2

Background and Related Work

"Try to relax! There is always a way."

Isela Macia
PhD classmate - class 2009 - PUC-Rio

Composition mechanisms supported by advanced programming techniques are used to define the binding of two or more modules. These mechanisms provide a variety of means to structure programs since their design phase, which leads to gains in the program modularity (FIGUEIREDO *et al.,* 2008a, FIGUEIREDO *et al.,* 2009). However, these gains come at a cost: developers now need to understand and deal with possible effects of composition code properties on the quality attributes of a program. A prominent impact of those properties is on program changes and thus its stability (Section 1.1). To deal with these changes is often required some reasoning about certain composition properties, which are not explicitly in the implementation or design artifacts. In order to deal with these effects, developers need first to have means to quantify the composition properties impact on stability.

Unfortunately, it is questionable if conventional modularity metrics capture the nuances of composition properties. Therefore, they might not provide appropriate means for indicating stability of an evolving program. In fact, there is no empirical study building this knowledge. In addition, there is no empirical study about program stability, focusing on whether quantifiable composition properties affect this quality attribute. In addition, there are also no means to explicitly specify composition properties in order to facilitate the reasoning about the composition and their prominent changes.

This chapter presents and discusses relevant topics to our research work. These topics vary from composition mechanisms to techniques for composition design. Moreover, we are particularly interested in assessing the correlation of composition properties and program stability. We also discuss existing limitations of related work that have inspired our research questions (Section

1.3). Section 2.1 presents a subset of composition mechanisms evaluated in the empirical studies of this thesis. Conventional software metrics, typically used to quantify modularity in software programs, are discussed in Section 2.2. Section 2.3 presents empirical studies regarding stability of programs produced with advanced programming techniques. Section 2.4 discusses existing work concerned in providing explicit support for composition design. Finally, Section 2.5 summarizes this chapter.

2.1 Composition Mechanisms

Software is typically decomposed into separate parts, which we called modules. Separation into modules allows programmers to compose pieces of the program that are functionally related, or that address closely similar responsibilities (NIERSTRASZ and MEIJLER, 1995). In this context, Aspect-Oriented Programming (AOP) and Feature-Oriented Programming (FOP) have been around and begun to get more attention from industry and academia (NARAYANAN *et al.,* 2006, HOHEENSTEIN, 2006, KULESHOV, 2007, MEZINI and OSTERMANN, 2002). The use of these techniques was mainly driven by their composition mechanisms, which are used to realize program composition (APEL *et al.,* 2008, HOFFMAN and EUGSTER, 2008). Program composition refers to the process of binding two or more program modules by means of composition mechanisms (NIERSTRASZ and MEIJLER, 1995).

Composition mechanisms also differ among them in the way they support the realization of compositions. Therefore, the choice for a particular composition mechanism influences the way of how software is structured. For example, inheritance mechanisms allow the definition of abstractions that may evolve in several ways and they are supported in different advanced programming techniques, such as AOP (*e.g.*, aspect inheritance) and FOP (*e.g.*, virtual classes). These techniques are attractive because all of them provide gains regarding modularity when compared with conventional techniques, such as OO (FIGUEIREDO *et al.,* 2008a). Table 2.1 presents the AOP and FOP composition mechanisms to be studied in this research work, which are presented in Section 2.1.1 and 2.1.2, respectively.

Table 2.1: A representative Set of Composition Mechanisms

Programming Technique	Composition Mechanism
AOP	Aspect Inherintance
	Pointcut-Advice
	Intertype Declaration
	Declare Precedence
FOP	Virtual Class
	Mixin Composition
	Wrappers
	Collaboration Interfaces

