1

Introduction

"Welcome to PUC-Rio! We have a long journey ahead.
Are you ready to start?"

Alessandro Garcia, March 2009
In the first meeting as Francisco's supervisor

PUC-Rio - Certificação Digital Nº 0912912/CA

The demand for incremental software design has been increasing over the last decades (KELLY, 2006). As a result, there is a continuous search for programming mechanisms that improve the composability of modules defined at the design and evolution stages. The term composability refers to the ability to bind software artifacts, such as implementation level modules, in different combinations, facilitating the realization of evolving software changes (CLARKE, 2009). In this context, a growing number of advanced programming techniques has emerged to support more expressive means to define module compositions (KICZALES *et al.,* 1997, PREHOFER, 1997, BERGMANS and AKSIT, 1992). Well-known examples of such techniques are Aspect-Oriented Programming (AOP) (KICZALES *et al.,* 1997), Feature-Oriented Programming (FOP) (PREHOFER, 1997), Composition Filters (BERGMANS and AKSIT, 1992), Delta-Oriented Programming (SCHAEFER *et al.,* 2010) and Traits (DUCASSE *et al.,* 2006). Some programming languages even support a hybrid programming model based on the blend of previous programming models (*e.g.*, AOP and FOP), such as CaesarJ (CAESARJ, 2012).

It is well recognized nowadays that the richer composition mechanisms of advanced programming techniques help to improve software modularity (FIGUEIREDO *et al.,* 2008a, MEZINI and OSTERMANN, 2004, APEL *et al.,* 2008). As a consequence, they have been used to develop a variety of industrial and academic software applications (APEL *et al.,* 2008, FIGUEIREDO *et al.,* 2008a, MEZINI and OSTERMANN, 2003). In particular, both AOP and FOP have been recognized as promising techniques to develop software product lines (FIGUEIREDO *et al.,* 2008a,

ARACIC *et al.,* 2006, MEZINI and OSTERMANN, 2003). These systems require higher composability support than standalone software systems (CLEMENTS and NORTHROP, 2001). In addition, a number of popular development frameworks, for instance, have emerged to support AOP in distributed software systems, such as Spring (SPRING, 2013) and JBoss AOP (JBOSS, 2013). AOP has also been used to develop a wide range of industry-strength applications, such as Web-based systems (NARAYANAN *et al.,* 2006), embedded software (NARAYANAN *et al.,* 2006), data management frameworks (HOHEENSTEIN, 2006) and code generators (KULESHOV, 2007). On the other hand, FOP is still at embryonic stage and it is not widely known in industry.

Software development based on advanced programming techniques relies on a wide range of advanced composition mechanisms[1], such as wrappers, mixin composition (MEZINI and OSTERMANN, 2004), pointcut-advice and intertype declaration (KICZALES *et al.,* 1997), to facilitate the definition of module composition. In spite of their differences, these advanced programming techniques share a similar goal: fostering the decomposition of systems into more stable composable modules. A software system or a particular module is considered stable if ripple effects (KELLY, 2006, YAU and COLLOFELLO, 1985) do not manifest when its implementation is modified. 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. Thus, we can state that the less a composition depends upon program elements, the more the former is likely to be stable; the reason is that changes in any other program element are likely to be propagated to the former and cause changes in it.

However, in order to decompose systems into composable modules, the notation and semantics of composition mechanisms provide a shift in the complexity of a program. As a consequence, due to changes to a single composition specification the structure or behavior of modules tend to be changed as well, which in turn affect the program stability. Dealing with composition changes is particularly difficult because some reasoning about composition properties that are not explicit at both implementation and design levels is required. These properties refer to composition particularities, which must be explicitly specified to developers when they are coding so that they can prioritize

[1]From herein, advanced composition mechanisms is referred just as composition mechanisms.

PUC-Rio - Certificação Digital Nº 0912912/CA

stability. For instance, developers need to be prepared to deal with the great number and diversity of modules (*i.e.*, different types of modules) to generate composition-enriched programs.

Unfortunately there is no understanding in the state of the art about the composition properties that affect program stability. Thus, once composition properties manifest in programs where composition mechanisms are used, it is particularly important, to foster investigations actions to make clear how these properties affect the program stability. The problem is that these investigations cannot be carried out before understanding how composition properties impact on stability. To start off, it is also hard for developers to identify and distinguish each composition property from the others. This understanding is hard to be reached because the composition-enriched programs tend to be diversified in terms of types and number of modules. Therefore, in order to deal with the effect of composition properties on program stability, developers need to be supported with means to quantify them, which in turn relies on an accurate characterization of composition properties.

