5

Design Models with Composition Properties

"Look back and see your way! Remember where
you came from and see where you are now."

Roberta Arcoverde
Her last words before I left Rio

Advanced Programming techniques, such as AOP (KICZALES *et al.,* 1997) and FOP (ARACIC *et al.,* 2006) are increasingly providing composition mechanisms to enable the flexible definition of module composition. They are moving away from supporting only simple composition mechanisms, such as method calls. This promotes a shift in the structure of software designs: while complexity is factored out of software modules, the composition design is much more complex (BERGMANS and AKSIT, 1992, KICZALES *et al.,* 1997, KUZNIARZ *et al.,* 2004, RAJAN and KEVIN, 2005, ARACIC *et al.,* 2006). Because of this new complexity flavor, software changes might become harder to realize (Chapter 4). Almost inevitably, software developers need to reason about the properties of module composition in order to accomplish software changes. Developers may need to analyze design artifacts to understand the composition code properties and implement the changes. Chapter 4 discussed that the module composition has a number of properties. However, these properties composition are not directly supported by modeling languages as discussed in Chapter 2. As consequence, the following question emerges: does the availability of design specification of composition code properties contribute to stability of composition-enriched programming tasks?

This question was previously answered by (DZIDEK *et al.,* 2008) in the context of object-oriented programming tasks using Unified Modeling Language (UML). According to them, the UML-visual representation of a system's declation contributes to software maintenance tasks. However, nothing is said in the literature about the impact of making available UML design models that show different kinds of composition details, which allow us to express the

nuances of their properties (*e.g.*, composition scope). Unfortunately, the reasoning about composition code properties is often hampered by the fact that most developers do not express critical composition code properties in their models. However, in the vast majority of cases, detailed models are harmful for code change (CLARKE, 2009). This happens because explicit information could actually create so much noise that the models become too complex (CLARKE, 2009, FARIAS *et al.,* 2012). Nevertheless, our assumption is that having a more detailed UML design (BRIAND, 2003), where composition code properties are more explicitly specified, will successfully support developers in maintaining their programs, thus improving the quality of the realized changes.

This chapter aims at assessing the impact of using UML design models at different kinds of composition details on the quality of program change tasks. It answers the fourth research question of this thesis (RQ4 in Section 1.3), which state: *Does the availability of richer composition modeling lead to better code stability than mainstream modeling?* This goal is achieved through the design and execution of an Internet-based experiment (Section 5.3). This experiment involved 28 participants (intermediate to senior-level) that were required to perform individually the same four maintenance tasks to an evolving game application. In order to evaluate the impact of the composition specification on the realization of programming tasks, the participants were divided in two subgroups of 14 participants each. In addition to the same source code, the first group had access to a complementary composition design in plain UML notation and the second one had access to a complementary design in an enriched UML notation, the *so-called* UML+ (Section 5.2). Section 5.1 motivates the use of design models enriched with composition code properties. Section 5.4 discusses the experiment results. Section 5.5 compares the goal of this experiment with related work. Section 5.6 identifies the threats to validity of this experiment. Finally, Section 5.7 summarizes the chapter.

5.1 Motivating Example

As explained in Chapter 4, the scope of a module composition is a property that refers to the set of modules taking part in such a composition. Let us consider a composition realized by a single method call `update(M1.setX())` in a given module `M2`. The scope of this composition is defined by three modules, namely `M1`, `M2`, and `M3`. The operation `update(M1.setX())` updates the value of the attribute x declared in module `M1`. However, let us consider that the original

value of x is used by another module `M3`. Thus, other update operations over `x` cannot be ignored by `M3` as the correct execution of these two modules depends on the computation of the correct value of `x`. Then, the scope of the aforementioned composition is not limited to `M1` and `M2`; `M3` also implicitly joins the composition, but it is not directly visible in the composition statement. The situation is likely to get more complicated if `M3` is, for instance, an aspect. If the composition scope is not well understood, it can impact negatively on the program maintenance. Then, developers might need to rely on design models, such as sequence diagrams, to understand the composition scope.

Understanding the composition scope is a key factor to manage the effect of updating the attribute `x` in `M1` over the program. Thus, the scope must be specified unambiguously in the composition design. Unfortunately, mainstream UML modeling does provide support to express the composition code properties. This way, an enriched UML notation is required to explicit the composition code properties in the composition design in the attempt of making developers generate programs more stable.

5.2

UML+ Notation

We chose UML as the design modelling language in our study due to its popularity. It is well-known and widely used in industry and academy research settings. Additionally, due to its wide adoption in software projects we did not need to provide any "artificial" training to the participants. For each programming task, class and sequence diagrams were provided. Both of them were presented at different kinds of details: UML and UML+, where the latter provides complementary information about the composition code. UML+ refers to those diagrams that make the composition code properties more explicit to developers (Chapter 4). With UML+, aspect-oriented design models are represented using the notation proposed by Chavez *et al.* (CHAVEZ and LUCENA, 2002, CHAVEZ *et al.,* 2005). Table 5.1 describes the UML+ design model representation proposed by Chavez *et al.* (CHAVEZ and LUCENA, 2002, CHAVEZ *et al.,* 2005) using the example provided in Figure 5.1.

