td>
<td>AspectJ/CaesarJ</td>
<td>CaesarJ</td>
</tr>
</tbody>
</table>

#### 6.1.2. Results of Metrics Specific to the Aspect Oriented Programming Paradigm

Figure 6 illustrates with a graphic the results obtained using metrics specific to the aspect oriented programming paradigm, these metrics reflect the use of new AOP concepts by AspectJ, JBoss AOP and CaesarJ. (lower values are better).

![Figure 6. The AOP Metrics Results](image)

Figure 6 show that the use of pointcuts by AspectJ exceeds JBoss AOP and CaesarJ. This gives a high value of CIM which in turn indicates a strong coupling between applications classes and aspects pointcuts, limiting subsequent by the reuse of aspects as they are and requiring redefinition of their pointcuts.
According to Sousa et al., [3], the use of pointcuts in CaesarJ should be left to situations in which behavior compound does not follow an identifiable pattern in the static structure of the system, for example, calls to a dispersed method or constructor.

By cons, in the account of NCI attribute we found that with a minimum number of additional components on the bases classes, AspectJ can achieve a different design pattern, which simplifies the understanding and reduces the application size.

6.1.3. Results of Performance Metric

Figure 7 illustrates with a graphic the measurement values of the performance metric (in milliseconds), for AspectJ, CaesarJ and JBoss AOP (lower values are better). This metric provide estimation about the supporting IDE and tools of AspectJ, JBoss AOP and CaesarJ.

![Figure 7. The Performance Metric Results](image)

In the eight design patterns, JBoss AOP had bad results for the time execution; this problem is more evident with patterns that use JBoss AOP interceptors such as Singleton and Decorator. That is because JBoss AOP achieves the weaving at load-time. It is significantly slowed down since it has to do a lot of work before a class can be loaded. Once all classes are loaded though, load-time weaving has zero effect on the performance of the application [26].

6.2. Discussion about Quantitative Comparison

As showed in the previous section, the implementation of the eight design patterns with AspectJ, JBoss AOP and CaesarJ highlighted some advantages and disadvantages of the three approaches. For example, the simplicity of the introducing mechanism of AspectJ allows it to take the first place for the size attribute and enhances the understanding of programs implemented in AspectJ.

JBoss AOP has a privilege which is the use of pure Java instructions, but in another part the use of XML files can influence the execution time especially when using interceptors.

CaesarJ ensures perfect cohesion; this was very clear in the implementation of the eight design patterns. The reason is that the basic idea of this approach is to assemble the classes that work together in wrapper units. By cons, for the coupling attribute, CaesarJ implementations were more coupled in comparison to the other implementations, especially if we count the coupling produced by inner classes. This kind of coupling can be seen as a disadvantage for CaesarJ since it is not clear and requires more attention.

It can also be noted that AspectJ uses fewer components in achieving the various patterns compared to CaesarJ and JBoss AOP. But unlike CaesarJ that focuses on the structure,
AspectJ is based on dynamic events which forces the programmer to use more significantly the pointcuts.

Our quantitative comparison validates remarks observed in other previous qualitative comparisons as the lack of cohesion in the AspectJ solution of the Observer design pattern compared with that of CaesarJ, as reported in [20] and [22].

6.3. Qualitative Comparison

For our qualitative comparison, the Master students were asked four categories of questions after they have finished the implementation of the eight design patterns in each language. The four categories are: the language understandability, the language concepts, the maturity of the supporting tools and finally the difficulty faced by the students when implementing the eight design patterns.

The following table shows the asked questions along with the responses we obtained.

<table>
<thead>
<tr>
<th>Table 4. Results of Qualitative Comparison</th>
<th>AspectJ</th>
<th>JBoss AOP</th>
<th>CaesarJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language understandability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is it easy to learn the language?</td>
<td>9</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>(1=difficult, 10=easy)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is the terminology easy to master?</td>
<td>9</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>(1=difficult, 10=easy)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is there a suitable documentation for the language?</td>
<td>10</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>(1=poor, 10=rich)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Language concepts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Are the concepts easy to understand and use?</td>
<td>8</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>(1=difficult, 10=easy)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do you think that the language concepts were suitable for design pattern implementations?</td>
<td>9</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>(1=not suitable, 10=suitable)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Are the concepts sufficient to create reusable abstractions?</td>
<td>6</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>(1=not sufficient, 10=sufficient)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Supporting IDE and Tools</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do you think that the IDE used was suitable and userfriendly?</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>(1=not suitable, 10=suitable)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is it easy to find tools dealing with the approach and extending the IDE?</td>
<td>8</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>(1=difficult, 10=easy)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do you think the tools supporting the approach are mature?</td>
<td>9</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>(1=immature, 10=mature)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design patterns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do you think that the language is suitable for implementing patterns?</td>
<td>9</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>(1=not suitable, 10=suitable)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do you think that the obtained implementations are easy to understand and evolve?</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>(1=difficult, 10=easy)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.4. Discussion about Qualitative Comparison

6.4.1. Language Understandability. AspectJ was the easiest to understand. Despite, our expectation that JBoss AOP will take the lead for this criterion, seeing that it doesn’t introduce new keywords (concepts), the responses showed that the terminology of AspectJ was easy to understand and use. CaesarJ is a little bit difficult to understand at the first time, and JBoss AOP makes it difficult to handle the relationship between XML files and the Java classes.
The lack of suitable documentations for JBoss AOP and CaesarJ, compared to AspectJ, is an important factor that may explain the result of the two first criteria.