The remainder of this chapter is organized as follows. Section 1.1 defines the problem tackled in this thesis. Section 1.2 points out some limitations of related work. Section 1.3 describes the aims and research questions. Section 1.4 presents the thesis contributions. Finally, Sections 1.5 and 1.6 point out how this thesis is organized.

PUC-Rio - Certificação Digital Nº 0912912/CA

1.1 Problem Statement

The decomposition of programs in composable modules comes at a cost: the notation and semantics of composition mechanisms provide a shift in the complexity of a program. While part of the complexity is presumably factored out of software modules, their composition code - that code that defines the binding of two or more modules in a program - is often far from trivial to be understood and maintained. For instance, the use of composition mechanisms usually enables: (i) multiple modules are involved in the composition code, and (ii) the structure or behavior of one or more modules are potentially extended, merged or replaced by the elements of the other modules. There are also cases of additional mechanisms to determine the ordering of multiple compositions being applied to the same modules (KICZALES *et al.,* 2001). As a result, programmers have now to devote a large extent of their time to implement

the composition code. More importantly, when changing a program, 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 (GREENWOOD *et al.,* 2007). These non-local changes are likely to affect the stability of the whole program. Many negative stability effects might occur: (i) ripple effects (YAU and COLLOFELLO, 1985) might affect the composition code and module interfaces, and (ii) implicit composition properties can be unconsciously misunderstood and lead programmers to perform changes incorrectly.

Figure 1.1 illustrates an example of the composition complexity introduced by the use of composition mechanisms. Two of them, namely intertype declaration and pointcut-advice (AspectJ notation), are used in the composition specification (KICZALES and MEZINI, 2006). The aspect `A2` affects the structural behavior of class `C1` by introducing new methods `m1()` and `m2()`. The latter changes the semantics of the base module by intercepting calls to the method `m4()`. In addition, `A1` and `A2` shares the same joinpoint: the `m4()` call in `C1`. Thus, a precedence mechanism (KICZALES and MEZINI, 2006) is used in the aspect `A3` to declare the order of the compositions defined within `A1` and `A2`, when they are applied to the same target modules (*i.e.*, the class `C1` and its subclasses in this case).

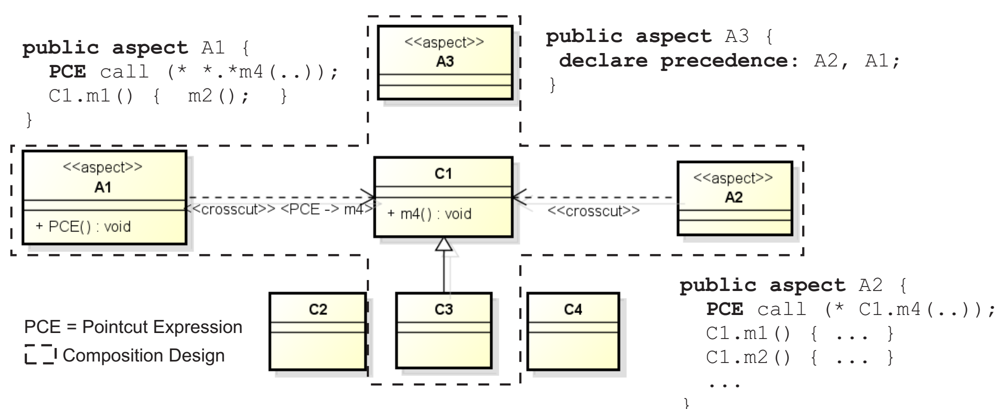

Figure 1.1: Composition Example using AspectJ notation

It is important to highlight that the composition code often requires reasoning about composition properties that are explicitly declared in the composition code. For instance, the pointcut `PCE` is intercepting all the calls to methods whose name ends in `m4` (`*.*m4()`). These calls are scattered through many modules of the system and determine the scope of this composition. Thus, programmers need to analyze the name of all the methods in order to confirm

PUC-Rio - Certificação Digital Nº 0912912/CA

that the composition was correctly implemented and no wrong method has been picked out. In addition, composition mechanisms, such as intertype declaration, virtual classes and superimposition affect the structure of a program, for example, by adding a method to a class or by changing the inheritance structure. This situation happens when a method is wrongly overridden by another method or a cyclic inheritance is introduced by the composition.

