2
Background and Related Work

Empirical studies are essential to evaluate the composition effort of design models in practice. These studies allow building a body of knowledge supported by empirical evidence, testing out hypotheses, identifying important context variables, and understanding how influential factors may affect developers' effort when composing models. Without these studies, it is not possible to realize effective improvements for the current state of the art of model composition.

The goal of this Chapter is to provide an overview of the main concepts and definitions required understanding the empirical studies of model composition presented in this thesis. This chapter also describes the relevant elements underpinning the three model composition factors investigated in this thesis. Finally, it also provides an overview of the limitations of related work considering the topics addressed in our research questions (Section 1.3).

The remainder of this chapter is organized as follows. To begin with, Section 2.1 presents the purpose of using model composition in practice. After that, the main characteristics of the design modeling languages are presented (Section 2.2) and the purpose of using design models is also discussed (Section 2.3). Then, the elements of the three influential factors are explained in the next sections. Section 2.4 describes the types of composition techniques. Section 2.5 presents the modeling languages used to represent design decompositions. Section 2.6 elaborates on the design characteristics studied, more specifically those related to model stability. In all previous three sections, the related works are discussed and contrasted.

2.1. Purpose of Using Model Composition

Model composition is a fundamental activity that addresses the limitations of humans for simultaneously dealing with a plurality of artefacts and tasks (Mistrík et al., 2010; Whitehead, 2007). Dijkstra advocates to master complexity

someone should deal with one important issue at a time (Dijkstra, 1976). With this in mind, software developer tends to work on simple tasks rather than on complex tasks; but each task manipulating small artefacts rather than big, complex ones. For example, developers work on small parts of an overall design model in order to focus on part of the model relevant to them. Unfortunately, they are unable to create a "big picture" view from the small parts created in parallel by different software development teams. The composition of the parts can be performed by using a model composition technique. Many academic and industrial composition techniques (Section 2.4) have been proposed to help developers to use model composition for different purposes.

In this thesis, we investigate the composition effort in the context of the evolution of design models. We identify three particular purposes of using model composition, which are presented based on the degree of relevance for the study. They are described below:

1. *Change of design models.* Developers use model composition to systematically change design models in collaborative development environment. Typically, they add, modify, remove, or even refine model elements of some existing design model in parallel. By using a more systematic way of bringing together changes, developers aim at implementing the changes rather than concerning on integrating the parts of even grasping the impact of the changes. Consequently, this absence of concerns on composing the models helps developers to effectively change the models.
2. *Reconciliation of design models.* Usually developers create design models in parallel and parts of these models conflict with each other. Thus, the model composition techniques can identify these contradicting parts and help developers to reconcile them. In (Clarke, 2001), Clarke defines a mechanism for identifying and reconciling these conflicts. This mechanism provides guidance to developers explaining how reconciling contradicting models.
3. *Analysis of overlapping parts.* Design models are realized in multiple ways, and hence at some point developers must converge on a single one. As humans, developers are unable to recall all myriad of changes performed during the composition time (Whitehead, 2007). Hence, they cannot foresee

when the changes are going to overlap. Therefore, the composition technique helps developers to identify the overlapping parts. This identification is critical because developers must decide which part will remain into the output composed model.

Regardless of the usage scenario, developers are always concerned with making the use of the composition technique to correctly produce the output composed model. The composition techniques studied in this thesis are explained in Section 2.4.

2.2. Properties of the Design Modeling Languages

Popular modeling languages, such as the UML (OMG, 2011), have particular properties and different diagrams that can play a role on model composition effort. Some relevant properties are described as follows.

Lack of a rigorous definition. The design modeling languages are defined by a metamodel, which specifies the syntax and semantics of the language' constructs. This specification is aided by a set of well-formedness rules that enable a more precise definition of the constructs. These rules can be expressed by using OCL (OMG, 2011), for example. Unfortunately, these rules are seldom represented in a formal way (Larman, 2004; OMG, 2011). Rather, they are usually expressed using natural language. If well-formedness rules are not formally specified, then they can jeopardize the benefits of using of model composition (Section 2.1). For example, if a composition incorrectly reports a high number of conflicts, then developers will invest some unnecessary effort to deal with them. A high amount of conflicts makes the composition unmanageable (Mens, 2002), increasing the likelihood of inconsistencies in the output composed model. Incorrect composed models jeopardize the communication between the developers, as misinterpretation may become inherent (Broy & Cengarle, 2011; Maoz et al., 2011a; Maoz et al., 2011b; Lange & Chaudron, 2004). If the syntax and semantics are formally specified, the conflicts and inconsistencies are reduced or even localized more quickly. Therefore, given the state of practice on software modeling, this thesis attempts to investigate model composition effort when rigorous definition is not available. We study the identification of conflicts and

inconsistencies in scenarios where developers need to deal with the lack of formal information. All the studies follow this strategy (Chapter 4, Chapter 5, and Chapter 6).