6.4.2 Language Concepts. The concepts are easy to understand and use for AspectJ more than JBoss AOP and CaesarJ. Concerning the suitability to implement design patterns, AspectJ and CaesarJ seem to have the same appreciation and are better than JBoss AOP. Concerning the ability to create reusable elements, the AspectJ abstract aspect, seem to be insufficient. In contrast, CaesarJ offers several levels of reuse (i.e., Collaboration interface and Implementation). The reusable elements in JBoss AOP languages are those of Java which explain the poor appreciation obtained.

6.4.3 Supporting IDE and Tools. The IDE used was Eclipse which is found to be suitable for all the three languages. The extension of the IDE with plug-ins was not difficult for AspectJ and CaesarJ. While JBoss AOP does not have a recent plug-in, which requires to make a small configuration on class path before using it.

The editor and programs visualization with AspectJ tool is the user-friendly and efficient (i.e., mature). It can show clearly the different relationships between aspects and the rest of the code. The CaesarJ plug-in is not as complete as that of AspectJ but it seems to be easy to work with.

6.4.4 Design patterns. The concepts offered by the three languages simplified the implementation of various design patterns. The concepts used are able to reduce drastically the implementation of some design patterns. This is the case for AspectJ where the use of the introduction concept reduced the implementation of the adapter and visitor design patterns to merely a simple concrete aspect.

We have also noted a few observations about these implementations:

— Several castings in the final patterns code are necessary in JBoss AOP, which decrease the general understanding.

— Dynamic weaving in CaesarJ and JBoss AOP, gives more benefits for certain patterns, such as the Decorator design pattern which allow adding or removing functionalities dynamically.

— The solutions proposed in AspectJ significantly reduce the number of participants involved in the implementation of a pattern. Indeed, the introduction mechanism of AspectJ allowed us to use only one aspect in some design patterns, when, in contrast, we need three elements with the mix-in mechanism of JBoss AOP (i.e. the interface containing the members to introduce, a class implementing them, and the XML file that establish the relationship between these elements and the Java classes).

— The results analysis of observer design pattern show that, the use of the abstract aspect in AspectJ is not sufficient to ensure better reuse in all situations.

— The CaesarJ implementation of the two patterns: Abstract Factory and Bridge is mainly based on the CaesarJ concepts, which placed the patterns code entirely in a CaesarJ module, including the base classes.
Threats to validity

— The limited size and complexity of the examples used in the implementations may restrict the extrapolation of our results.

— Questions concerning IDE support need larger examples and professional software applications to achieve tests than those used by our master students in the context of academic study.

7. Related Work

The works in [6, 14] and [15] are interested in the implementation of design patterns using the mechanisms and concepts introduced by the aspect technology. The authors showed the positive impact of the AOP (using AspectJ) over OO paradigm, on the implementation of all the GoF patterns. In contrast, our work focus on the difference between AOP approaches in supporting design patterns and not on their contributions.

Mik Kersten presents in [16] an interesting comparison between four major tools of the ASOC approaches that are AspectWerkz, AspectJ, Spring, JBoss AOP. He tried in his work to confront these tools for monitoring the implementation of new concepts of each one. The advantage of our study compared to that of Kersten is the use of design patterns as hypothetical benchmarks which helped us to get a clearer view of how these approaches solve different problems.

In [3], Sousa et al., describe the implementations of seven GoF design patterns with CaesarJ they conclude their report with preliminary assessment of CaesarJ in which AspectJ is used as a basis for comparison. Unlike the Sousa et al work, our comparison focuses more on the quantitative assessment, and it is based on new metrics devoted to AOP.

Bartolomei et al., present in [27] a coupling measurement framework (MuLATo) that takes into account both AspectJ and CaesarJ. This framework supports the different composition mechanisms inherent to both languages. Unfortunately, the current version of MuLATo supports only a few size metrics for CaesarJ. In our work, although we have used MuLATo, computing the metrics was not an objective for its own but a mean to compare AOP approaches.

New classifications for the GoF design patterns are proposed in [23] and [25]. These classifications are based mainly on the aspect mechanisms and concepts used in the structure of these patterns, that are repeated constantly and with strong similarities between the patterns. In this study we worked with the old classifications (i.e., proposed by the gang-of-four), but we think that using these new classifications may be interesting, since they focus on the concepts introduced by AOP approach.

8. Conclusion

This article presented a comparison between the three popular separation of concerns languages using GoF design patterns as hypothetical benchmarks. The results show significant differences concerning several quantifiable and qualitative criteria between the three selected languages. This is in favor of the use of design patterns as elicitation artifacts to compare ASOC approaches. Our goal was not to appoint the best language but rather to provide a referential work that can help to choose the right language during software development or devising new ASOC approaches that combine best qualities of AspectJ, JBoss AOP and CaesarJ.
We conducted a quantitative study using metrics dedicated to the AOP over eight GoF design patterns that have been selected for being representative of the GoF catalog. The qualitative comparison was achieved through a set of questions divided into four categories.

As a perspective to this work, we will consider achieving the comparison to cover the whole GoF catalog and eventually include other ASOC languages.

References


Authors

Soumeya Debboub, is a PhD. student in Complex Software Engineering. She obtained her Master of Science degree in Computer Science from the University of Badji Mokhtar-Annaba (UBMA), Algeria in 2009. She completed his thesis under the supervision of Prof. Meslati. Her research interests include software development and separation of concerns.

Djamel Meslati, is a professor in the department of computer science at the University of Badji Mokhtar-Annaba (UBMA), Algeria. He has been the head of the research group on evolution and reuse of software systems at the Laboratory of Research on Computer Science (LRI) from 2001 to 2012. Currently, he is the head of the Laboratory of Complex Systems Engineering (LISCO). His research interests include software development and evolution methodologies, separation of concerns.