Taking into consideration that motivating example, it becomes clear that certain composition properties are not explicitly specified at design time, implementation time or both. Composition design is concerned about how program elements are represented and combined in order to define a coherent program functionality (see Figure 1.1). Moreover, they cannot be easily inferred from the source code either. In this context, our problem statement is divided into three sub-problems. First, we need to produce ways for developers to deal with composition properties in the absence of a proper composition properties characterization. Second, it is also needed to quantify their impact on program stability. Otherwise, their effect on program stability cannot be determined. Finally, we need also to identify means to alleviate the developers' maintainability effort when they need to deal with composition properties.

Sections 1.1.1 and 1.1.2 discuss, respectively, the characterization of composition properties and importance of quantifying them in order to prioritize program stability. Finally, Section 1.1.3 discusses the importance of studying how developers can alleviate their effort in changing programs; for instance, if the availability of design information about composition properties can better support them in their implementation maintenance tasks.

PUC-Rio - Certificação Digital Nº 0912912/CA

1.1.1
The Problem of Characterizing Composition Properties

As aforementioned, the composition code management demands extra reasoning about each composition in order to control the propagation of changes. The analysis of the motivating example (Figure 1.1) demonstrates the need for extensive reasoning about the composition code. A number of properties of the composition code, independently of the mechanisms applied, need to be considered. For instance, the identification of which elements of the program modules are explicitly affected by the composition is required. In other words, it is required to identify the composition scope. The scope of the composition over the base code is not given only by these explicit references, but

also in terms of elements indirectly affected or considered by the composition semantics. There are also cases where the composition design requires the preparation (*e.g.*, refactoring) of one or more target modules. Figure 1.1 illustrates how these composition properties can make program comprehension quite challenging. The composition code in `A1`, `A2` and `A3` establishes direct and indirect dependencies with modules `C1`, `C2`, `C3`, `C4` and many others in the system.

Given the aforementioned issues we consider that the existing strategies are unable to differentiate those program elements that belong to the composition code from the other program elements. Therefore, it becomes clear that, in order to deal with composition properties effects on program stability, firstly we need to understand how to distinguish properties one from another. In other words, composition properties must be conceptually characterized. By means of this characterization developers will be able to determine which program elements belong to the composition code and which do not. This distinction will also allow them to specify the composition properties properly, opening the possibility to quantify their impact on program stability.

Therefore, the first problem can be described as follows:

> *There is no understanding about how to deal with composition properties, in terms of program stability, mainly because they need to be characterized.*

PUC-Rio - Certificação Digital Nº 0912912/CA

1.1.2
The Problem of Assessing the Impact of Composition Properties on Program Stability

Ideally, the (re)use of program modules through composition mechanisms should not require invasive modifications in the target modules. For instance, modifications should not be made to either interfaces or internal members of program modules as the program evolves. Similarly, other composition specifications (*e.g.*, other pointcuts and intertype declaration) should not be modified either. Otherwise, the program instabilities, provoked by these harmful modifications can also lead to the harmful program changes (KELLY, 2006, GREENWOOD *et al.,* 2007). In this fashion, positive and neg-

ative effects of adopting advanced programming techniques and their composition mechanisms need to be systematically investigated.

In this context, there is a lack of empirical studies on the impact of composition properties on program stability. This gap in the literature is mainly because there is no means to properly quantify the composition properties. In this context, the characterization of composition properties (Section 1.1.1) will work as a basis to identify which program attributes belong to the composition and which not. This characterization is the starting point for quantifying the impact of composition properties on program stability and how to deal with the program changes.

Thus, our second problem can be stated as:

> *Even after the composition properties are characterized, their impact on program stability is still unclear if there is no suitable means to measure the properties.*

1.1.3
The Problem of Alleviating the Maintainability Effort

Once the identification and quantification of composition properties are carried out, means to avoid or to alleviate unintended program changes are required. In this context, considering that the biggest difficulty to manage composition properties is the fact that they are not available to developers when they are coding, the most intuitive way to alleviate the developers' effort is to make the information about composition properties available to them in design artifacts since the early stages of development. In other words, developers need to be informed about the composition properties during their maintenance tasks in order to better prioritize program stability.

