Reuse vs. Maintainability: Revealing the Impact of Composition Code Properties

Francisco Dantas
Opus Research Group – Software Engineering Lab
Informatics Department, PUC-Rio, Brazil
fneto@inf.puc-rio.br

ABSTRACT
Over the last years, several composition mechanisms have emerged to improve program modularity. Even though these mechanisms widely vary in their notation and semantics, they all promote a shift in the way programs are structured. They promote expressive means to define the composition of two or more reusable modules. However, given the complexity of the composition code, its actual effects on software quality are not well understood. This PhD research aims at investigating the impact of emerging composition mechanisms on the simultaneous satisfaction of software reuse and maintainability. In order to perform this analysis, we intend to define a set of composition-driven metrics and compare their efficacy with traditional modularity metrics. Finally, we plan to derive guidelines on how to use new composition mechanisms to maximize reuse and stability of software modules.

Categories and Subject Descriptors
K.6.3 [Software Management]: Software Maintenance;
General Terms
Measurement, Experimentation, Languages.
Keywords

1. MOTIVATION
With the pressing need for building reusable software, there is an continuous search for programming mechanisms that improve module composableiy. Over the last years, a growing number of programming techniques (e.g. [1][2][3]) have emerged to support more expressive means to define module compositions. Well-known examples of such techniques are aspect-oriented programming (AOP) [1], feature-oriented programming (FOP) [2], and composition filters (CF) [3]. They support a wide range of advanced composition mechanisms – such as, \textit{mixin composition} [4], \textit{pointcut-advice} [1], and \textit{superimposition} [3], but share a similar goal: fostering the decomposition of systems into modules that can be better reused and changed independently. From herein, advanced composition mechanisms as referred just as composition mechanisms, and the produced composition code is simply referred as composition

\textbf{Complexity Shift: from Modules to Composition.} To achieve this goal, the notation and semantics of composition mechanisms provide a shift in the complexity of a program. While part of the complexity is presumed factored out of software modules, their composition code is often far from trivial to be understood, reused and maintained. For instance, the use of composition mechanisms usually enables that: (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 [1]. 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 [5].

\textbf{A Motivating Example.} Figure 1 illustrates an example of the complexity of the composition code introduced by the use of composition mechanisms. Two of them, namely \textit{intertype declaration} and \textit{pointcut-advice} (AspectJ [1] notation), are used in the composition specification. The first one affects the structural behavior of class \texttt{C1} by introducing new methods (\texttt{m1()} and \texttt{m2()}). The latter changes the semantics of the base module by intercepting calls to method \texttt{m4()}. In addition, a precedence mechanism [1] is used in \texttt{A3} to declare the order of the compositions defined within \texttt{A1} and \texttt{A2}, when they are applied to the same target modules (i.e. the class \texttt{C1} and its subclasses in this case). It is important to highlight that composition specifications with composition mechanisms often require the reasoning about modules not explicated declared in the composition code. For instance, the pointcut PCE is intercepting all the \textit{calls} to methods which name ends with \texttt{m4} (*., \texttt{m4}()). These calls are scattered through many modules of the system. This also implies that programmers need to analyze the name of all the methods in order to certificate that the composition was correctly implemented and no wrong method was picked out.

\textbf{Reuse vs. Maintainability: The Impact of Composition Code.} It is well recognized nowadays that advanced composition mechanisms [1][2][3] help to improve modularity [10][12][13]. However, given the new complexity flavors of composition code, it is questionable if the modularity gains are always translated to easier reuse and maintenance. The problem is that it is usually taken for granted that the use of composition mechanisms improves the reusability and maintainability of software modules.
There is a little empirical knowledge about how these mechanisms simultaneously affect reuse and stability. This is a non-trivial problem to address due to two main reasons. First, it is not always the case that the reuse promoted by a given composition mechanism is going to lead to better maintainability. While the use of a specific mechanism can somehow contribute to modules’ reuse, it might also require developers to make various undesirable changes in their implementation. Second and more importantly, there are no metrics that enable to quantify the complexity properties of composition code. As a result, it is not possible to objectively assess to extent the intrinsic characteristics of composition mechanisms exert positive or negative influences on software reuse and maintenance.

