3

The Role of Modularity in Program Stability

"I think I am getting crazy because according to my findings, modularity metrics do not indicate program stability," said Francisco to Alessandro.
"Great!" said Alessandro smiling.

Conversation between Francisco and Alessandro
In their first meeting of 2010 at PUC-Rio

Software stability stands out as one of the most critical attributes of high-quality software design (KELLY, 2006, MOHAGHEGHI *et al.,* 2004). As a consequence, a key goal of advanced programming techniques, such as AOP (KICZALES *et al.,* 1997) and FOP (MEZINI and OSTERMANN, 2002), is to promote program stability. They aim to support developers in achieving easy-to-change programs. The resulting programs ought to better accommodate requirements' changes than their OO program counterparts.

However, the use of composition mechanisms implies that the program complexity is at some extent transferred from the module structure to the composition specification. It is often the case that properties of multiple modules are involved in the composition implementation. This means that, to a large extent, much more of the time spent on programming is now devoted to implement, comprehend and potentially revisit the composition specification. When changing a program, for instance, it is also not trivial to understand the change effects on the composition specification. Even worse, when the target of a change is the module, all the related compositions might need to be revisited and modified in certain circumstances (GARCIA *et al.,* 2005).

It is well recognized nowadays that composition mechanisms help to improve modularity (GARCIA *et al.,* 2005). However, given the new complexity flavors of composition code, it is questionable if modularity metrics are the most effective indicators of quality attributes such stability. Existing module-driven metrics have suffered criticism and they are failing to be used with confidence in empirical studies (BURROWS *et al.,* 2010). For instance, such metrics are

criticized for not taking into account some properties of the multiples modules of composition implementation such as the scope of composition code in the source code (Section 1.1.2). This means that composition specifications increasingly require the reasoning about modules not explicitly declared in the composition code.

This chapter presents an exploratory study about the impact of advanced programming techniques and their composition mechanisms on program stability. This study aims at answering the first and second research questions of this thesis (RQ1 and RQ2 in Section 1.3), which state, respectively: *How to objectively analyze the impact of using composition mechanisms on program stability?* and *Are modularity conventional metrics good indicators of composition-enriched program stability?* First, we need to identify if there are specific programming techniques, such as, AOP and FOP (see Chapter 2), that play a central role in promoting program stability when compared to OO programming techniques. Second, we also need to obtain insights about how better quantify the impact of composition code on program stability. The study was carried out on the top of AspectJ and CaesarJ implementations of three evolving systems (Appendix A). The comparison between AspectJ and CaesarJ was motivated by many reasons, such as: (i) they entail the most popular programming models that support existing composition mechanisms, such as pointcuts, advice, and intertype declarations, and (ii) most of the previous evaluations were limited to evaluate AspectJ programs and this language mechanisms only.

Finally, our analysis was based on previously-defined modularity (SANTANNA *et al.,* 2004) and stability (KELLY, 2006) metrics, so that we could verify if existing modularity metrics indicate well-accepted symptoms of program stability (Section 3.1). Differently from what we expected, we have observed that modularity attributes were not the main factor that contributed to superior stability of the analyzed software systems. This happens mainly because the impact of composition properties seems to exert a great influence on program stability. The explanations for this conclusion are given in the analysis discussion (Section 3.2). The analysis was carried out taking into consideration the methodology presented in Section 3.1. Our study was compared to previous studies in Section 3.3. Threats to validity and our final considerations are presented in Sections 3.4 and 3.5, respectively.

3.1
Evaluation Methodology

This section presents the methodology used in our exploratory study (Section 3.1.3). The exploratory investigation comes in handy as we aim at first acquiring new insights into the impact of using composition mechanisms on program stability. In addition, the use of composition mechanisms is in a preliminary stage and, thus, the data regarding its usage is difficult to collect. In Section 3.1.1 the target systems used to support our investigation are presented. Our research aims and study procedures are presented in Sections 3.1.2 and 3.1.3, respectively. Finally, the metrics used to quantify the degree of software modularity and stability are introduced in Section 3.1.4.

3.1.1
Target Systems

In order to conduct a systematic evaluation on the stability of composition-enriched programs, the selected cases were three evolving systems: (i) a large open-source software framework, called iBatis (110 KLOC), (ii) an embedded mobile software for media management, called MobileMedia (5 KLOC), and (iii) a family of board games, called GameUP (3 KLOC). Both MobileMedia and GameUp are considered software product lines (SPLs) (ALVES *et al.,* 2006). The SPL approach aims to decompose software functionalities into features (CLEMENTS and NORTHROP, 2001). These systems were chosen as they have been evolved and underwent various forms of changes (Section 3.2.2) over a long period of time.