2.1.1

Modularity with AOP Mechanisms

In this section, an overview of AOP composition mechanisms is presented. In this context, AspectJ is presently the most popular and widely used aspect-oriented programming language. AspectJ provides several mechanisms to enable the decomposition of AOP system into modules called *aspects*. An aspect refers to both sets of join points, called pointcuts, and the associated advice. The advice consists of method-like portions of code that will be applied to the existing program. It is applied at all join points associated with that advice. Typical examples of join points are a method call, a method execution or a field access. A pointcut expression (or pointcut declaration) selects a set of join points by means of declarative expressions. A pointcut declaration is generally formed by patterns (*e.g.*, method signatures) and predicates. AspectJ also allows modifications of class structure and hierarchy through inter-type declarations and other declare-like expressions. Inter-type declarations provide a way to inject new fields or methods into an existing class using an aspect. In addition, as it is possible to define hierarchies of classes, it is also possible to define hierarchies of aspects. Aspects can extend classes, implement interfaces, and even extend other aspects. Finally, AspectJ provides declare precedence mechanisms for controlling aspect precedence in cases where more than one aspect shares the same jointpoint. The declare precedence construct must be specified inside an aspect.

Figure 2.1 demonstrates an example of modularization with aspects. The aspect `A1` intercepts the class `C1` adding the method `m1()` to it. Additionally, it also intercepts the class `C1` when its method `m2()` is called. At the same time, a second aspect `A2` intercepts the call of the method `m2()`. As the aspect `A1` and `A2` share the same jointpoint (`m2()` call), a third aspect, `A3`, is required

to control the execution precedence of the aspects. Finally, it is important to highlight that the aspect `A2` extends the aspect `A4`.

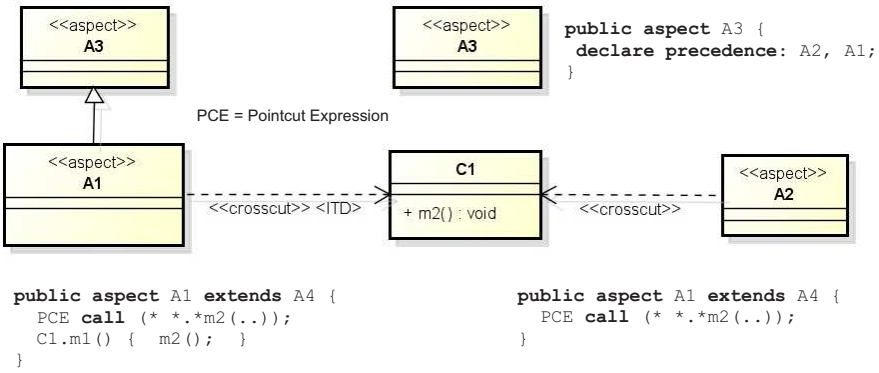

Figure 2.1: AOP in AspectJ

2.1.2

Modularity with FOP Mechanisms

The composition mechanisms investigated in this section are supported by FOP techniques and presented using a CaesarJ programming language (ARACIC *et al.,* 2006). In recent years, the CaesarJ programming language attracted attention among researchers, due to its promising advantages in relation to flexible support for modularity (MEZINI and OSTERMANN, 2003). CaesarJ uses a joinpoint model similar to that of AspectJ (Section 2.1.1).

A CaesarJ implementation is a top-level CaesarJ class (cclass) that encloses all elements of the concrete implementation of the component that are potentially reusable across multiple cases and scenarios. Though it usually specifies concrete state and behavior, it is often abstract. Typically, a CaesarJ provides implementations to some, though not necessarily all, the inner classes declared in the collaboration interface. A collaboration interface (MEZINI and OSTERMANN, 2002) is an approach proposed by the authors of CaesarJ, in which an abstract CaesarJ class is used to specify, in abstract terms, a collaboration between several abstractions. This way, module integration typically comprises several program elements which developers wish to keep separate and be able to evolve separately.

Collaboration interfaces are the basis of CaesarJ binding. A CaesarJ binding is a top-level CaesarJ class that inherits from the collaboration interfaces and encloses the logic that glues the module to a specific program. Thus, application specific program elements should be placed in a CaesarJ binding. A CaesarJ

binding corresponds to the expected part of the component. It performs the same role as a concrete aspect in AspectJ. Both implementations and bindings are subclasses of the collaboration interface. Thus, we need a mechanism to compose them to yield the complete, unified module. The mechanism provided by CaesarJ is a form of mixin composition (BRACHA and COOK, 1990). The main idea of mixins is to specify additional functionality to not just one, but to an opened set of existing modules in a transparent and non-invasive way. The mixin composition mechanism is used to address this, through which a final component is generated. The composition is realized with the operator &.