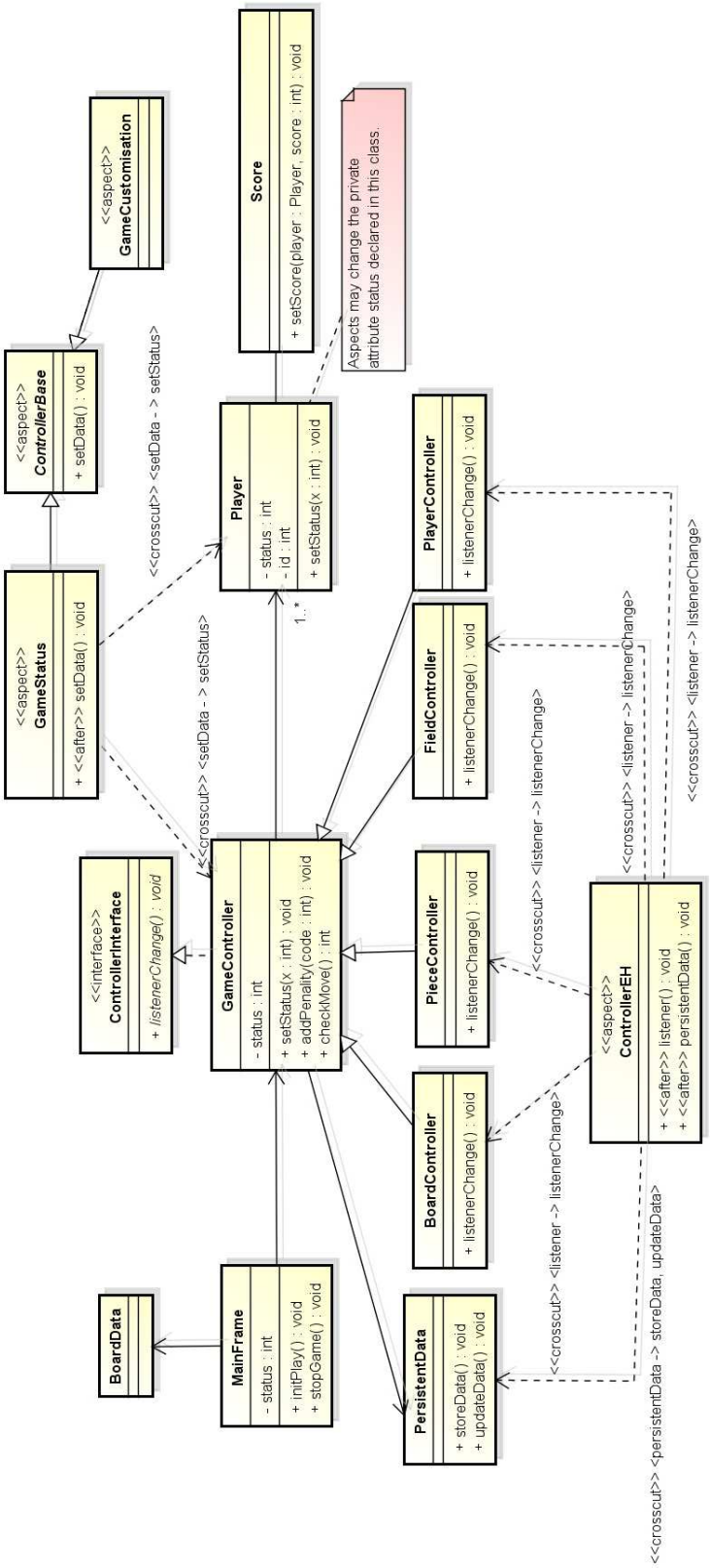

Figure 5.1: Design Model in UML+

Table 5.1: UML+ Notation

AOP Notation	Description
<<aspect>>	It defines an aspect through a special stereotype for UML classes. In Figure 5.1, examples of aspects are `GameStatus`, `GameCustomization`.
<<precede>>	It defines an order between two aspects that affect the same joinpoint. The aspect on the start of the arrow executes before the aspect touching the arrow's head.
<<crosscut>>	It defines the crosscutting relationship between the composition modules and the other modules. In this case, the aspect `GameStatus` crosscuts the classes `GameController`
<<require>>	It provides an explicit dependency between two aspects; this means that a given aspect requires the presence of another.
<<pointcutname − > operations>>	It defines the operations that are affected by an advice. In this case, we have an advice that is executed after the operation click (<setData − > setStatus>) declared in the `GameController`.

The additional composition information in UML+ models was introduced via existing UML notations, such as stereotypes and comment boxes. Moreover, UML abstractions are well aligned with the key abstractions of AspectJ. This language was chosen because it supports both (i) conventional programming mechanisms for module composition, such as inheritance, which are very directly supported by UML, and (ii) advanced composition mechanisms, such as pointcuts and intertype declaration, which are not directly represented by UML.

5.3 Experimental Design

The experiment follows the standard Between-Subject design as each group performed the experiment with exactly one treatment: UML and UML+ models. We chose this design for several reasons: (i) we aim at comparing the effectiveness of using UML and UML+ for realizing a set of programming tasks (Section 5.3.3) in the same application, (ii) as the application was the same, participants could not be part of both groups using UML and UML+ due to learning-related side effects, (iii) to reduce the threat on the use of this design, the separate groups were equally created, treated and composed of participants with equivalent expertise (see Section 5.3.4), and (iv) due to practical reasons, such as execution time restrictions and availability of suitable applications.

Figure 5.2 illustrates in detail the design in five steps: (1) we picked participants for the experiment; (2) after we assigned them to the UML and UML+ groups; (3) we treated the experiment's independent variables (Section 5.3.6); (4) we compared the experiment results in terms of stability and finally, (5) we interpreted the results statistically in order to test our hypotheses (Section 5.3.1). The main elements of our experiment design encompass: the research question and hypotheses (Section 5.3.1); the object of our experiment (Section 5.3.2); the design of the change tasks (Section 5.3.3); the participants involved in the experiment (Section 5.3.4), the experimental procedures (Section 5.3.5); and the variables and analysis (Section 5.3.6).

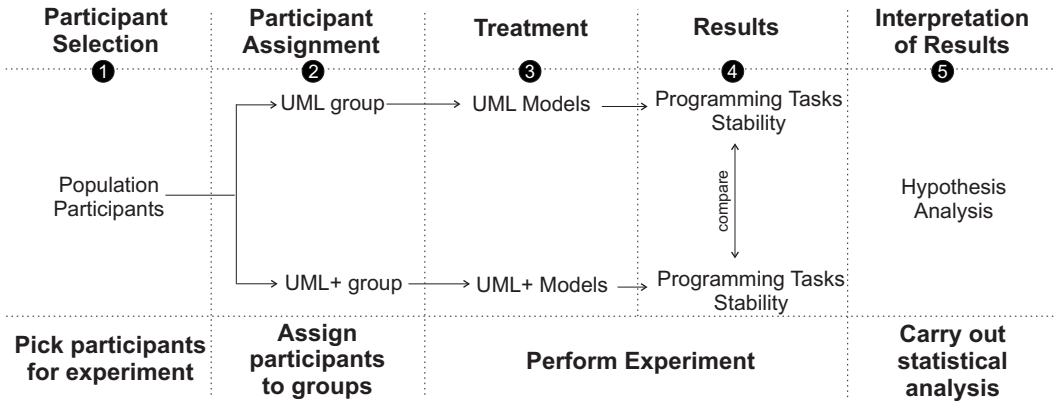

Figure 5.2: Experiment Design

5.3.1

Research Questions and Hypotheses

Our experiment aims at analyzing whether the influence of design models enriched with composition details affects the quality of changes made in software maintenance tasks. Based on that, we define the following research question:"*Does the availability of richer composition modeling lead to better code stability than mainstream modeling?*" In order to answer this question, we compare the quality of code changes made by maintainers when using either plain UML models (named UML group) or enriched UML models (named UML+ group).