There are many others reasons that justified the choice of these systems. They are also interesting to our study because they contain different categories of: (i) functionalities implemented using composition mechanisms, (ii) domain-specific functionalities and (iii) there were good practice guidelines on how to modularize these chosen features with the composition mechanisms (MEZINI and OSTERMANN, 2004, CAESARJ, 2012). These diverse characteristics would enable us to expose AspectJ and CaesarJ, representative AOP and FOP programming languages, implementations to varied requirements for modularizing systems functionalities. In addition, the systems changes would enable us to assess to what extent certain mechanisms, such as pointcut-advice and collaboration interfaces, could help to reduce modifications of existing modules through systems changes. Moreover, the three software systems are

from significantly-different domains, and well-designed implementations with Java and AspectJ are already available, facilitating the analysis of the composition mechanisms in this study. These existing implementations have prioritized stability of systems functionalities. In particular, the GameUP provides us new functionalities combinations make possible to generate several additional systems versions, whereas the MobileMedia is considered a benchmark for evaluating advanced programming techniques. Additional information regarding these three applications is available in Appendix A. Distinguishing characteristics of the target systems are presented in the following.

iBatis. It is a Java-based open source framework for data mapping. It is composed by more than 60 versions incrementally implemented. Four versions were chosen and implemented using the AspectJ and CaesarJ programming languages. The following functionalities were chosen to be refactored with modularity mechanisms of AspectJ and CaesarJ: type mapping, error context, and design patterns.

MobileMedia. It is a program family (FIGUEIREDO *et al.,* 2008a) that provides support to manage (create, delete, visualize, play, send) different types of media (photo, music and video) on mobile devices. During its development and evolution, the initial core architecture was systematically enriched with mandatory, optional and alternative features. Seven versions of the MobileMedia were analyzed and implemented on both AspectJ and CaesarJ programming language. All the variabilities were implemented using composition mechanisms.

GameUP. GameUp is a SPL developed following the reactive approach (ALVES *et al.,* 2006). It encompasses three open-source board games where each of them is an SPL: Shogi (SHOGI GAME, 2013), JHess (JHESS GAME, 2013) and Checkers (CHECKER GAME, 2013). Checkers is an American checker whereas Shogi and JHess are chess games. All of them provide features to manage various functionalities for customizing the board (*e.g.* indicating moveable pieces) and the matches between players (*e.g.* indicating player turns). Four versions of the GameUP were analyzed and implemented on both AspectJ and CaesarJ programming language

3.1.2

Research Aims

In this exploratory study, we evaluate whether the use of AOP and FOP techniques contributes to better evolve software systems in terms of software stability (Section 2.2.1). In order to achieve this goal we took into consideration the AspectJ and CaesarJ versions of the target systems (Section 3.1.1). Our analysis was performed following the procedures described in Section 3.1.3.

Before performing this analysis, we checked whether these techniques, in fact, provide a positive and significant impact on our conventional notion of modularity. In particular, our research aims were threefold. First, *we aim at analyzing the role of modularity in terms of stability in composition-enriched programs; in other words, programs developed and evolved with advanced programming techniques and their composition mechanisms.* Second, *we analyzed if modules implemented with different composition mechanisms are equally modified.* In this case, we analyzed whether the use of different programming mechanisms promotes software evolution with equivalent stability. This step was important to check whether the enriched composition code yielded by these techniques play, to a larger or lesser extent, a role on program stability. This analysis helped us to check if modularity metrics are effective to detect the varying influence of composition code characteristics. Finally, *we also aimed at discussing some implementation factors that are detrimental to stability of evolving systems.* These factors were mostly related to the complexity of the composition code.

3.1.3

Procedures

To begin with, it is important to highlight that all target systems were already implemented using Java and AspectJ languages. For this evaluation, CaesarJ implementations were generated for all the target systems as well. During the implementation process, the versions were analyzed according to a number of programming alignment rules in order to assure equal compliance to coding styles and included functionalities. Moreover, the implementations followed the same design decisions in all implementations to ensure a high degree of module stability.

The designs and implementations with CaesarJ were also reviewed by other two independent developers and researchers. One of them is a co-author of CaesarJ language, who is also experienced on the use of all sorts of aspect-oriented and feature-oriented programming mechanisms. All these initial procedures were carried out to ensure that the comparison between the implementations was equitable and fair. Inevitably, during the CaesarJ implementation, some refactorings in the AspectJ versions had to be performed when misalignments were observed at the implementation or even at the design level. When these misalignments were discovered, the particular versions were modified accordingly. There were also cases where the code structure needed to be improved equally in all the versions.

As a second step, the degrees of modularity and stability of the target systems (Section 3.1.1) were measured and compared. Software modularity was quantified by existent modularity metrics (FIGUEIREDO *et al.,* 2008a), namely coupling metrics, through the versions of all target systems. The intention was to verify if modularity metrics are good indicators of program stability when composition mechanisms are used as with in OO programming mechanisms. Later, we analyzed the degree of stability of the modules as the target systems evolved. The idea was to verify if the composition mechanisms that have promoted higher modularity degree would be the same that produced more stable systems. Finally, our third step consisted on the comparison of the degree of stability of the system modules using different composition mechanisms.

3.1.4

Modularity and Stability Metrics

In order to achieve our research aims (Section 3.1.2), some previously-validated metrics for modularity and stability were used. Therefore, two groups of metrics were used: (i) modularity metrics and (ii) stability metrics. These metrics were applied to multiple evolving systems versions with the intention of respectively computing: (i) the constancy of pivotal modularity properties through software systems versions and (ii) the unintended software systems modifications; *i.e.*, ripple effects (YAU and COLLOFELLO, 1985).