2. PROBLEM STATEMENT AND RELATED WORK

Given the motivating context above, this PhD research focuses on analyzing the impact of advanced composition mechanisms on key properties of software reuse and maintenance. First, we plan to identify how composition mechanisms contribute to improving the reuse of modules in multiple contexts, i.e. within the same and different projects. Second, we aim at analyzing the effects of composition mechanisms on software stability [6]. A software is considered stable if its interface or implementation is not undesirably modified and ripple effects do not manifest in the presence of changes. Third, we will investigate when and which composition mechanisms make it possible to reach a better tradeoff of software reuse and stability. For this, a set of initial stability and reuse metrics were used [11]. In the following, we detail our research problems and the limitations of related work.

Characterizing Composition Code Properties. Composition code entails new dimensions of complexity in a program. Therefore, our first problem consists in revealing the properties that characterize the composition code and may exert an impact on software reuse and stability. An analysis of the motivating example (Section 1) demonstrates the need for extensive reasoning about the modules involved in the composition. A number of properties of the composition code, independently from specific mechanisms applied, need to be considered. For instance, it is required to identify which elements of the program modules are explicitly affected by the composition. In addition, 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 specification requires the preparation (e.g. refactoring) of one or more target modules. Figure 1 illustrates how these composition properties can make program comprehension quite challenging. The composition code in A1, A2 and A3 establishes not only direct dependencies among them, but also directly or indirectly with modules C1, C2, C3, C4 and many others in the system.

Revealing Reuse vs. Stability Tradeoffs. Our second problem can be described as follows: the influence of the composition code on software quality is not well understood. In particular, there is a lack of empirical studies about the impact of using composition mechanisms on software stability. A careful analysis of Figure 1 gives us some insights on how the stability of the program can be negatively impacted. As the scope of the composition embraces several modules, any change in the base code implies in either revisiting or changing the composition code. For this reason, even though composition mechanisms bring modularity and reuse improvements, each composition in the system need to be carefully analyzed and potentially modified.

Ideally, reusing software modules through the use of 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 modules that are being reused in a different context as software evolves. Otherwise, the software instabilities, provoked by these harmful modifications, are likely to degrade software design over time [5][6]. In this fashion, positive (and negative) effects of adopting contemporary programming techniques need to be systematically investigated. However, while proponents of these techniques claim that reuse and stability of software systems can be simultaneously improved [1][2][3], there is not much knowledge on whether they achieve this promise or not.

Limitations of Related Work. There is only limited assessment of advanced programming techniques and their composition
mechanisms. The problem is that all empirical studies developed so far tend to carry out a narrow analysis, focusing solely on either modularity or stability [7][11]. For instance, Sven Apel et al. [8] have contributed with an evaluation where they did not embrace multiple composition mechanisms and they did not perform any tradeoff analysis of reuse and stability. Finally, there is no investigation of how these techniques affect simultaneously reuse and stability. A tradeoff analysis of reuse and stability has been only studied in the context of conventional programming techniques [9]. For this reason, software developers cannot be informed on when and how to use composition mechanisms.

**The Dominance of Module-Driven Measurement.** The limited empirical knowledge on the effects of composition mechanisms is largely due to the lack of proper software metrics. There is no measure to characterize and quantify properties of composition code. 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. The properties quantified by these metrics vary from cohesion and coupling to method complexity [11]. However, they are all focused on measuring properties of the module structures rather than the properties of composition code itself. As a consequence, these are the only characteristics that are used in empirical studies of advanced programming techniques [10]. It is questionable whether these conventional metrics are indicators of key quality attributes of a system, such as stability and reuse.

### 3. QUESTIONS AND HYPOTHESES

Our central research question is to understand how composition mechanisms and their common properties affect software stability and reuse. This question can be further decomposed in three more specific questions:

**Q1** How to objectively analyze the impact of composition properties in software reuse and stability?

**Q2** What is the impact of composition mechanisms on the reuse and stability of a program and its modules?

**Q3** How to support developers using composition mechanisms in order to maximize reuse and stability in recurring programming scenarios?

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>Conventional modularity metrics are not able to predict the quality of composition in terms of stability and reuse.</td>
</tr>
<tr>
<td>H2</td>
<td>Reusable software modules implemented with certain composition mechanisms require less code modifications than others.</td>
</tr>
<tr>
<td>H3</td>
<td>Composition mechanisms are more appropriate to support reuse and stability in certain programming scenarios.</td>
</tr>
</tbody>
</table>

Table 1. Research Hypothesis

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Binding</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOP</td>
<td>Aspect inheritance</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Pointcut-advice</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>Intertype declaration</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Declare precedence</td>
<td>--</td>
</tr>
<tr>
<td>FOP</td>
<td>Virtual class</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Mixin composition</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>Wrappers</td>
<td>✓</td>
</tr>
<tr>
<td>CF</td>
<td>Filters</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>Superimposition</td>
<td>✓</td>
</tr>
</tbody>
</table>

EX = Extension / CO = Connection
BE = Behavior / ST = Structural

**4. METHODOLOGY AND PROGRESS**

In order to address the aforementioned research questions, a range of empirical studies will be conducted. Combination of empirical research methods, such as exploratory case studies, controlled experiments, and surveys will be used in order to enable the testing of our hypotheses (Table 1). In fact, we have already carried out some initial studies [10][14]. The composition mechanisms have been assessed in the context of a set of software product lines, where reuse and stability are naturally key drivers.