However, composition design based on mainstream modeling languages, such as the Unified Modeling Language (UML), does not provide means to explicitly specify the composition properties. This means that even if the developers have access to some composition information, they are not able to identify the details associated with the composition properties. As a consequence, developers need to extensively reason about the properties of module composition in order to accomplish program changes. In this context, our third problem relies on

PUC-Rio - Certificação Digital Nº 0912912/CA

analyzing the influence of having available composition properties specification on program stability. Our third problem can be described as follows:

> *Even considering that composition properties can be measured and their impact on program stability determined, there is a lack of guidance on managing them in order to alleviate the maintainability effort.*

1.2 Limitations of Related Work

This section discusses related work in the context of program stability and advanced programming techniques. Previous contributions in three key areas of research are relevant in this context: stability of composition-enriched programs, and quantification and specification of composition properties. Therefore, we discuss to what extent existing work could be used to circumvent the problems addressed in this research: (i) lack of characterization and quantification of code composition properties, and (ii) poor specification of composition properties. We concluded that existing work cannot be used to address those problems and, therefore, their limitations are made clear throughout this section. Section 1.2.1 presents the state of the art regarding empirical studies on stability yielded by advanced programming techniques, as well as the characterization and quantification of composition properties. In Section 1.2.2 we discuss the literature in terms of specification of composition properties at design level.

1.2.1 Program Stability vs. Composition Properties

There is only limited assessment of advanced programming techniques and their composition mechanisms on program stability. The problem is that all the empirical studies developed so far tend to carry out a narrow analysis by focusing solely on either modularity or stability (FIGUEIREDO *et al.,* 2008a, APEL *et al.,* 2008). In addition, they do not investigate the interplay of code composition properties and program stability. Apel *et al.* (APEL *et al.,* 2008), for instance, have contributed with an evaluation towards the use of ad-

PUC-Rio - Certificação Digital Nº 0912912/CA

vanced programming techniques. However, they did not embrace multiple composition mechanisms and did not perform any analysis of stability in terms of the composition properties. These limitations of these works are due to the lack of means to explicitly quantify the impact of these properties on program stability. 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 metrics can be used to determine the quality of the modular decomposition of a system (FIGUEIREDO *et al.,* 2008a), (BARTOLOMEI *et al.,* 2006), (BARTSCH and HARRISON, 2006). The properties quantified by these metrics vary from cohesion and coupling to method complexity. However, they are all focused on measuring properties of the module structure rather than the properties of composition code itself (KRUEGER, 1992).

The aforementioned limitations of the existing studies and metrics open gaps in the literature that cannot be fulfilled without: (i) characterizing the composition properties that are harmful for program stability; and (ii) quantifying their impact of program stability through suitable measurement means. Without a proper characterization and quantification of composition properties, it is not possible to objectively analyse their impact on program stability. These topics are discussed in Chapters 3 and 4.

1.2.2
Composition Design

Some research works have explored the impact of mainstream UML models on-software maintenance (AGARWAL and SINHA, 2003, BRIAND *et al.,* 2005, ANDA *et al.,* 2006, ARISHOLM *et al.,* 2006, DZIDEK *et al.,* 2008). However, there is no evidence regarding to what extent the use of UML models with explicit composition design improves the stability of programs. Genero *et al.* (GENERO *et al.*,2008) presented a family of experiments on the investigation whether the use of stereotypes improves the comprehension of UML sequence diagrams. They concluded that using stereotypes improves the comprehension of UML sequence diagrams. However, their study did not focus on the analysis of composition design using different advanced programming techniques. In particular, these studies can be complemented in order to derive information impact of using composition design during maintenance programming tasks in terms of stability.

PUC-Rio - Certificação Digital Nº 0912912/CA

The aforementioned limitations of the existing studies cannot be overcome without empirical findings on how and whether an enriched composition design with suitable composition properties declarations reduces the developer's maintenance effort during their programming tasks. This topic is discussed in Chapter 5.

1.3 Goals and Research Questions

This thesis aims at supporting developers in generating software with high stability when they use composition mechanisms. The general goal is achieved through the satisfaction of the following sub-goals.