Modularity Metrics. The modularity metrics (GARCIA *et al.,* 2005) were used to enable us to analyze to what extent a certain modularity principle remained constant through out the software system evolution. It was also useful

to compare the degree of modularity achieved by each advanced programming technique (Section 3.2.1). The modularity metrics were used to quantify the coupling. Coupling metrics received increased recognition by OO software systems developers when they were found to be indicators of important quality attributes, such as stability (BRIAND *et al.*,1999, BRIAND *et al.*,1997). However, there is no consensus on which coupling metrics are effective quality indicators ffor stability of composition-enriched programs (Section 3.2.1). The attribute coupling was measured with the following metrics: Coupling between Modules (CBC) and Depth of Inheritance Tree (DIT). CBC counts the number of components (*e.g.*,modules) from which a given component invokes a program element, whereas DIT defines the maximum length from the node to the root of the composition tree. CBC and DIT metrics are two out of six metrics proposed by (CHIDAMBER and KEMERER, 1994) widely used for measure coupling in OO programs (BRIAND *et al.*,1999)(BRIAND *et al.,* 1999) (SHEO *et al.*,2008). In addition, such metrics were chosen because (i) they are used consistently as those representing modularity (TAUBE-SCHOCK *et al.*,2011, CACHO *et al.*,2007, FILHO *et al.*,2006, GARCIA *et al.,* 2005, GREENWOOD *et al.,* 2007), even in the context of advanced programming techniques (FIGUEIREDO *et al.,* 2008a, GARCIA *et al.,* 2005, GREENWOOD *et al.,* 2007) and (ii) coupling is also a classical indicator of program faults (BURROWS *et al.,* 2010) and program changes (FIGUEIREDO *et al.,* 2008a); fixing bugs and realizing changes are the two basic sources of program instabilities (KELLY, 2006).

Stability Metrics. Stability metrics were used with the purpose of quantifying the degree of modifications on program implementation (Section 3.2.2). For instance, they enable us to check if a change, originally targeted at adding a new functionality, also affected the other functionalities and/or modules of a system. The used metrics were defined to quantify two complementary forms of modification that are directly related to the occurrence of ripple effects, which are harmful to software stability: (i) refactorings - when the change is aimed at improving the system structure while preserving the existing code semantics, and (ii) alterations - when functionalities are added, removed, or modified through the system modules. For the first case, we used a metric, called Refactoring of Modules (RoM). This metric is used to quantify structural changes in classes, aspects and/or in their respective inner elements. For the second case, a metric, named Alterations in Program Elements (APE), was used to compute the number of increments, deletions, and actual modifications in program elements (Section 3.2.2). Examples of these elements can be a class, a method, an aspect, an advice and a pointcut (see Table 3.1).

Table 3.1: Program Elements per Programming Language

Language	Program Element
AspectJ	classes, aspects, operations, pointcut and advices
CaesarJ	classes (java class and cclass), operations
Java	classes, interfaces and operations

3.2

Programming Techniques: Analysing Evolving Systems

Program evolution is driven by regular updates of modules in order to accomplish new requirements. The evolution process is improved if software designs and their implementations are modular (FIGUEIREDO *et al.,* 2008a, FAYAD,2002). In fact, the long life of a software project is only possible if changes can modularly be incorporated into the program. Otherwise, the instabilities are likely to be observed, thereby hindering the system's longevity. Therefore, as previously presented, the evolution of the target systems (Section 3.1.1) was analyzed under the perspective of two quality attributes: modularity (Section 3.2.1) and stability (Section 3.2.2). In particular, we discuss if advanced programming techniques achieve a better stability; and, to fully achieve our research aims, we also checked the relation of modularity metrics and the stability in the analyzed composition-enriched programs (Section 3.2.2).

3.2.1

Modularity Analysis

This section presents the results for the first step (Section 3.1.3) where we analyst the modularity of the target systems (Appendix A) throughout their evolution. We used the modularity metrics presented in Section 3.1.4.

New Composition Mechanisms: Is Modularity Improved? As far as modularity is concerned, the use of advanced programming techniques was evaluated based on the results of a metrics suite (Section 3.1.4). The metrics were used to quantify a fundamental attribute of modularity, namely coupling (Section 3.1.4). In order to obtain the data sample, we have considered the coupling of modules present in all the versions associated with all the target systems (Appendix A). Lower values for coupling (CBC) represent better modularity as modules with lesser dependencies are likely to confine and not propagate changes with higher frequency. As someone would expect, we have observed that the composition mechanisms supported by CaesarJ and AspectJ

led to more modular code when compared to Java (See Figures 3.1, 3.2 and 3.3). This result confirms the expectation that advanced programming techniques play a role in improving the conventional notion of modularity in software systems. As illustrated in Figures 3.1, 3.2 and 3.3, advanced programming techniques promote gains regarding code modularity for all the applications that were analyzed. A careful analysis of measures confirms the superiority of CaesarJ when compared to AspectJ and Java in terms of CBC and DIT. According to the CBC values in Figures 3.1, 3.2 and 3.3, we could state that the program modularity is to advanced programming techniques in the following order: FOP, AOP and OO, for both CBC and DIT.