In addition, CaesarJ also supports the use of virtual classes (MADSEN *et al.*,1989) and comprises the capability to treat inner, nested classes, polymorphically. We give to the term virtual the same meaning as in the context of the C++ language. In other words, they are class members that can be overridden in subclasses and whose actual implementation is dynamically bound, or late bound, according to the actual type of the object at runtime. Figure 2.2 demonstrates an example of modularization with virtual classes. The cclass `Root` implements the base functionality of a given application. The new cclass `Sun` refines cclass `Inner1` adding the method `getmaxX()`.

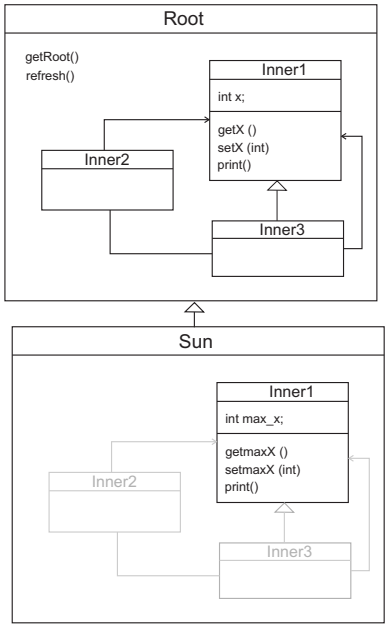

Figure 2.2: Virtual Classes

Since a functionality may need functionality of others classes it is important to have some forms of multiple inheritance. The mixin composition mechanism is used to address this, through which a final component is generated. The

composition is realized with the operator '&'. For instance, let us suppose that the class `Root` composes `Sun` with other functionality namely `Date`. The composition operator realizes a variant of multiple inheritance that linearizes the superclasses, thereby avoiding ambiguities such as duplication of inherited state as follows: `cclass Root extends Sun & Date { ... }`.

We also work with wrapper mechanisms, which are dynamic extensions of classes. Similarly to AspectJ's intertype declarations, they can be used to adding methods and attributes in a target class. They bind types from two domains. These bindings are explicitly called by invoking the wrapper constructor, which receives the object that will be adapted as parameter.

2.1.3

Modularity with Other Advanced Programming Techniques

There are other advanced programming techniques for modularizing programs, such as Compose* (COMPOSE PROJECT, 2012), delta-oriented programming (SCHAEFER *et al.,* 2010) and Traits (DUCASSE *et al.,* 2006). Compose* supports a composition filter model that extends the object abstraction in a modular and orthogonal way. Modular extension means that filters can be attached to objects expressed in different languages without modifying the definition of objects in these languages. Orthogonal extension means that the semantics of a filter is independent of the semantics of other filters. The modular and orthogonal extension properties distinguish the composition filter model from most other aspect-oriented techniques. Modular extension makes filters independent of the implementation. Orthogonal extension makes filters composable. Compose* aims at enhancing modularity and composition of modules through the composition filters model (BERGMANS and AKSIT, 1992). The idea is that objects can send messages between each other, *e.g.*, in the form of method calls or events. In the composition filters model, these messages can be filtered using a set of filters. Each filter has a type (*e.g.*, Dispatch, Meta, After, Before and Error), which defines the behavior that should be executed if the filter accepts the message and the behavior that should be executed if the filter rejects the message.

Regarding the delta-oriented programming technique, the idea is to provide composition mechanisms for implementing evolving programs also in a modular way. A program is divided into a core module and a set of delta modules. The core module comprises a set of classes that implement a complete software

product for a valid configuration. On the other hand, Delta modules specify changes to be applied to the core module in order to implement other products. A delta module can add classes to a product implementation or remove classes from a product implementation.

There are many other emerging advanced programming techniques that enable to combine fine-grained code units based on the use of traits. Traits-based composition promotes fine-grained program elements reuse only in methods. A trait is a set of methods that can be composed in arbitrary order (DUCASSE *et al.,* 2006). Traits are similar to mixins (ARACIC *et al.,* 2006), but whereas mixins can be composed only using the inheritance operation, traits offer a much wider selection of operations, including symmetric sum, method exclusion, and aliasing.