- To perform experimental studies to evaluate the impact of using different composition mechanisms on program stability;
- To propose a measurement framework, rooted at empirical studies, to support the conceptual characterization and quantification of composition properties, which have influence on program stability;
- To investigate to what extent composition properties explicitly specified at design level better support developers on constructing stable programs and performing their changes;
- To develop a tool to support the programmers in the quantification of composition properties;

PUC-Rio - Certificação Digital Nº 0912912/CA

Therefore, in order to achieve the aforementioned macro goal, the following overall Research Question (RQ) needs to be answered:

How does the composition code affect the program stability?

This question has been decomposed in the following four research questions (RQ1, RQ2, RQ3 and RQ4):

RQ1 *How to objectively analyze the impact of using composition mechanisms on program stability?*

The goal of RQ1 is to analyze the impact of using different composition mechanisms on program stability. In this thesis, we decided to study the impact of the composition mechanisms supported by two techniques: AOP and FOP, which are represented by AspectJ and CaesarJ programming languages. Despite the popularity of both programming languages, they were mainly chosen because they support significantly-different composition mechanisms while having similar goals of improving programming stability. CaesarJ, in its turn, supports a combination of AOP and FOP mechanisms. In addition, their compiler implementations are open and proved to be robust enough during our pilot assessments. Equally important, there are also public reports of their successful adoption in industrial software development projects (CAESARJ, 2012, KICZALES and MEZINI, 2006). In particular, AOP continues exerting an influence on mainstream technologies, such as dependency injection with the Spring framework (SPRING, 2013). In this context, we aim at knowing which of these particular mechanisms tend to promote positive and negative effects over stability of programs and what the composition properties associated with these effects are. In order to answer RQ1 a set of exploratory studies was required. The findings and discussion derived from these studies are presented in Chapters 3 and 4.

PUC-Rio - Certificação Digital Nº 0912912/CA

RQ2 *Are modularity conventional metrics good indicators of composition-enriched program stability?*

The second research question (RQ2) aims at (i) evaluating whether the conventional modularity indicators, represented by coupling metrics, are effective enough to determine the stability of evolving programs, (ii) identifying what code composition properties influence program stability, and (iii) getting insights on how the impact of code composition properties can be quantified. In other words, we aim at identifying whether the use of composition mechanisms make conventional coupling measures unsuitable for indicating stability. We discuss the relation of conventional metrics and composition-enriched program stability in Chapter 3.

RQ3 *What is the impact of composition properties on program stability?*

In RQ3 we aim at investigating and quantifying what is the impact of the composition properties on program stability. For this end, we need first to characterize these composition properties. As a second step, a suitable

measurement framework, which is able to capture the composition properties nuances, is proposed and used for quantifying the impact of composition properties on program stability. The idea is to verify if composition properties are good indicators of stability or not. This way, we can also complement our investigation through ranking the composition properties according to their influence on software stability. These discussions are presented in Chapter 4.

RQ4 *Does the availability of richer composition modeling lead to better code stability than mainstream modeling?*

Finally, based on our conclusion in RQ3, we investigate whether the availability of design models enriched with explicit composition information can contribute to improve software maintenance in RQ4. For this end, we analyzed to what extent composition properties are correlated with stability of evolving programs. In order to carry out the analysis,we perform an investigation on how changes are correlated with composition properties. The research aims are twofold. First, we aim at evaluating whether the composition properties are related to stability of evolving programs. Second, our goal is to discuss some implementation factors that were detrimental to the program stability. RQ4 is answered in Chapter 5.

1.4 Contributions

This section briefly describes the contributions of this thesis, including: (i) empirical findings revealing the role of modularity in composition-enriched program stability, (ii) a measurement framework based on a meta-model and terminology used to characterize and quantify the impact of key properties of composition code on program stability; and (iii) empirical findings on how to reduce the developer's maintenance effort during their programming tasks. An overview of each contribution follows.

- **Empirical Findings on the Role of Modularity in Indicating Software Stability (RQ1 and RQ2).** At first, by means of empirical studies, we aimed at evaluating the influence of modularity on composition-enriched program stability. The studies undertaken in this phase (Chapter 3) were carried out using three different systems (see

PUC-Rio - Certificação Digital Nº 0912912/CA

Appendix A). To be more specific, these studies compare the effectiveness of different composition mechanisms on program stability and the role of the program modularity in its stability as well. This step is particularly important because (i) it provides concrete evidence associated with the impact of using different composition mechanisms on program stability, and (ii) it makes evident how to objectively indicate stability on composition-enriched programs. These studies revealed that coupling metrics – typically proposed and used as stability indicators – are not good indicators of stability for advanced programming techniques as they do not take into consideration the composition properties. These studies are related to two research questions, namely RQ1 and RQ2. Based on these findings, developers can understand how the choice of different composition mechanisms exerts different influences on program stability. The findings derived from this study is presented in details in Chapter 3.