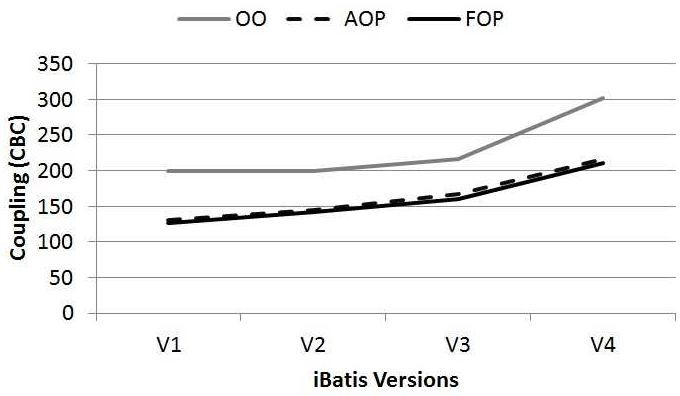

Figure 3.1: Modularity for iBatis

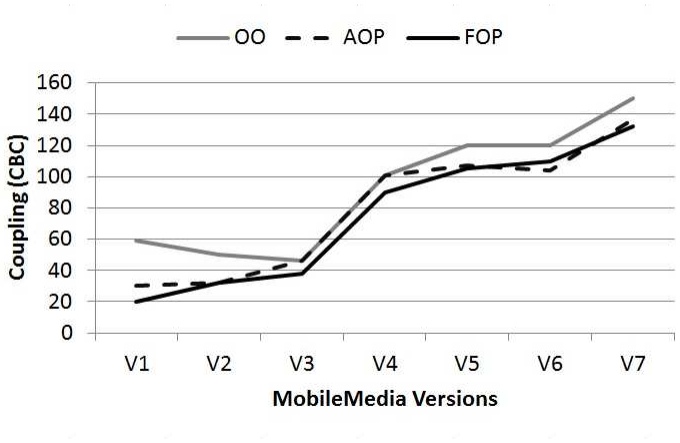

Figure 3.2: Modularity for MobileMedia

Revisiting the Figures 3.1, 3.2 and 3.3, it is possible to observe the gains varies in different proportions when OO implementations (worst technique) are compared to FOP ones (best technique), which are: from 12% to 195% (MobileMedia), from 40% to 58% (iBatis) and from 26% to 85% (GameUP). The gains variation is associated with the scope of the code used to implement a given maintenance scenario. When the scope is low, the coupling measures

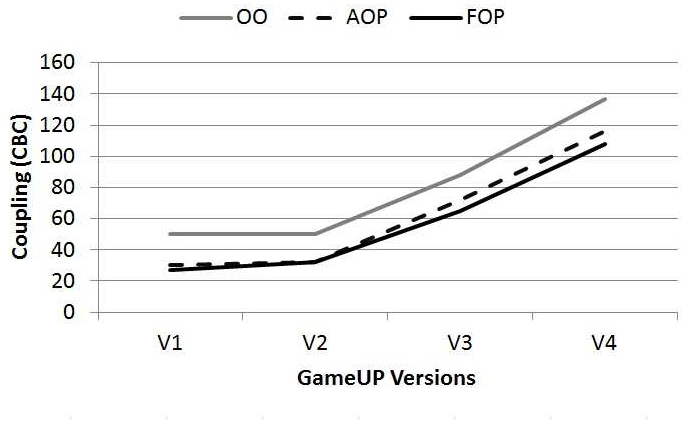

Figure 3.3: Modularity for Games

(metric CBC) tend to have a little variation, regardless of the programming technique that was used. However, the results are always favorable to advanced programming techniques. On the other hand, a high scope clearly shows the superiority of the advanced techniques. The exceptional handling implemented in MobileMedia (version 1), for example, has a high scope, which tends to make the Java code more coupled. The exception handling for the feature `IncludePhoto`, for instance, requires the implementation of three different exception types. In AspectJ, the three exceptions are implemented in just one aspect and they are used by three different classes: `C1`, `C2` and `C3`, creating a coupling equal to 3 (CBC=3). For the same scenario, the Java implementation requires the creation of a new class to implement each one of the exceptions - `C4`, `C5` and `C6` - so that they can be treated differently. This way, `C1` is coupled with `C4`, `C5` and `C6`, generating a CBC = 3, as well as `C2` (CBC = 3) and `C3` (CBC = 3), generating a coupling amount equal to 9.

Comparing the modularity measures of AspectJ and CaesarJ, the differences were almost insignificant (around 5% on average). An in-depth analysis of both measures and implementations revealed that the superiority of CaesarJ is thanks to the fact they address some expressiveness shortcomings of the composition mechanisms supported by AspectJ. For instance, composition mechanisms of CaesarJ (Section 2.1.2) enable further modular decomposition of the code beyond the use of AOP-specific composition mechanisms supported in both languages. There are more modules that can be advised and composed with aspects. As a consequence, the opportunity for advising join points is significantly increased in CaesarJ code. The FOP mechanisms of CaesarJ enable more granular definition of modules. Modules in CaesarJ are not only restricted to aspects that modularize crosscutting concerns. Modules are also cclasses (Section 2.1.2) that modularize fine-grained features or functionalities

that are decomposed hierarchically and have no crosscutting effect on the program. Finally, a small variation in the coupling of MobileMedia (version 6) is unfavorable to CaesarJ. This happens because, for this particular version, the depth of inheritance relationships (DIT values) is considered high and generates a high coupling.