The refinement of our research question consists of terms of a null hypothesis, denoted H1, which can be formulated as follows:

H1: Code change quality with UML is the same as code change quality with UML+.

The hypothesis is defined to test the impact of realizing change tasks on a program when either UML or UML+ design models are available. Through this hypothesis, it is possible to evaluate whether the variation in the "*change quality*" is influenced (or not) by the use of UML+ or UML models. The notion of change quality measured in our study is presented in Section 5.3.6.

5.3.2

Object

We selected four code change tasks for the experiment. The tasks (Section 5.3.3) mimic recurring maintenance scenarios that emerged in the project history of four evolving applications: MobileMedia (5KLOC), iBatis (110KLOC), HealthWatcher (5KLOC) and GameUP (3KLOC). The change tasks were extracted from these projects. These applications were chosen as the objects of our analysis as they were, in fact, implemented in AspectJ. More details about these applications can be found in Appendix A.

We designed the programming tasks in the experiment using only the GameUP application as it encompasses representative scenarios of maintenance programming tasks. In addition, its design follows the MVC decomposition, a well-known architecture in industry and academy (BUSCHMANN *et al.,* 1996). This means that the participants will not spend much of their time trying to become familiar with the overall design decomposition. Therefore, they can mostly concentrate their effort on the realization of the code change tasks (Section 5.3.3). Still, the design models can support them on learning the global design decisions, including the constraints governing module composition.

GameUP architecture is divided into three layers by following the MVC (model-view-controller) structure. The MVC architecture provides a low coupling between the modules present in each layer. In this way, the modules that implement GameUP interface are completely decoupled from the modules that implement GameUp controllers. Similarly, the controller modules are architecturally decoupled from data modules. Consequently, the MVC architecture facilitates the process of changing the existing source code of modules in each layer by separating the different responsibilities of the game application.

The interaction between the GameUP and the users is realized by the modules that implement the GameUP screens, such as `MainFrame` and `BoardFrame` (Appendix A). The access to the game functionalities is performed through the

controllers, such as `GameController` and `BoardController`. Data are persisted via classes that belong to the Model layer.

5.3.3 Task Design

In order to define the code change tasks, we followed three important criteria: (i) they should be representative of recurring program maintenance scenarios that require the use of composition mechanisms, (ii) they should be diverse enough to require reasoning on module composition code properties (Chapter 4), (iii) it should be doable to complete the set of tasks in the limit of one hour; and (iv) they should be representative of programming scenarios where the understanding of composition code properties is required. To this end, we defined four tasks on program maintenance. All the tasks require reasoning about the modules and their composition code properties. They also require different types of changes, such as refactorings and functionality increments (TOURWE *et al.,* 2003, FERRARI *et al.,* 2010, ENDRIKAT and HANENBERG, 2011). Therefore, these are maintenance tasks where the use of UML+ models could play a role. We also focused on this limited, albeit significant, set of tasks in order to follow the design guidelines for controlled experiments, *i.e.*, we could not demand more than one hour of work.

In addition, in order to fulfil our fourth criterion, we revisited research work, independently performed by several researchers in the community (BERGMANS and AKSIT, 1992, RAJAN and KEVIN, 2005, CAMILLERI *et al.,* 2009, KATZ and KATZ, 2009, SCHAEFER *et al.,* 2010, BURROWS *et al.,* 2010), who observed that changes made in the composition code are the cause of maintenance problems. Katz *et al.* (KATZ and KATZ, 2009), for instance, discuss the role of "invasive aspects" in software maintenance (explored in task 3). The problem of "invasive aspects", although realized through different implementation mechanisms, have their counterparts in hybrid FOP languages (PREHOFER, 1997) (*e.g.*, CaesarJ), composition filters (BERGMANS and AKSIT, 1992), and delta-oriented programming (SCHAEFER *et al.,* 2010). We had to focus on AOP in our experiment as both (i) it does not require artificial training (see Figure 5.3), and (ii) it has been used in development of industrial applications given its popularity (JBOSS, 2013, SPRING, 2013).

For each task (Table 5.2 and Appendix C), we provided both a class and a sequence diagram with the relevant design elements. These diagrams can be found in Appendix C. Regarding the tasks, we have chosen to work with open questions instead of multiple-choice ones in order to avoid resorting to simply guessing and to make them more representative under the perspective of real maintenance scenarios. Each task consisted of a request to change slices of code involving module composition. The expected answer in each task is made up by a number of required steps (operations). For each answer in a task, the participant can earn from 0 to 1 point, depending on the number of steps performed correctly (Section 5.3.5).

Table 5.2: Description of the Tasks

Task	**Description**
T1	The GameUP developers must evolve the games with a new functionality, which consists of displaying a new screen before the match starts, where the player can specify whether the game will be played on the network or not. This new functionality must be implemented through a method named `setupGame( )` using AspectJ mechanisms. The existing method `startGame()` is in charge of initializing the game.
T2	The player's status is indicated by a colored button. When the button is green (status=0) it indicates that the player can play. If the button is red (status=1) the player cannot play due to some restrictions and another match must be started. Having said this, you are required to add a new functionality which aims to change the button color to yellow (status=2) when a player suffers a penalty and passes your next turn on to your opponent. This new functionality must be implemented in a separate aspect using an after returning advice.
T3	The current version of Game UP allows both saving the board configuration in JPG format and informing the player about the result of such operation by message on the screen. The GameUP developers need your help to evolve the source code below in order to allow saving the game in XML format in addition to the JPG format.
T4	Help GameUP developers to add the method int `saveScore( )` to the class `PersistentData` using AspectJ's mechanisms. This method aims at providing a new functionality, which is to save the player's score at the end of each game match.