2.2

Program Stability and Modularity Metrics

In this section, we are particularly interested in discussing program stability (Section 2.2.1) and to survey existing assessment means associated with modularity metrics (Section 2.2.2). We focused or study on modularity metrics as they have found to be mostly related to software instabilities when conventional programming techniques are used (*e.g.*, Object-Oriented (OO)) (SHEO *et al.*,2008, BRIAND *et al.*,1999, BRIAND *et al.,* 1999).

2.2.1

Program Stability

Stability is one of the most desirable quality attributes in the context of program maintenance (ELISH and RINE,2003, FAYAD,2002, YOUNG,2005). However, there is no precise definition of stability. This means that its meaning is generally implied but not defined. Yau and Collofello (YAU *et al.*,1980, YAU and COLLOFELLO, 1985) have pioneered the study of stability at both levels of code and design. Both definitions are presented in terms of ripple effects over software versions. The computation of ripple effect is based on the effect that a change to a single program element will have on the rest of the program. Later, Alshayeb and Li (ALSHAYEB and LI,2005) and Kelly (KELLY, 2006) provided definitions accompanied by the concept of metrics. This definition is based on the general knowledge that, as changes are made to core program modules, other parts of the program may be affected due to the propagation of ripple effects. As an example of ripple effect, consider an interface which defines a set of method signatures. These methods are implemented by concrete classes that inherit from this interface. Modifications in this interface, such as adding, deleting, or modifying a method signature, require additional changes to be made to all classes that implement it. Therefore, in this case we say that changes in a particular module provoke ripple effects in other classes of the program. According to Kelly (KELLY, 2006) a program is considered stable when its interface or implementation is not undesirably modified and ripple effects (YAU *et al.*,1980) do not manifest in the presence of changes.

We chose to observe the stability from the standpoint of the number of changes of modules because we focus on the assessment of stability based on module changes, which are a direct consequence of ripple effects. In addition, we also

decided not to choose one or more particular structural metrics (*e.g.*, size and modularity) because surely only a restricted set of instability would be captured, since other structural characteristics would be ignored. Moreover, the number of module changes is a better direct measure of changes and occurrence of ripple effects. In this context, the definition of stability used in this work follows:

Definition 2.1*(Program Stability)* A program is considered to be stable as changes in any of its program elements are unlikely to generate ripple effects, causing the propagation of changes.

Minimising the ripple effects of a program change is a difficult, time consuming, and error-prone activity, especially for badly designed programs. Therefore, some authors (ELISH and RINE,2003, YOUNG,2005) have proposed the use of metrics to quantify design stability. This discussion regarding metrics is presented in Section 2.2.2.

2.2.2

Modularity Metrics

In the context of Empirical Software Engineering, software measurement exerts an important role as it provides quantitative evidence of the software engineering solutions over the generated artifacts. Software metrics for modular programming, fostered by advanced programming techniques, must capture some composition properties (Section 1.1). There is no measure to characterize and quantify composition properties. In the state-of-the-art, developers and researchers are forced to assume that classical modularity measures can be used to determine the quality of the modular decomposition of a system. However, all of them are focused on measuring properties of program modules rather than properties of the composition code itself. As a consequence, it is questionable whether these conventional metrics are indicators of key quality attributes of a system, such as stability.

At the implementation level, several metrics have been proposed and used in the context of Object-Oriented (OO) design and programming (CHIDAMBER and KEMERER, 1994, HENDERSON-SELLERS, 1995, ETZKORMN and DELUGACH, 2000). Software metrics have been widely used in the OO software development in order to improve software maintainability and reliability. Recently, some of these metrics were adapted to the context of AOP (SANTANNA *et al.,* 2003) and FOP

(BARTOLOMEI *et al.,* 2006). The set of proposed metrics captures pieces of information associated with the project and its code in terms of internal quality attributes, such as coupling and cohesion. The goal of these metrics is to provide a better modularization of the source code and thus also support analysis and prediction of quality attributes, such as stability. The metrics proposed by Sant'Anna *et al.* (SANTANNA *et al.,* 2003) were already evaluated and validated through a range of empirical studies. However, there is a lack of metrics to confidently evaluate the composition mechanisms described in this research work (Sections 2.1.1 and 2.1.2).