We also observed that some of the AspectJ shortcomings were caused often by the stronger coupling between crosscutting feature implementations (within aspects) and the base program. An example is the definition of a non-functional aspect (*i.e.*, an aspect realizing a non-functional requirement) by enumerating the join points by name or according to certain naming conventions in AspectJ. In CaesarJ a new functionality is often a result of a polymorphic composition by means of virtual class and mixin composition (Section 2.1). This means that there were fewer opportunities for pointcut fragility scenarios in CaesarJ.

Do Composition Mechanisms Affect Stability Similarly? In fact, someone can notice that there was no visible difference between the AspectJ and CaesarJ curves in the graphics for all the three systems. This result is a clear indicator that coupling metrics are not able to reveal the nuances of the composition code in AspectJ and CaesarJ, and their different influences on program stability. This result is quite surprising as different AOP and FOP mechanisms were employed in the implementations. We are going to observe later that, in fact, the differences of stability measures of AspectJ and CaesarJ versions varied through the systems' evolution (Section 3.2.2). However, there is no perceptible variation in the coupling measures through those systems as we can observe in the Figures 3.1, 3.2 and 3.3. The role of composition mechanisms on program stability is discussed throughout the Sections 3.2.2 and 3.2.3.

3.2.2

Stability Analysis

This section presents the results for the second step (Section 3.1.3) regarding program stability. We analyse the stability of the target systems (Appendix A) throughout their evolution. This analysis was carried out based on the stability metrics presented in Section 3.1.4.

The more changes are required to realize a new software evolution scenario, the more unstable the system design is likely to become. In this manner, we calculated the number of changes that occurred in AspectJ and CaesarJ implementations. Table 3.2 summarizes the change classification observed in

MobileMedia, iBatis and GameUP in terms of the stability metrics (Section 3.1). Table 3.2 relies on the following notation pattern: AspectJ results are labeled as "AJ" and CaesarJ results are labeled as "CJ".

Table 3.2: Changes in SPL

Changed Type	MobileMedia		iBatis		GameUP	
	AJ	CJ	AJ	CJ	AJ	CJ
Alterations in Program Elements (APE)	153	105	366	226	78	66
Refactoring of Modules (RoM)	129	59	294	172	64	63
Total	**282**	**164**	**660**	**398**	**142**	**129**

Virtual classes succeed in promoting program stability. Taking Table 3.2 into consideration, we can observe that the number of changes was significantly lower in the CaesarJ implementations when compared with AspectJ ones. For instance, when refactoring names of classes, all the implementations suffer with these changes. However, some composition mechanisms of AspectJ, such as intertype declarations require more changes than the counterpart virtual classes of CaesarJ. These changes are associated with some properties of the composition implementation, such as the scope, which is associated with the set of modules taking part in such a composition. We should recall that the coupling measures (Section 3.1.4) were similar and do not follow the differences of stability measures for AspectJ and CaesarJ observed in Table 3.2.

The use of virtual classes is illustrated in Figure 3.4. The method `getNumberOfViews()` was added to the class `MediaData` in `PhotoSorting` (line 06 - box #2). This addition overrides the class `MediaData` in `MediaManagement cclass` as `MediaData` is virtual. It is important to notice that no modification was required. The same operation is illustrated in Figure 3.5 using intertype declarations. In this case, the aspect `SortingAspect` was modified (Figure 3.5) and, as this method cannot be overridden, it will be constantly changed along the SPL evolution process.

The use of virtual classes (Section 2.1.2) promotes the evolution using feature specializations with an inheritance mechanism. Thus, many methods and cclasses are overriden instead of being intrusively and partially modified. Figures 3.6 and 3.7 shows an example of this scenario before and after the modification. Notice that the method `showMediaList( )` was overriden by the addition of cclass `Favourite`, which is in charge of allowing users to specify and view their favourite photos. The same technique was used to add the other features in the evolution of the other SPL applications. In a nutshell, we can say that with the use of virtual classes the degree of stability was ameliorated 65% - iBatis application - when compared with AspectJ, as classes