To successfully realize the tasks, it is required that the participants (Section 5.3.4) understand and observe the composition code properties underlying the composition code. For instance, in the UML+ class diagram of task T2 (Figure 5.1) the UML note in class Player helps (as a recall mechanism) the programmer to reason about the program elements affected by the use of after returning mechanism. This UML note highlights important information about the implicit and wide scope (Chapter 4) of after returning advice over the affected modules. An overview of the expected modifications is presented in Table 5.3.

Table 5.3: Tasks vs. Code Change

Task	Manipulating Program Elements - Necessary Modifications
T1	T1 requires at least the addition of one module and the modification of two other modules, which represent 30% of the code involved in the composition.
T2	This task requires modification in at least two modules. To be more specific this task requires the change of operations implicitly related to other modules. Approximately, 60% of the code involved in the composition is modified.
T3	This task requires the addition of one module and modification in two other ones. T3 requires modification in 50% of the modules involved in the composition.
T4	At a minimum, T4 requires the addition of one new module, the modification of another one. This task is directly associated with the expressiveness power of composition mechanisms. In this task, at least 50% of the modules are modified.

5.3.4

Participants

The Internet-based experiment involved 28 participants: 22 Ph.D. students, 4 M.Sc. students, 2 postdocs in Computer Science from different Brazilian Universities and companies. All the students had experience with software development in industry. In particular, 71.42% of the participants have worked for more than two years in the industry. We tried to diversify the participant's profiles as we understand that a group made up entirely by students might not represent developer population as stated by Di Penta *et al.* (DI PENTA *et al.,* 2007). All the participants were invited to participate without any obligation to accept the invitation. All of them have a good level of experience (at least two years) in working with AspectJ, the chosen programming language (see Figure 5.3).

The participants were asked to answer a questionnaire in order to assess their level of expertise in areas that are essential in the experiment. For instance, they were questioned about their knowledge about Java, AOP, UML and software maintenance. Using a five-point Likert scale, we quantified the participant's level of expertise per field varying from 0 (no knowledge) to 4 (expert). In particular, we required a maximum score of 2 for all fields. The result is illustrated in Figure 5.3. The participants of each group (UML or UML+) were chosen randomly. The random distribution of the participants is presented in Table 5.4 regarding the expertise level and groups.

5.3.5

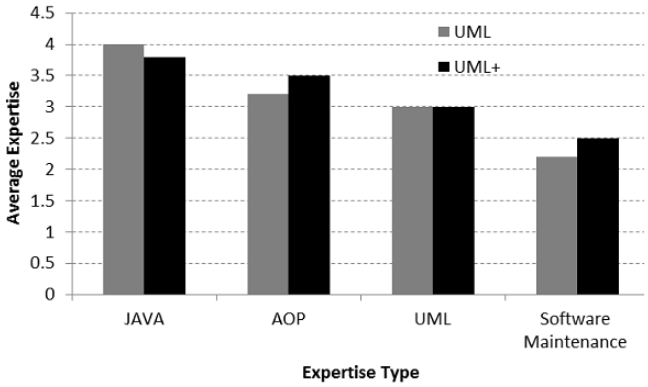

Figure 5.3: Average Expertise of the Participant Groups

Table 5.4: Participant Distribution: Expertise vs. Groups

Group	Participants Expertise Level		
	2	3	4
UML	5	5	4
UML+	6	5	3

2 = good expertise / 3 = very good expertise / 4 = expert

Procedures

The participants were required to fill in an Internet-based questionnaire about their technical skills given their different locations. Based on their answers, we designed the experiment to be performed in 60-minute Internet-based sessions. The total execution time was determined by running a pilot experiment. Each session had four scenarios on the maintenance of composition code. Considering the high level of the participant's knowledge and in order to avoid bias towards the tasks, no tutorial or training on the required technologies were needed. We divided the participants into two groups of 14 participants each: one group using UML design diagrams and the other one using UML+ design diagrams. It is important to highlight that the participants were not familiar with the experimental design goal.

For the first group, the maintenance tasks (Section 5.3.3) were given to them, along with the plain UML diagrams and the corresponding slice of source code, which refers to the existing code relevant to the task. The second group receives the same tasks given to the first group. In addition, we provide UML+ design diagrams, where information about composition code properties is explicitly represented. These models were enriched with the use of stereotypes and UML notes to explicit represent information related to composition code properties. The diagrams contained all the information about module and compositions

properties required to answer each task. The premise behind the separation of each group in two subsets aims at answering our research question (Section 5.3.1).

At the end of the experiment they were required to give their opinion on points such as time pressure and difficulty of the tasks. The participants were required to realize the experiment within 60 minutes. We decided to establish this time limit in order to prevent the participants from interrupting the experiment and restarting it later. We just estimated a time of 60 minutes in order to not get the participants tired and, thus, affect their performance. Fortunately, the adjustments in the complexity of the tasks promoted a desired effect as none of the participants reached the execution time estimate (60 minutes) to finish the experiment. The minimum and maximum execution time was 50 and 57 from UML+ participant and 51 and 55 minutes from UML participant respectively, except one UML+ participant who exceeded this time and needed 5 additional minutes. As a consequence, time was not taken into consideration in this experiment.

The experiment procedures can be summarized in three steps: (i) filling the Internet-based technical skill questionnaire (Appendix B), (ii) performing the code change tasks, and (iii) filling in the feedback questionnaire. The latter helped us to conduct a qualitative analysis and justify the quantitative differences between the UML and UML+ groups. In order to know if the participants took advantage of the provided UML-based design models, a set of questions was posed to them at the end of the experiment and also on the feedback questionnaire. These questions are summarized in Table 5.5.