Similar to (SANTANNA *et al.,* 2004), Ceccato and Tonella (2004) adapted the object-oriented metrics proposed by Chidamber and Kemerer (CHIDAMBER and KEMERER, 1994) to the context of AOP. They also proposed new metrics for measuring coupling that results specifically from AOP mechanisms. These metrics take into consideration the crosscutting degree of an aspect and the coupling on advice execution. However, no empirical evaluation on composition properties was conducted. Furthermore, nothing is known about the use of these metrics for evaluating other techniques, such as FOP. The Ceccato and Tonella's metrics were empirically evaluated by Shen *et al.* (HAIHAO *et al.,* 2008). As a result, Ceccato and Tonella's metrics evolved them. The effects caused by pointcuts is now considered in the Shen *et al.* metrics. Similarly, their metrics quantify the crosscutting degree of an aspect caused by intertype declarations and the coupling on advice execution caused by advices. However, they did not correlate their metrics with stability of the modules using different advanced programming techniques. Shen *et al.* (HAIHAO *et al.,* 2008) extended their prior work in a study that evaluates their metrics when applied to check the maintainability of evolving aspect-oriented programs. However, they do not take into consideration the impact of composition properties on program stability.

Burrows *et al.* (BURROWS *et al.,* 2010) analyzed the feasibility and effectiveness of using coupling metrics as indicators of fault-proneness in aspect-oriented programs. However, they did not correlate their metrics with stability of the modules, but the existing relation of a set of metrics with the fault-proneness of aspect-oriented programs. Regarding software stability, Yau and Collofelo (YAU and COLLOFELLO, 1985) presented design stability measures which indicate the potential ripple effect characteristics due to modifications of the program at the design level. Further, Kelly (KELLY, 2006) used conventional metrics for assessing modularity and stability of software decompositions. However, she did not considered different composition mech-

anisms supported by advanced programming techniques. To the best of our knowledge, existing metrics for advanced programming techniques were only focused on AOP (GREENWOOD *et al.,* 2007, FIGUEIREDO *et al.,* 2008a). To date, however, there is limited empirical knowledge about the relationship between the use of these metrics to indicate or predict the degree of program stability.

In this direction, we developed a first evaluation about the impact of advanced programming techniques on program stability. We were particularly interested in analyzing instabilities which are undesirable (*i.e.*, those generated by the occurrence of ripple effects) for the software maintenance. We have used the mechanisms supported by two programming languages, which are representative of AOP and FOP technique respectively: AspectJ and CaesarJ.

2.3

Empirical Studies on Stability vs. Advanced Programming Techniques

Empirical Software Engineering aims at improving software engineering solutions through empirical methods. According to (BASILI *et al.,* 1999), empirical studies are the most efficient way of gathering knowledge that can be used to improve the quality of the software engineering techniques and methods. For this reason, it is important to identify and evaluate the use of advanced programming techniques through empirical assessments.

It has been widely recognized that the task of evolving software systems should not be detrimental to the software stability. Gathering knowledge to identify which programming techniques achieve better program 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, some of composition mechanisms supported by these techniques, such as intertype declarations and virtual classes can be considered competitive. Finally, it is important to know which of these particular mechanisms tend to promote positive and negative effects on software stability.

Regarding empirical studies of advanced programming techniques, the understanding is that better stability is always achieved using such composition mechanisms (KICZALES *et al.,* 1997, CZARNECKI and HELSEN, 2006, GRISWOLD *et al.,* 2006). The first steps in this direction were given by Mezini and Ostermann (MEZINI and OSTERMANN, 2002). They evaluated the composition mechanisms provided by the CaesarJ language in

the direction of software product line development. However, these studies do not evaluate CaesarJ in terms of software stability. Apel and Batory (APEL and BATORY, 2006) also discuss the use of aspects or features in the software development process. However, they did not assess stability and did not focus on different advanced programming techniques. Roo *et al.* (ROO *et al.,* 2008) proposed the Compose* programming language. By means of this language, the concept of AOP was included through the composition filters model. Other studies assessing the composition filter model were carried out. However, none of them (BERGMANS and AKSIT, 1992, AKSIT and BERGAMANS, 1998) focused on the evaluation of stability.