– **Composition Measurement Framework (RQ1 and RQ3).** The proposed measurement framework is based on a meta-model, which is independent of particular program languages and programming techniques. The framework plays a role by supporting the characterization and quantification of composition properties by means of a language-independent-formalism and a suite of composition metrics. These metrics are the basic means for analyzing composition properties. They can be applied to programs developed with any advanced programming techniques. The measurement framework is presented in details in Chapter 4. The generality of the measurement framework is evaluated by using it in the instantiation and comparison of several and distinct composition-enriched programs (Appendix A). This framework is particularly important to developers as it serves as a means to quantify stability of programs regardless of programming techniques. Finally, we design and implement a tool, named `CoMMes`, which supports the use of the proposed framework.

  `CoMMes` is a tool that supports the measurement of composition properties at the implementation level. `CoMMes` supports: (i) the importation of the composition-enriched programs, and (ii) the application of the composition metrics. In Section 4.4 it is described each of these elements and the architecture of the `CoMMes` tool.

– **Empirical Findings about Maintenance Improvements with Explicit Composition Modeling (RQ4).** By means of an Internet-based experiment we investigated if the availability of design models, enriched with composition properties, provided benefits in terms of program sta-

PUC-Rio - Certificação Digital Nº 0912912/CA

bility when developers are performing code changes. Our investigation took into consideration recurring scenarios of evolving applications (see Appendix A). The experiment consists of two different groups of developers: one group using mainstream design models and the other group using design models enriched with composition details. This study is extremely important as it provides evidence on how to alleviate the developers' effort of maintaining composition-enriched programs. The discussion of the experiment results is presented in Chapter 5.

These contributions resulted in scientific publications presented in Table 1.1. There are also some indirect publications that emerged during the definition and conception of this thesis, which are described in Table 1.2.

1.5

Links between Research Questions, Papers and Chapters

The chapters and papers (Tables 1.1 and 1.2) that make up the main body of this thesis give answers to the research questions presented in Section 1.3. Figure 1.2 illustrates the relationship between both research questions and chapters and research questions and their main publications. Each chapter and paper included in the thesis covers some aspects of the research questions and goals. RQ1 and RQ2 are covered in papers 4 and 5, which discuss if coupling metrics are good indicators of program stability and how program stability can be better quantified (Chapter 3). RQ2 is also partially covered in paper 3. RQ3 addresses the impact of composition properties on program stability. This subject is discussed in papers 1,3,5 and 8 and Chapter 4 of the thesis. RQ4 is concerned with the impact of using models with explicit composition properties in software maintenance tasks. This topic is covered in paper 2 and Chapter 5.

1.6

Thesis Outline

The structure of the thesis is divided into six different chapters. Figure 1.3 illustrates the flows of ideas throughout the three core chapters (see the numbered black circles): (1) initially, we identified that modularity metrics are not good indicators of stability and for this reason composition metrics are needed; (2) to overcome this need a measurement framework rooted on

PUC-Rio - Certificação Digital Nº 0912912/CA

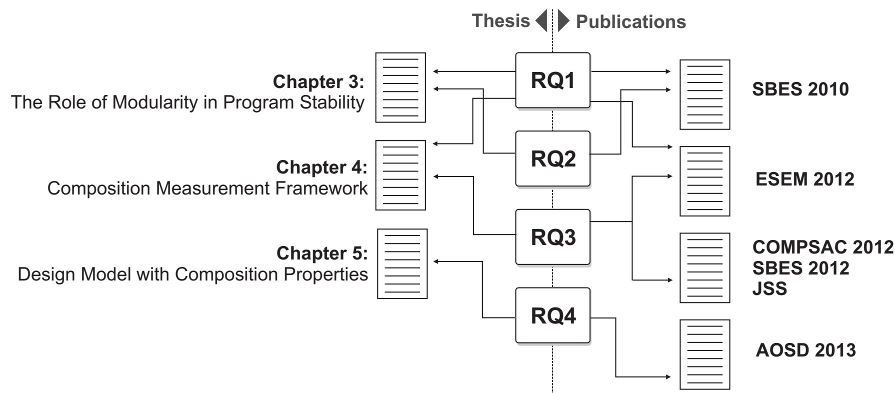

Figure 1.2: The relationships among RQs, Chapters and Main Publications

a set of composition metrics was defined and evaluated on the top of three evolving applications (Appendix A), where we verified that the composition metrics are good indicators of stability; and, finally (3) we gathered evidence about the benefits of explicit composition modeling and their usefulness to help participants in code maintenance tasks mainly with respect to stability.