Table 5.5: Summary of Quality Questions

Question (Q)	**UML**	**UML+**
Q1: Did the scope of the pointcuts affect the realization of tasks?	2/14	3/14
Q2: Did the occurrence of multiples aspects sharing the same joinpoint make the realization of tasks harder?	3/14	2/14
Q3: Did the existence of different types of modules, *i.e.*, classes and aspects affect the realization of tasks?	0/14	0/14
Q4: Did the dependency among aspects and classes, which are highly based on the program language syntax, affect the realization of tasks?	10/14	12/14
Q5: Did the occurrence of methods with similar signature pattern (*e.g.*, names starting with the prefixes "set" and "get") make the realization of tasks harder?	9/14	7/14
Q6: Did you take advantage of Class diagrams?	12/14	12/14
Q7: Did you take advantage of Sequence Diagrams?	10/14	12/14
Q8: Did you take advantage of composition code properties specifications?	-	11/14

We can summarize the procedures in four steps as illustrated in Figure 5.4: (i) the participants answered a technical skill questionnaire to capture their background and expertise; (ii) a set of tasks was designed and realized by two groups of participants: those who used UML models and those who used UML+; (iii) the answers were analyzed; (iv) at the end, a feedback questionnaire was filled in by the participants. It is important to highlight that a set of tasks was designed and realized by a small group of participants in a pilot experiment. Based on the pilot experiment feedback, we adjusted the experiment in different ways, such as number of tasks and the required total execution time.

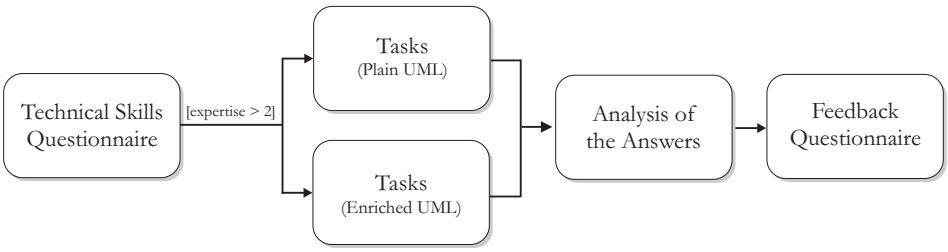

Figure 5.4: The Experimental Overview

5.3.6

Variable and Analysis

The independent variables in our experiment are the UML-based design models. Based on them we consider one dependent variable in our experiment: the change quality (or solution quality) provided in each programming task. There is more than one solution available and they are associated with varying degrees of change. The given answer is evaluated considering the number of acceptable steps to realize each task. The change quality of each answer is measured as follows. Firstly, we transform the answers into quantitative values. An entire correctly answered task is worth 1 point. Wrong answers count negatively. Thus, if the task contains 5 operations, each correct operation is worth 0.2 point. If the final score received in the task is 1 point, we classify it as correct. If it is between 0 and 1, not inclusive, we classify it as partially correct. Finally, if the final score is equal to 0 we classify it as incorrect.

The slice of code provided in Code 5.1 was given to the participants in task T1. In this task they were required to add a new method `setupGame()` using AspectJ mechanisms (see code slices provided in 5.1). In order to realize this operation, it was expected at least 3 operations: (1) add a new aspect; (2)

introduce a pointcut expression to advise the method `startGame()` (line 04 - Code 5.1) before its call, and (3) include the new aspect in the declare precedence statement within the aspect `ExecutionOrder` (lines 10 to 12). In this case each operation is worth approximately 0.33, which is the resulting value for the math operation 1 divided by 3 (total of operations). After analyzing the participants' answers, a feedback questionnaire was applied. The goal was to understand both the level of satisfaction regarding the experiment execution and the relevance of the UML+ and UML models during the tasks.

Code 5.1: Slice of Code in Task 1 using AspectJ

```
01 class MainFrame {
02   int status;
03   GameController controller;
04   void startGame(){
05       ...
06       Controller.setStatus(status);
07   }
08   void stopGame(){ //stop the game  }
09 }
```

```
10 aspect ExecutionOrder {
11   declare precedence: BoardStartup, NickNameDefinition
12 }
```


5.4 Discussion

This section presents and discusses the experimental results. First, we present the statistical test used in our analysis in Section 5.4.1. A qualitative discussion follows in Section 5.4.2. Finally, we summarize the main findings of our experiment in Section 5.4.3. The UML-based models of the tasks can be found in Appendix C.

5.4.1 Statistical Test

The statistical tests were used to evaluate the hypotheses listed in Section 5.3.1. To this end, we used the R language and environment (R-ENVIRONMENT, 2012). First, we verified whether the sample was normalized by applying the Shapiro test. After that, we applied t-test and

Mann-Whitney; this latter is used when the sample distribution was not normal. Given that the samples were not normalized, we selected the non-parametric Mann-Whitney test to analyst our hypothesis (Section 5.3.1). We used a confidence level of 95% (α=0.05). The statistical results are summarized in Table 5.6. In a nutshell, the UML+ group had 44.70% more acceptable solutions than the UML group. This means that our hypothesis can be accepted (Section 5.3.1).

Table 5.6: Descriptive Statistics of the Experimental Results

	Group	**Min**	**Max**	**Avg**	**Mean**	**Diff**	**Shapiro p-value**	**Mann-Whitney p-value**
Solution Quality	UML	0.57	4.00	2.39	2.28	44.70	0.242	0.021
	UML+	2.62	4.00	3.46	3.75		0.006	

UML+ vs. UML: Change Quality. Tables 5.7 and 5.8 summarize respectively the performance of all participants and the statistical analysis per task. The values of each task vary from zero to 1, in which case the participant had a fully-acceptable solution to a given task. Figure 5.5 illustrates the quality distribution of the 56 answers from each group (14 participants answering 4 questions). As illustrated in this figure, there is a significant advantage in favor of the group that worked with UML+ design models. For instance, none of the UML+ participants had a hit rate less than 25%. More importantly, the majority of UML+ participants provided answers with hit rate superior to 50%, but with a trend towards the range of 75-100%. The same consistent performance was not achieved by the UML group, where: (i) the hit rate tended to vary from poor (0-25%) to very good (76-100%), and (ii) most of the participants achieved a hit rate in the range of 26-50%.