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 and also they did not analyze stability taking into consideration composition properties. Macia *et al.* (MACIA *et al.,* 2011) analyzed the impact of code smells in three evolving real-world systems. These systems were implemented using hybrid architectural decompositions - *i.e.* based on multiple architectural styles, including Model-View-Controller, Layers and Aspectual design. A large part of the system was implemented with Java and AOP. The architectural analysis was performed by comparing the intended architecture's models versus the actual architecture recovered from the source code. However, they did not focus on different composition mechanisms and also they did not analyze stability in the light of the composition effects.

In this context, little is known about the real influence of the use of advanced programming techniques over the stability of software artifacts. For instance, to the best of our knowledge there is no systematic analysis of modifications when changes in the composition code are required. In addition, there is a lack of studies in the context of advanced programming techniques with the goal of supporting the stability analysis when software systems evolve. For this reason, developers are not well informed about the stability problems associated with the use of composition mechanisms when they are developing.

2.4 Composition Design

In agreement with Whitehead *et al.* (WHITEHEAD,2007), composition design emerges as a fundamental activity to support developers when dealing

with the complexity level that is usually associated with the composition-enriched program diversity, which refers to the types and amount of modules. Composition design refers to the process of planning, using a graphical notation for instance, the composition implementation (*i.e.*, program elements that combine two or more modules). Based on this understanding, we claim that composition properties should be available to developers in modeling artifacts when they are coding. Unfortunately, techniques for explicit composition modeling do not focus on providing suitable means to specify composition properties. Different techniques for composition design have been proposed over the last years and each of them has a proper way to express compositions. However, none of them focuses on support the explicit modeling of composition properties. Even standard modeling languages, such as UML (ANDA *et al.,* 2006), do not provide specific abstractions to represent composition properties. In addition, there is a plenty of innovative model composition techniques, such as traditional composition algorithms (CLARKE, 2009), IBM RSA (IBM RSA, 2011), Epsilon (DAVID, 2000), MATA (WHITTLE *et al.*, 2009), Kompose (KOMPOSE, 2011)). Moreover, Chavez *et al.* (CHAVEZ and LUCENA, 2002, CHAVEZ *et al.,* 2005) have proposed a modeling notation for AOP-based projects. However, none of them explicitly support modeling of composition properties.

This lack of studies towards the evaluation of these modeling techniques, in the context of advanced programming techniques, has led to a major problem: there is a lack of guidance on managing composition properties using modeling techniques with the intention of alleviating the maintainability effort. In order to fulfil this gap, we investigated whether and how the availability of a detailed composition properties declaration help developers to evolve software minimizing the number of changes. Our investigation puts together a popular design modeling language, UML, and the notation proposed by Chavez *et al.* (CHAVEZ and LUCENA, 2002, CHAVEZ *et al.,* 2005) in order to express the composition properties in the composition design. The results of our investigation and the explanations about why we chose UML in our investigation can be found in Chapter 5.

2.5 Summary

This chapter presented a review of existing quantitative means for assessing stability of composition-enriched programs. Software metrics (Section 2.1)

are the basic means to assist software developers in improving the software stability. The metrics are intended to provide developers with efficient stability indicators, for instance, when composition mechanisms are used. Section 2.1 presented the composition mechanisms used in this study. Sections 2.2, 2.3 and 2.4 presented some existing metrics, empirical studies and modeling programming techniques and their limitations, respectively. Although studies on the use of advanced programming techniques have started to be reported in the literature (Section 2.3), the area of composition measurement is still in its infancy. This current scenario is also due to the lack of a standard terminology and formalization of composition properties. To address some of the limitations, we present in Chapter 4 a generic measurement framework for composition properties. This framework is built based on the discussion about the role of modularity in program stability presented in Chapter 3.