The plan is to evaluate when specific composition mechanisms in diverse programming scenarios promote simultaneous satisfaction of reuse and stability (Section 4.1). Our initial studies also served to gather some insights about: (i) the effectiveness of conventional modularity measures, and (ii) properties of composition code that can exert major effects on software reuse and stability (Section 4.2).

In our research, we have decided to focus on the composition mechanisms listed in Table 2. These are representative mechanisms supported by AOP, FOP and CF languages. For each composition mechanism, a categorization is provided taking into consideration: (i) the nature of the composition – i.e. extension (EX) or connection (CO), and (ii) the impact of the composition mechanism in the structure (ST) or behavior (BE) of the modules involved in the composition.

### 4.1 Relation between Stability and Reuse

In the first exploratory study [11], we analyzed how and when the composition mechanisms (Table 2) make it possible to reach a better tradeoff of software reuse and stability. The evaluation was carried out based on 11 releases of 2 product lines, which were originally built to promote the stable reuse of common modules across different products. Our results revealed that the combined use of FOP and AOP mechanisms seems to maximize the reuse and stability of evolving product lines in most of the cases. For instance, the hybrid use of some composition mechanisms, such as *virtual inner classes, mixin composition*, and *pointcut-advice* tended to promote product-line modules with both superior stability and reusability. We also found out that modularity properties do not seem to be the key factor to determine the degree of stable reuse of modules in the analyzed programs. We observed in both SPL cases the complexity of the composition code was the most determinant factor on the superiority (or
inferiority) of the composition mechanisms involved. This motivated us to define new metrics for quantifying composition properties, as discussed in the following.

4.2 Definition of Quantitative Indicators

Based on our first findings [11], we are working on a set of quantitative indicators for analyzing the composition code based on different programming mechanisms. Our first analysis indicates that existing modularity metrics were not correlated with the level of stability and reuse of the involved modules. Thus, a first suite of metrics is being defined to quantify attributes of composition code. The metrics were used to quantify, in a general way, some properties that are not measured by conventional module-driven metrics, including: (i) locality – quantifies the number of modules directly affected by the composition, (ii) scope – measures the set of modules directly or indirectly affected by the composition code, and (iii) scaffolding – measures the degree of preparation of the modules taking part in a composition, i.e. how many elements are introduced or modified in the source code to make a given composition possible.

4.3 Deriving Guidelines

Based on our initial studies (Section 4.1), we have started to identify some guidelines about the use of certain composition mechanisms. In general, the use of FOP composition mechanisms supported by CaesarJ, such as virtual classes and mixin composition, makes it easier to compose modules developed by different parties. However, dependencies introduced by deep inheritance trees (with promote reuse) lead to extremely coupled code, which can, in turn, hinder long-term software stability. We also found that evasive aspects [1] lead to some maintainability problems when there is a mismatch between the semantics of the base code and the representation expected by a reusable aspect. As the reusable aspect is never explicitly expected from the base program, binding it to the aspect requires adaptation in both directions. When intertype declarations belong to abstract aspects, the associated code cannot be reused in multiple concrete aspects providing alternative realizations of the introduced operations. Therefore, the abstract module needs to be changed in order to evolve the application and one alternative or optional feature needs to be changed into a mandatory one [7].

5. EXPECTED CONTRIBUTIONS AND EVALUATION

Our research work is expected to result in the following contributions: (i) empirical evidence about the improvements and drawbacks of composition mechanisms in terms of stability and reusability; (ii) a set of metrics for quantifying composition properties that will be theoretically validated [16] and also empirically validated with the support of the GQM methodology [17]; (iii) a comparative analysis of module-driven and composition-driven measures to evaluate reusability and maintainability of contemporary software systems, and (iv) a family of guidelines to support the use of composition mechanisms in recurring programming scenarios. In order to validate them, two distinct groups of programmers will be asked to use the composition mechanisms in their projects. One group will follow the guidelines and another one not.

6. REFERENCES