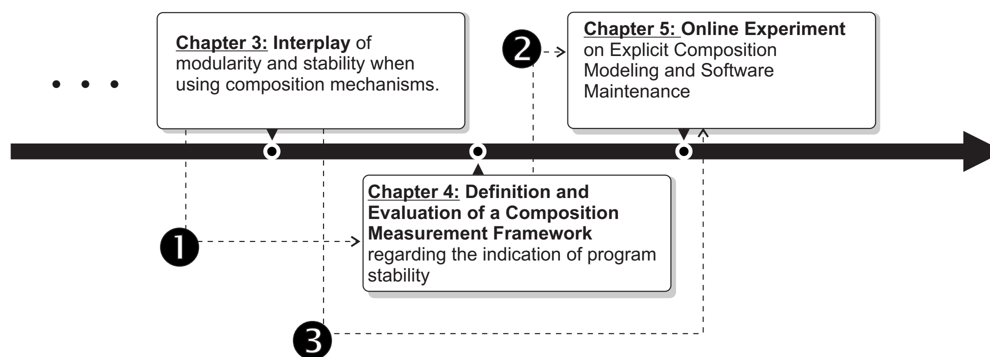

Figure 1.3: Thesis Overview

Chapter 1 has presented the introduction, discussed the problem that motivated this research, the purpose of the research, research questions and limitations of related work. In Chapter 2, we set the present research in the context of a literature review. In Chapter 3, we study the role of modularity in program stability. A measurement framework is presented and evaluated in Chapter 4. In Chapter 5, the evaluations are carried out in terms of the influence of explicit composition models on software maintenance tasks. Chapter 6 contains discussions of the results with references to research questions and objectives. In addition to the research contributions, directions for future research and conclusions are also included in this chapter.

PUC-Rio - Certificação Digital Nº 0912912/CA

Table 1.1: Direct Publications

#	Publication	RQ#
1	**Francisco Dantas**, Alessandro Gurgel, Alessandro Garcia, Camila Nunes e Cláudio Sant'Anna. On the Impact of Feature Integration Properties on Software Product Lines: A Study of Reuse versus Stability. Journal of Systems and Software, 2012 (submitted)	3
2	**Francisco Dantas**, Alessandro Garcia, Jon Whittle, João Araújo. Enhancing Design Models with Composition Properties: A Software Maintenance Study. Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD'13), Fukuoka, Japan, March 2013.	4
3	**Francisco Dantas**, Alessandro Garcia and Jon Whittle. On the Role of Composition Properties on Evolving Programs. Proceedings of the 6th International Symposium on Empirical Software Engineering and Measurement, ESEM.12, Lund, Sweden, September 2012.	2 and 3
4	**Francisco Dantas**, Alessandro Garcia. Software Reuse versus Stability: Evaluating Advanced Programming Techniques. Proceedings of the ACM SIGSoft XXIII Brazilian Symposium on Software Engineering (SBES'10), Salvador, Brazil, September 2010.	1
5	**Francisco Dantas** and Alessandro Garcia. Stability of Product Lines with Composition Filters: An Exploratory Study. Proceedings of the 1st International Workshop on Empirical Evaluation of Composition Techniques (ESCOT 2010), 2010.	2 and 3
6	**Francisco Dantas**, Eduardo Figueiredo, Alessandro Garcia, Cláudio Sant'Anna, Uirá Kulesza, Nélio Cacho, Sérgio Soares and Thaís Batista. Benchmarking Stability of Aspect-Oriented Product-Line Decompositions. In: 4th Workshop on Assessment of Contemporary Modularization Techniques (ACoM.10), 2010, Jeju Island. Proceedings of the 4th Workshop on Assessment of Contemporary Modularization Techniques (ACoM.10), 2010. p. 7-12.	1
7	**Francisco Dantas** (student), Alessandro Garcia (supervisor). Reuse vs. Maintainability: Revealing the Impact of Composition Properties. Proceedings of the International Conference on Software Engineering (ICSE'11) - Doctoral Symposium, Hawaii, USA, May 2011.	1,2 and 3
8	**Francisco Dantas**, Alessandro Gurgel and Alessandro Garcia. Towards a Suite of Metrics for Advanced Composition Mechanisms. Proceedings of the International Workshop on Empirical Evaluation of Software Composition Techniques (ESCOT 2011), 2011, Lancaster (UK).	2 and 3