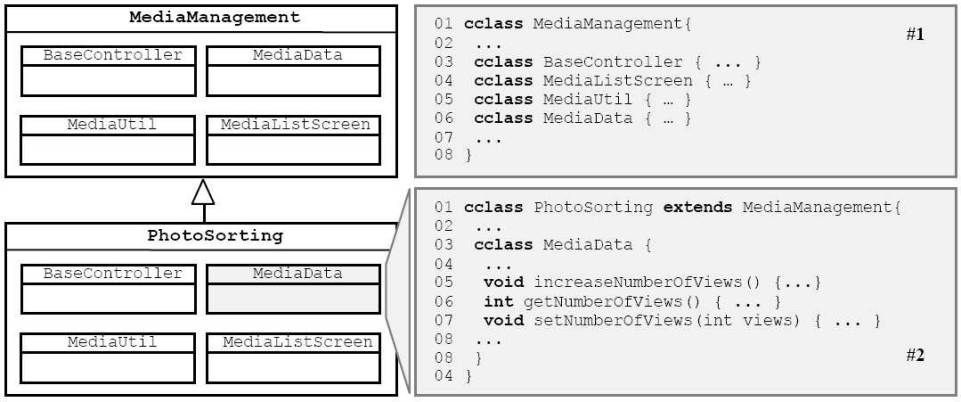

Figure 3.4: CaesarJ example of MobileMedia

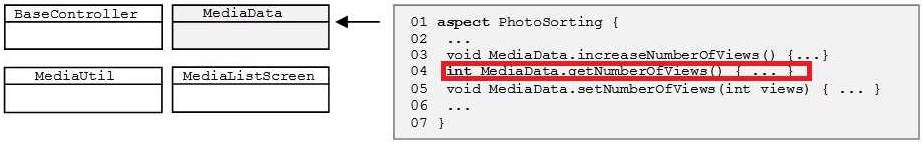

Figure 3.5: AspectJ example

in CaesarJ can be partially overridden. This value was calculated based on the number of changes presented in Table 3.2. Basically, we calculated the percentage difference between AOP and FOP iBatis versions.

```
01 cclass Favourite extends IListControl{
02   ...
03   void showImageList(..){
04     if (favorite) {
05       if (images[i].isFavorite())
06         imageList.append(..);
07     }
08   }
09   ...
10 }
```

Figure 3.6: Modularization of features with virtual classes

AspectJ fails in promoting stability of crosscutting functionalities. One of the maintenance scenario of GameUP involves `Persistence`, a crosscutting feature. This feature involves modules related to pieces, boards and user interface. These modules and methods are refined in order to allow storing both each piece type and the co-ordinates of its position in the board. The system also needs to provide the user with interface elements to save and load the game. As a consequence, several modifications in AspectJ implementation were needed due to the volatility of pointcuts and intertype declarations. Moreover, it was hard to specify a set of pointcuts and to refine the collaboration among modules; it was necessary a considerable refactoring effort to expose parts of original application code that would be advised by the aspects. In

```
01 cclass MListControl {
02   ...
03   void showMediaList(..){...}
04   ...
05 }
06
07 cclass Favourite extends MListControl{
08   ...
09   void showMediaList(..){
10     if (favorite) {
11      if (medias[i].isFavorite())
12        mediaList.append(..);
13     }
14   }
15 }...
```

Figure 3.7: Slice of code partially modified in CaesarJ

fact, the extent of modules modified during the integration was widely scoped. As a result, such refactorings were responsible to make the use of AOP more cumbersome.

Wrappers, pointcuts and intertype declarations can go hand-in-hand. The use of wrappers can be evaluated in some of the GameUP maintenance scenarios. One of them includes the customization of board tiles in the JHess game. In this scenario, there was no difference in terms of stability for CaesarJ and AspectJ. The AspectJ solution was based on the use of pointcuts and intertype declarations whereas the CaesarJ code focuses on the use of wrappers. We observed that the use of wrappers presented the same limitations of the use of intertype declarations. In fact, wrappers proved to be very tied to syntax, in particular, to the name of the types that are adapted by the wrapper. In modules that use wrappers, the changes adapt the wrapper to binding a proper class from context of the original application. In the evolution, the modules that call the constructors of wrappers also require modifications in the original code. In this sense, our evaluation suggests that the use of wrappers leads to modules with the same degree of stability when compared to those which are implemented using intertype declarations.

Figure 3.8 illustrates the use of wrappers in a piece of code. Note that there are changes in the wrapper code (`ColorSchemaBinding`). These changes were required to adapt a new kind of virtual class (`Gui`) and the module that instantiates the wrapper (`MainFrame`).

```
public class ColorSchemaBinding wraps GuiJhessUserGui {
  ...
  public void initComponents () {
     ...
     wrappee.getContentPanegetViewJMenu().add(comp);
  }
}

public cclass MainFrameJhessUserGui {
  ...
  protected void initinitComponents() {
    ...
    ColorSchemaBinding(this).initComponents();

  }
}
```

Figure 3.8: Example of modifications using Wrappers

3.2.3

Discussion: Are Conventional Metrics Indicators of Program Stability?

According to our findings, the use of composition mechanisms of AOP and FOP tends to produce systems with a high degree of modularity and stability (Sections 3.2.1 and 3.2.2, respectively). Table 3.3 summarizes the analysis of AspectJ and CaesarJ. The quality attributes are listed in the first column, the worst and best percentage of superiority in relation to the second position are given in the second column. These percentages in second column were calculated taking into consideration the values presented in Table 3.2. The name of the programming techniques that presented superiority is presented in the third column. As we can see, CaesarJ is the lead in terms of stability and modularity.

Table 3.3: Quality Attributes vs Advanced Programming Techniques

Quality Attribute	**Advantage (worst - best)**	**Advanced Programming Technique**
Modularity	3% - 8%	CaesarJ
Stability	10% - 71%	CaesarJ

Figure 3.9 presents a noticeable trend involving the two quality attributes: modularity and stability . This trend is illustrated using MobileMedia application as a representative case. However, it applies for all the target applications (see Apendix A). Modularity is expressed through the measures of coupling as it represents the behavior of the other modularity metrics as well (*e.g.*, DIT metric). It is clear that while the degree of stability varies along the versions, the degree of modularity tends to decrease. This is a strong evidence that modularity is not a good indicator of stability (Section 3.2.2). On the other hand, Figure 3.10 illustrates the same measures for coupling considering the Java versions of the MobileMedia application as a representative example. As

we can observe, different from what happens to the use of advanced composition techniques, coupling measures are good indicators of stability for OO programs.