In Table 5.8 we present statistical data of our analysis per task. The average values presented in the second and third columns refer to the quality of the change given by all the participants of each group (UML and UML+). The most significant difference is 59.61%, which was identified for task T2 with a p-value of 0.003. We observed that most of the participants in the UML group were not conscious about the modules affected by the new advice required by the change. In this task, a new advice with an after returning semantics was added to the existing code in order to advise join point shadows placed in the class `GameController`. Figure 5.1 illustrates this example. The advice changes the attribute status by reference and its new value is incorrectly manipulated by the class `Player`. The participant did not realize that the composition scope

also includes the class `Player`. Similarly to tasks T2, T3 and T4 are equally statistically relevant with p-values equal to 0.036 and 0.025, respectively.

Table 5.7: Participants' Performance Per Programming Task

	Task 1		**Task 2**		**Task 3**		**Task 4**	
ID(*)	UML	UML+	UML	UML+	UML	UML+	UML	UML+
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
3	1.00	1.00	0.80	1.00	1.00	1.00	1.00	1.00
4	1.00	1.00	0.40	0.75	1.00	1.00	1.00	1.00
5	0.75	1.00	0.50	0.75	0.80	1.00	0.75	1.00
6	0.75	1.00	0.60	1.00	0.75	1.00	0.66	1.00
7	0.75	0.75	0.66	1.00	0.75	1.00	1.00	1.00
8	0.50	1.00	0.40	1.00	0.50	1.00	0.32	1.00
9	0.50	0.80	0.40	0.75	0.32	0.75	0.32	0.75
10	0.32	0.66	0.66	0.75	0.32	0.75	0.50	0.75
11	0.25	0.80	0.00	0.50	0.00	0.66	0.32	0.75
12	0.32	0.66	0.40	0.75	0.32	0.75	0.50	0.80
13	0.32	0.60	0.25	0.75	0.25	0.75	0.50	0.80
14	0.25	0.50	0.25	0.66	0.25	0.66	0.25	0.80
AVG	**0.62**	**0.84**	**0.52**	**0.83**	**0.59**	**0.88**	**0.65**	**0.90**

Table 5.8: Descriptive Statistics Per Task

Task	**UML**	**UML+**	**% Diff**	**Mann-Whitney**
Task 1	0.62	0.84	35.48	0.051
Task 2	0.52	0.83	59.61	0.003
Task 3	0.59	0.88	49.15	0.036
Task 4	0.65	0.90	38.46	0.025

In these tasks, the use of stereotypes to highlight the advised operations helps the participants recall the role of each module on the composition. In task T1 the scenario was different. Our statistical test indicated that in this task the availability of UML+ models (with enhanced composition information) did not make any difference in terms of the change quality. This happens because the information associated with the composition makes explicit the required execution order between aspects. As this information could be inferred from the UML models as well, the difference was not significant. Thus, the conclusion is that the composition models bring benefits in the realization of tasks regardless of their degree of complexity. Corroborating with our answer, the participants' feedback attested this. When asked about the tasks design, participants from both groups pointed out that tasks T2 and T4 presented a higher complexity when compared to the others. This consideration strengthens our perception that the explicit composition details in design models exert a positive influence regardless of the tasks complexity.

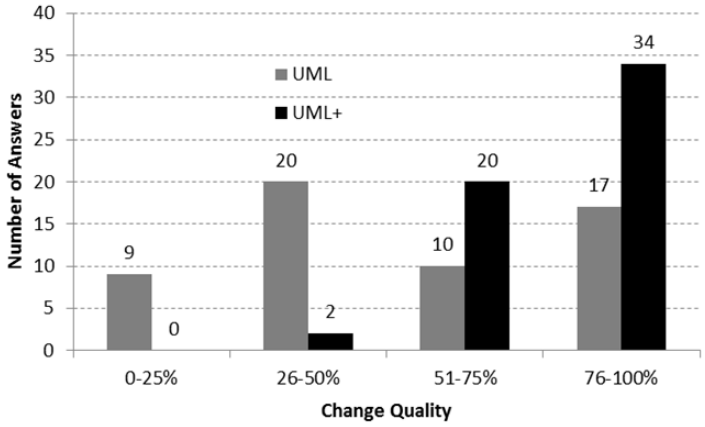

Figure 5.5: Average Percentage of Change Quality vs. Number of Answers Provided

5.4.2

Qualitative Discussion

In terms of the change quality, the highest gain with UML+ was observed in task T2. This task requires that the participant takes into consideration that there are multiple modules affected by a single pointcut-advice pair. However, this broadly-scoped impact of the composition code properties is only implicit in the code and in the UML models. The pointcut-advice pair added to the task T2 affected the value of the attribute `status` in the class `Player`. Thus, the gains reached in task T2 are associated with the composition details provided in the UML+ design models. These details make evident the *scope* associated with the use of the composition mechanism *after returning*, which was previously explained in Section 5.4.1.

UML+ group succeed in the realization of tasks. The benefits associated with the availability of UML+ design diagrams are attested by the participants. When asked "*which were the characteristics present in the source code and also in the UML-based diagrams that make the changes more difficult to be performed?*", 35.7% of the participants from the UML group answered both "*the lack of information associated with the aspects*" and "*the dependency 1:n between aspects and classes*". This claim about the lack of information was not mentioned by any participant of the UML+ group, which makes us infer that this lack of information is associated with the need of declaration of composition code properties (Chapter 4).

In addition, it should be noted that the tasks T2 and T4 were pointed out by the participants as the most difficult ones. Interestingly, these same tasks

were the ones that the UML+ group performed much better than the UML group. These results might suggest that the additional composition information available in UML+ design models became more useful as the task complexity increased. From our quantitative results we point out two main findings: (i) the use of UML+ was always favorable in terms of requiring less changes involving module composition, and (ii) the use of UML+ promotes gains in terms of the change quality regardless of participant expertise, which varies from intermediate to advanced (see Figure 5.3). We did not explicitly analyse the data with respect to the experience factor. However, we could assert that that expertise did not affect the key finding as the participants of both groups have approximately the same level of expertise and the UML+ users always yielded better results than UML participants. Finally, UML+ and UML groups consisted of participants with heterogeneous expertise, with different combinations of skills on UML, Java, AOP and maintenance (Figure 5.3).