PUC-Rio - Certificação Digital Nº 0912912/CA

Table 1.2: Indirect Publications

#	Publication
9	Bruno Cafeo, **Francisco Dantas**, Alessandro Gurgel, Everton Guimarães, Elder Cirilo, Alessandro Garcia, Carlos Lucena. Towards Indicators of Instabilities in Software Product Lines: An Empirical Evaluation of Metrics. Proceedings of 4th International Workshop on Emerging Trends in Software Metrics (WeTSOM 2013) at ICSE 2013, San Francisco, USA, May 2013. (accepted to appear)
10	Bruno Cafeo, **Francisco Dantas**, Alessandro Gurgel, Everton Guimarães, Elder Cirilo, Alessandro Garcia, Carlos Lucena. Analysing the Impact of Feature Dependency Implementation on Product Line Stability: An Exploratory Study. ACM SIGSoft XXVI Brazilian Symposium on Software Engineering (SBES'12), Natal, Brazil, September 2012.
11	Renato Novais, Camila Nunes, Caio Lima, Elder Cirilo, **Francisco Dantas**, Alessandro Garcia, Manoel Mendonça. On the Proactive and Interactive Visualization for Feature Evolution Comprehension: An Industrial Investigation. Proceedings of the 34th International Conference on Software Engineering (ICSE'12), Software Engineering in Practice, Zurich, Switzerland, June 2012.
12	Leandra Mara, Gustavo Honorato, **Francisco Dantas**, Alessandro Garcia, Carlos Lucena. Hist-Inspect: A Tool for History-Sensitive Detection of Code Smells. Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), Tools Session, Porto de Galinhas, March 2011. (extended abstract)
13	Leandra Mara, Gustavo Honorato, **Francisco Dantas**, Alessandro Garcia, Carlos Lucena. Hist-Inspect: Tool Support for History-Sensitive Analysis of Source Code. Proceedings of the ACM SIGSoft XXIII Brazilian Symposium on Software Engineering (SBES'10), Salvador, Brazil, September 2010.
14	Leandra Mara da Silva, **Francisco Dantas**, Gustavo Honorato, Alessandro Garcia, Carlos Lucena.Detecting Smells in Evolving Source Code: What the History Can Tell? Proceedings of the 4th Brazilian Symposium on Software Components, , Salvador, Brazil, September 2010.
15	**Francisco Dantas**, Camila Nunes, Alessandro Garcia, Uirá Kulesza and Carlos Lucena. Stability of Software Product Lines with Class-Aspect Interfaces: A Comparative Study. Proceedings of the 4th Workshop on Assessment of Contemporary Modularization Techniques (ACoM.10), 2010, Jeju Island.
16	Alessandro Gurgel, **Francisco Dantas** and Alessandro Garcia. Um Estudo de Composições de Padrões de Projeto em CaesarJ. In: V Latin American Workshop on Aspect-Oriented Software Development (LA-WASP 2010), 2010, Salvador. Proceedings of the IV Latin American Workshop on Aspect-Oriented Software Development (LA-WASP 2010), Salvador, Brazil, September 2010.
17	Nélio Cacho, **Francisco Dantas**, Alessandro Garcia and Fernando Castor. Exception Flows made Explicit: An Exploratory Study.Proceedings of the ACM SIGSoft XXIII Brazilian Symposium on Software Engineering (SBES'09), Fortaleza, Brazil, October 2009. (accept. rate 18%)
18	Alessandro Gurgel, **Francisco Dantas**, Alessandro Garcia and Claudio Sant'Anna. Integrating Software Product Lines: A Study of Reuse versus Stability. Proc. of the 36th IEEE Computer Software and Applications Conference (COMPSAC 2012), Izmir, Turkey, July 2012.

PUC-Rio - Certificação Digital Nº 0912912/CA