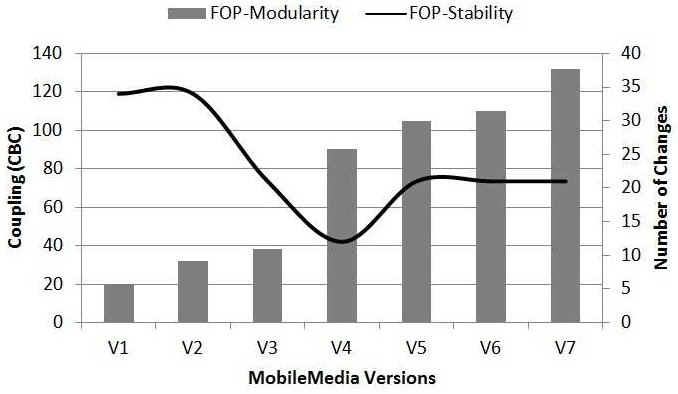

Figure 3.9: Modularity vs Stability for CaesarJ

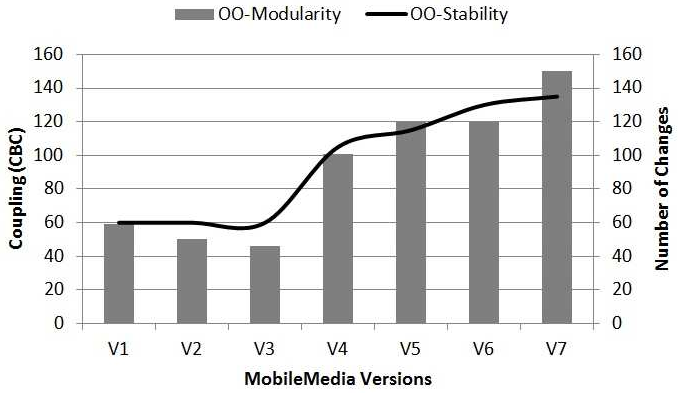

Figure 3.10: Modularity vs Stability for OO

Combined modularity metrics fail in indicating program stability. An additional analysis can be carried out taking into consideration the combination of coupling metrics that considers two coupling forms between composition code and the original code: CBC and DIT. The Figure 3.11 illustrates the relationship between DIT and stability. In Figure 3.9 the stability is associated with CBC. Considering these two figures, we can observe that both DIT and CBC increases together. However, the stability increase and decrease along the evolution. As a consequence, it is not possible to establish a relationship of these two combined metrics with the variation of stability. As a result, we can conclude that DIT and CBC when combined do not indicate program stability as well.

Composition code particularities are harmful to program stability. It is important to highlight that most of the differences associated with the

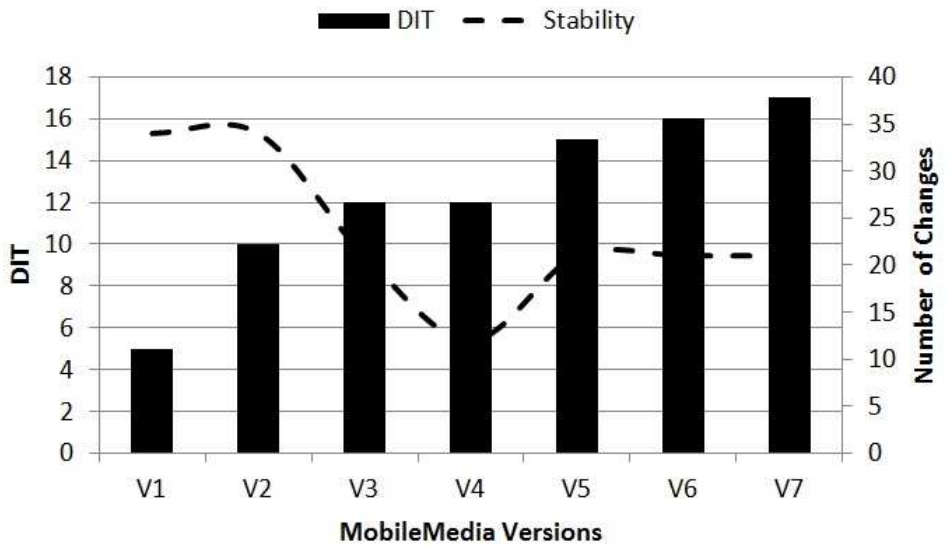

Figure 3.11: DIT vs Stability for CaesarJ

use of AOP and FOP mechanisms, such as intertype declaration and virtual classes, take place due to some particularities of the composition code. For instance, the use of intertype declaration makes evident the volatility of the composition code. This means that the use of this mechanism usually leads to the break of composition dependencies between program elements. These dependencies are established in order to prepare the existing program code to work with composition mechanisms.

In addition, the scope of the composition code goes beyond the dependencies explicitly declared in the program source code. We identified that there are many other indirect and thus implicit dependencies that are harmful to composition-enriched program stability. For instance, more than often a composition realized by a single method call change a value of an attribute in a given module. However, the original value of this attribute is used by another module - an aspect. Then, the scope of the aforementioned composition is not limited to module that is target of the method call, because there are other modules, in this case the aspect, that implicitly joins the composition, but it is not directly visible in the composition statement.