Implicit composition code properties were detrimental to change quality. The qualitative analysis derived from both groups (*i.e.*, UML and UML+ design models) is presented in Table 5.5. The questions in this table appeared on the feedback Internet-based questionnaire (Section 5.3.5). The number of participants who answered each question positively is indicated in the second (UML group) and third (UML+ group) columns. For both groups, the majority of participants answered that overall the UML design models were useful for the realization of tasks (Q6, Q7 and Q8 - Table 5.5). When answering the question "*Which UML models were used?*", the majority of the participants (around 70%) in UML and UML+ groups confirmed they used both class and sequence diagrams. Only two participants for both groups said that UML diagrams were not useful. The participants, who did not take advantage of UML design models, explained why they have not used the models: they were very concerned on spending more time by analyzing them and, therefore, to not being able to realize all the tasks. Finally, a total of 9 and 17 participants used either the sequence diagrams or class diagrams only, respectively. Regarding the relevance of these diagrams in the execution of the tasks, the most frequent answer was "*the UML models were useful to understand how different modules interact with each other*". In this case, different modules are related to the involvement of both aspects and classes in a composition, which is associated with the composition diversity (Chapter 4).

Finally, most participants indicated that the two main factors that led them to change the existing code were: (i) the dependencies among classes and aspects are highly based on the program language syntax, and (ii) similar pattern of

method signatures (see Table 5.5 - Q4 e Q5). These answers indicate that both UML and UML+ groups were aware of changes violating existing composition code properties. However, participants who used UML+ in their tasks also pointed out the scope of the composition (Q1) as a detrimental factor to generate a solution with better quality. This means that even the participants do not have a deep understanding about the effect of these composition code properties in evolving programs, they were able to identify them as detrimental factors to provide acceptable solution to the program maintenance tasks.

5.4.3

Solution Quality meets Stability

The UML+ design models were consistently beneficial in terms of the change quality. However, we observed the following phenomenon in the performance of both groups (UML and UML+): the more changes are required to realize a programming task (Section 5.3.3), the more prone are the participants to introduce mistakes in their answers. Figure 5.6 illustrates the existing correlation between the performance of the participants in terms of number of changes (change quality) and the percentage of changes required to realize the programming tasks. It is important to highlight that there is not only one way in which each task can be realized to obtain a solution. For this reason, the number of changed program elements can vary in each task.

However, taking Figure 5.5 into consideration, we can observe that the percentage of changes was lower in the UML+ group when compared to the UML group. For instance, when UML+ participants were required to add the method `int saveScore( )` to the class `PersistentData` using AspectJ's mechanisms, some of them decided to add as well a new pointcut in the aspect `BoardTracing` instead of modifying the both the existing pointcut (line 08 - Code 5.2) and advice (lines 10 to 12 - Code 5.2). On the other hand, UML participants realized the same tasks changing the pointcut expression and also the advice of the aspect `Boardtracing`. In particular, this means that the participants realized the programming task in a way that the existing code is changed more frequently when compared to the UML+ group.

Code 5.2: Slice of Code from Task T4

```
01 class PersistentData {
02     ...
03     int saveJPG( ) {...}
04     int saveXML( ) {...}
05     ...
06 }
```

```
07 aspect BoardTracing{
08     pointcut traceSaveBoard():call(int PersistentData.save*( ));
09     ...
10     after( ): traceSaveBoard (){
11         System.out.println("Board Saved");
12     }
13     ...
14 }
```

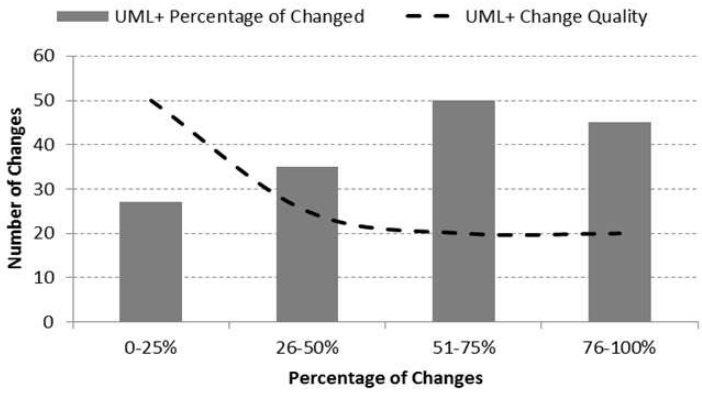

Figure 5.6: Percentage of Changes realized by UML+ Group

5.5 Related Work

Related work can be divided into two categories: the impact of UML models on software maintenance (Section 5.5.1), and quality analysis of programs built with advanced programming techniques (Section 5.5.2). None of related work discussed below analysed the impact of UML models designed with explicit composition code properties. We provide an overview of these works and highlight the main differences when compared to ours.

5.5.1

The Impact of UML Models

Previous research work explored the impact of 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). All of them claim that the availability of UML design models provides developers with a more effective representation of several system properties. In their experimental investigation they evaluate whether using UML is profitable in a realistic context for a large project system. In particular, according to them, all of the participants found the UML diagrams useful in terms of maintenance. In addition, they also concluded that the use of UML models establishes a better communication among software developers. Nevertheless, all these approaches are different from ours because they have only focused on evaluating the general benefits of UML models in object-oriented systems. There is no evidence on what extent the use of UML models with proper composition declation improves the maintenance of software systems when advanced programming techniques are used. For this reason, our work is a first attempt to empirically investigate the impact on software maintenance when developers use UML models enriched with composition code properties.

5.5.2

Analysis of Advanced Mechanisms

Several works have analysed the use of AOP on software maintenance and development (KASTNER *et al.,* 2007, GREENWOOD *et al.,* 2007, FIGUEIREDO *et al.,* 2008a, ALVES *et al.,* 2006, BURROWS *et al.,* 2011, ENDRIKAT and HANENBERG, 2011). For instance, Figueiredo *et al.* (FIGUEIREDO *et al.,* 2008a) investigated the impact of AspectJ mechanisms on software evolution. The authors focused on the source code analysis based on a suite of modularity metrics. However, they do not discuss at all in terms of the use of modeling techniques for making the composition code properties explicit. Other works have evaluated the use of AspectJ on refactoring software systems (*e.g.*, (ALVES *et al.,* 2006, KASTNER *et al.,* 2007)). There are other works that have investigated specific attributes; *i.e.*, how faults are introduced during maintenance tasks involving aspects (BURROWS *et al.,* 2011) and the time spent by developers to evolve and manage the complexity of composition code (BASILI *et al.,* 1999, ENDRIKAT and HANENBERG, 2011). However,

similar to Figueiredo *et al.* (FIGUEIREDO *et al.,* 2008a), all these works only investigated the impact of AspectJ mechanisms on the quality of the source code without taking into consideration the impact of composition code properties on the software maintenance. Also, they do not provide any evaluation in terms of the impact of using modeling techniques on software maintenance. Thus, our work is different from the aforementioned ones as we are interested in analyzing how the UML models with composition code properties are effective to assist developers on software maintenance tasks.

5.6 Threats to Validity

Some relevant threats to validity and the manners in which we have addressed them are discussed as follows (WOHLIN *et al.,* 2000).

Internal Validity. Threats to internal validity reside on the participant's expertise, participant's assignment bias, task bias and time restriction. Regardless of the fact that the participants may not be sufficiently expert, we have reduced this threat through a priori application of a technical skills questionnaire. It intends to assess the participants' expertise degree on dealing with the technologies involved in the study. According to the results of this questionnaire, we selected the participants with score equal or superior to 2 for all fields, which means a similar and high knowledge level. These procedures were important given the between-subject nature of the experiment design. Another question concerns the fair distribution of the participant's knowledge. In fact, they were randomly distributed to each group. However, this threat was minimized since all of them have a high level of knowledge (score >=3) about the involved technologies (see Figure 5.3 and Table 5.4). Another prominent criticism is about the motivation of the participants and also their knowledge about the experimental goal. These criticisms are mitigated by the fact that all participants participated on a voluntary basis and although they could guess, they were not familiar with the research questions and hypothesis (Section 5.3.1). Another threat is associated with the assignment bias as we have a Between-Subject experiment. This threat was reduced as we selected participants with good and similar expertise. A final threat relies on the time to perform the tasks and their degree of difficulty. We attempted to mitigate this threat performing one pilot study with four extra participants who had the same level of knowledge required to attend the experiment. Based on their performance it was possible to adjust the level of difficulty and the required

time to perform each task. With respect to the composition specification, it is true that the use of a detailed composition specification (UML+) could make the program maintenance hard to perform. In fact, we tried to select design slices that had many non-trivial characteristics of aspect-oriented designs (aspect inheritance, multiple aspects with global quantification - *i.e.*, resulting in many crosscut relationships). On the other hand, we also considered that some good modeling practices were applied. In addition, we are considering the context here where the programmer is supported by a tool to navigate through the models and code, and it is, therefore, focusing only on the modules involved in maintenance task.

External Validity. External validity involves the extent to which the results of a study can be applied. In order to minimize this threat, we chose developers with heterogeneous backgrounds who are, at most, professional developers and postgraduate students as well (HOLT *et al.,* 1987, HÖst *et al.,* 2000, ARISJOLM and SJOBERG *et al.,* 2004). In addition, the evolving tasks by themselves can be considered a threat as they can be considered far from real evolving software scenarios. We tried to neutralize this threat with the choice of the application. When we decided to work with applications from well-known domains such as mobile devices and games, we took into consideration the facility of simulating real scenarios. This means that the functionalities discussed in our experiment mitigate the real functionality of cell phones and card games. In addition, as we are using programming scenarios from different subject applications (Section 5.3.2), we tend to generate more reliable results.

Generalization of results also depends on the scale of the tasks considered in the experiment. For context of this experiment design, the changes are, in fact, not expected to affect too many modules. On the contrary, the context of the tasks was assumed to one where developers were concerned with modularity principles; this is also why advanced programming techniques were applied in the first place. A narrow scope of changes was also adopted for the experiment tasks as the target program was representative of those where good programming/design practices are adopted (*e.g.*, the MVC architecture style and other design patterns in the case of the application used in the experiment). Hence, the maintenance tasks indeed did not require a high number of changes. We also tried to reduce the threat associated with this point mimicking maintenance scenarios of real applications (Section 5.3.2).

Construct Validity. The threat to construct validity includes the methodology used both for quantifying changes and reusability degree. In order to reduce this threat, we used a set of composition metrics that allowed us to

predict changes and reusability from each task point of view. We adopted these metrics because all of them were empirically found to have correlation with design change. In addition, they enabled us to make a more objective comparison with outcomes of relevant previous studies (FIGUEIREDO *et al.,* 2008a) (Chapter 2). Another factor that contributes to neutralize is the data collection process. In this process, we consider partially correct answers following the criteria presented in Section 5.3.5.

5.7 Summary

A considerable part of software design is dedicated for the composition of two or more modules. The implication is that changes made later in the implementation often require some reasoning about module composition code properties. However, these properties are often not explicitly specified in design artifacts. Moreover, they cannot be easily inferred from the source code either. As a result, implicit composition code properties may represent a major source of software maintenance complexity. This fact is particularly true with the advent of advanced programming techniques, which are increasingly providing advanced mechanisms to enable flexible module composition. However, there is little empirical knowledge on how design models with explicitly-specified composition code properties can improve software maintenance tasks.

An Internet-based experiment was carried out in order to investigate the contributions associated with the availability of composition-enriched (UML+) design models during maintenance programming tasks (Section 5.3). The UML models were used to support the program change tasks and they had a different degree of module composition information. Half of the participants realized the programming tasks using plain UML models. The second half realized the same programming tasks using UML+ diagrams (Section 5.3.5).

Our results showed that, regardless of participants' expertise, developers who had an UML+ model outperformed the other case in terms of leading to better change quality. In addition, a complementary analysis highlighted the correlation between the change quality and the number of required changes to perform the tasks. Users of composition-enhanced models consistently yielded better results when compared to users of plain UML models. The use of explicit composition specification also led to an average increase of 44.7% in the quality of the change produced by the experiment participants.