With these observations, we come to the conclusion that similar modularity results are both achieved with AspectJ and CaesarJ. Comparing all the measures, both techniques tend to exhibit almost the same level of modularity, which was clearly better than the level achieved with Java. The difference among all the AspectJ and CaesarJ measures does not exceed 5%. However, the different performances in stability cannot be explained by modularity

metrics as those differences are explained by the particular properties of the composition code produced.

3.3 Related Work

This section discusses prior works and constraints that somehow have inspired this study or have guided it. The mechanisms provided by the CaesarJ language have been studied in the direction of SPL development (MEZINI and OSTERMANN, 2003, MEZINI and OSTERMANN, 2004, APEL *et al.,* 2008). However, these studies do not evaluate different composition programming language, such as CaesarJ, in terms of software stability. Yet in the domain of composition mechanisms, Roo *et al.* (ROO *et al.,* 2008) proposed the language AspectJ on the top of composition filters studies. However, none of them (AKSIT *et al.,*, 1991, AKSIT and BERGAMANS, 1998), similar to CaesarJ studies, was focused on stability.

Figueiredo *et al.* (2008) present a case study in which they assess quantitatively and qualitatively the positive and negative impacts, in terms of modularity and stability, of AOP on a number of changes applied to both the core architecture and variable features of software product lines. However, in general, most of the empirical investigations only concentrate on the qualitative analysis of the modularization process (APEL and BATORY, 2006, ALVES *et al.,* 2007). Apel *et al.* (2008) (APEL *et al.,* 2008) contributed with an evaluation where they did not embrace multiple composition mechanisms regarding stability. The problem is that all empirical studies developed so far tend to carry out a narrow analysis, focusing solely on either modularity or stability (FIGUEIREDO *et al.,* 2008a). Finally, Gurgel *et al.* (GURGEL *et al.,* 2010) reported a qualitative evaluation of using the mechanisms supported by CaesarJ and AspectJ language to compose different design patterns. However, they did not conduct a quantitative analysis on program stability.

3.4 Threats to Validity

In our study, the *conclusion validity* threats are related to the implementation treatments. The versions were implemented by one AOP developer, who has a good knowledge on the programming languages involved in the study, namely

AspectJ and CaesarJ. Even though the developer has not developed the AspectJ original versions of the SPLs, he had a partial knowledge of the possible modifications in future versions. However, we tried to minimize this threat by involving independent CaesarJ and AspectJ researchers in order to check the quality of the design and code produced.

A threat to *construct validity* includes the suite of metrics used for quantifying changes and modularity properties. We used the CBC and DIT metrics that allowed us to evaluate the modularity properties, such as dependencies of the core/variable modules. We adopted these metrics because they were all empirically found to have correlation with design stability (GARCIA *et al.,* 2005, FIGUEIREDO *et al.,* 2008a).

Threats to *internal validity* reside on alignment rules used to implement the CaesarJ and AspectJ versions. To reduce this threat, we performed a detailed analysis of the AspectJ code of the SPLs in order to reduce the inconsistencies in the pointcut interfaces and not propagating problems from the original AO implementations. It was necessary to ensure the quality of design in all versions and to do a fair and equitable comparison.

Threats to *external validity* are conditions that allow results generalization. In order to minimize this threat, we chose a SPL that had already been used in another empirical study (FIGUEIREDO *et al.,* 2008a) and another well-known case study in the development of Java-based applications. These applications are representative and have a significant size. This way, they enabled us to observe the differences among the results. However, it is still necessary to conduct other evaluations with other evolving systems to be able to provide more evidences regarding our conclusions.

3.5 Summary

Gathering knowledge to identify which composition mechanisms achieve a better stability is particularly important due to many reasons. First, software engineers need to be better informed about which modularity mechanisms can maximize stability of a system. Second, it is important to know which of these particular mechanisms tend to promote positive and negative effects over stability of software and how to quantify these effects. In this context, this chapter reported a comparative assessment of AspectJ and CaesarJ in the context of evolving systems development. Our study confirms that the use of

CaesarJ collaboration interfaces and virtual classes tend to increase the degree of stability.

We have also observed a number of new interesting outcomes as discussed through Section 3.2. For instance, we found that modularity properties, as fostered by programming techniques, do not seem to be the key factor to determine the degree of stability. According to our experience in the target systems cases (Appendix A), the different composition mechanisms exerted the main influence on stability superiority (or inferiority) of a technique. This justifies why AspectJ presented the best modularity results, but CaesarJ was superior in terms of stability. In addition, the same evaluation presented in this chapter was carried out using a third advanced programming technique namely Composition Filter (DANTAS and GARCIA, 2010) and the findings were similar to those findings presented in Section 3.2.

As a consequence, there is a need for measurement frameworks (Chapter 4) able to capture the composition properties and quantify their impact on software stability. The idea behind this investigation is to identify if composition properties are correlated with program stability or not. As these properties are not explicit in the source code, there is also a demand for strategies to make them explicit in order to minimize the maintenance effort since the early stages of software development (Chapter 5). A viable alternative seems to be make the composition properties specification available already in the composition design.