Multi-view design modeling languages. The design modeling languages also define a range of structural and behavioral diagrams to represent static and dynamic aspects of software systems. The elements of complementary diagrams (e.g., UML class and sequence diagrams) should have a precise consistency with each other; otherwise, conflicting information in different views of the same system may lead to misinterpretations. For example, an abstract class in a class diagram cannot be used in a sequence diagram, as abstract classes cannot be instantiated. Otherwise, developers may not observe the inconsistency and make different interpretations about this class. Some of them may infer that the class is concrete, while others will infer that the same class is abstract. The rate of conflicting information typically increases when developers evolve design models in parallel or even when the synchronization of design models is not fully realized. Different developers tend to assign values to the model's properties that are conflicting. This thesis attempts to investigate how this lack of agreement between the models leads to problems during the composition. Essentially, we are concerned on understanding how these multi-view inconsistencies influence the effort of composing design models and how developers deal with such inconsistencies in practice.

Complexity of the design modeling languages. The size and complexity of the design models have grown in recent years (Lange, 2007b) as developers are increasingly creating systems that are more complex. To deal with these problems, the design modeling languages have also grown and delivered new constructs. For example, the UML and its extensions provide 13 diagram types, totaling more than 150 constructs (Dori, 2002). However, the high number of diagrams and constructs has led the language to become more complex than it was originally planned. If design models are complicated, then their compositions can also tend to be more complicated. Consequently, developers tend to modularize the design models in such a way that the size and complexity of the design models can be minimized. For example, developers may use object-oriented or aspect-oriented modeling in order to better modularize design models. This thesis attempts to understand how the use of different modeling languages can minimize the

complexity of the design models; hence, reducing the composition effort (Chapter 4, Chapter 5, and Chapter 6). For example, we are concerned with knowing how different forms of decomposing designs can influence the composition of such models.

Therefore, this thesis studies model composition effort in the presence of imprecise model semantics as well as non-trivial, multi-view design models.

2.3. Purpose of Using Design Models

Many modeling languages have been proposed in recent years, such as the UML (OMG, 2011) and its extensions (Clarke & Banaissad, 2005; Baniassad & Clarke, 2004). These languages provide a set of modeling resources to developers so that they can represent concepts and their relationships. According to (Rumbaugh et al., 1999), the representations created by using these resources are abstractions in essence from a reality observed and reported at a specific level of detail. Developers can use these modeling resources in a range of situations such as specifying software architectures, communicating design decisions, and documenting software systems. In this thesis, we use UML class diagrams and UML component diagrams, and their respective extensions in aspect-oriented modeling. These two modeling languages (and diagrams) were chosen because some reasons.

First, UML is de fact the standard design modeling language adopted by researchers and professionals in practice. The UML class and sequence diagrams are the most used diagrams (Dobing & Jeffrey, 2006). Second, most modeling tools are dedicated to create and manage UML models and its extensions such as IBM Rational Software Architect (IBM, 2011). Third, the AO modeling is the state-of-the-art modeling language for the modularization of software systems (Clarke & Walker, 2005; Clarke & Banaissad, 2005). Fourth, the UML is a general-purpose modeling language for systems engineering applications. It supports the specification, analysis, and design of a broad range of systems (OMG, 2011). Fifth, as the UML is the basis of most modeling languages today, the results can be possibly transferable to other modeling languages based on it. Sixth, both languages define notations to allow developers to graphically represent

static and dynamic views of a software system. These notations are available in thirteen diagram types described in (OMG, 2011; Clarke & Walker, 2005). The UML and AO models were used for three proposes during the empirical studies:

1. *Communication.* Developers use design models to communicate design decisions between teamwork members.

2. *Comprehension*. Developers use design models to comprehend the modules of a software system before implementing them.

3. *Documentation for maintenance*. The UML's diagrams are used during maintenance to locate system elements that are affected by a maintenance request.

Additionally, design models can be also used for other purposes such as code generation (Schmidt, 2006), effort estimation (Mohagheghi et al., 2005; Uemura et al., 1999), quality prediction (Genero et al., 2003; Cortellessa et al., 2002), and testing (Briand & Labiche, 2002). However, we do not use models for these specific purposes during the empirical studies. In the next section, we present the model composition techniques investigated in this thesis.

2.4. Model Composition Techniques

Academia and industry have proposed many model composition techniques in recent years. These techniques differ in their manner of expressing the compositions. While some of them require the explicit specification of how the compositions should be carried out, others rely on composition heuristics to "guess" how the elements of the input models will be composed. Therefore, the techniques can be grouped into two broad categories as follows:

- *Specification-based technique.* This category brings together the techniques with which developers express the compositions by explicitly determining the manner how the input model elements will be matched and composed. Two state-of-the-art examples of this category are the MATA (Whittle et al., 2009) and Epsilon (Epsilon, 2011) techniques.
- *Heuristic-based techniques.* Techniques in this category are characterized by a set of predefined composition heuristics, which are responsible for "guessing" the correspondence between the input model elements. Based

on such guessed similarities, the techniques can then combine the input model elements. Two examples of the heuristic-based techniques are the IBM RSA (IBM, 2011) and conventional composition algorithms of model elements, including merge, union, and override (Clarke & Walker, 2005).

The specification-based technique used in our study was the Epsilon technique (Kolovos et al., 2011), and the heuristic-based techniques were the one supported by the IBM RSA tool (IBM, 2011) and traditional composition algorithms (Clarke, 2001; Clarke & Walker, 2001). They are explained in the next sections. Figure 1 shows an illustrative example that will be used to support the discussion of the studied composition techniques.

2.4.1. Traditional Composition Algorithms

We have studied three manual, heuristic-based composition algorithms: *override*, *merge*, and *union*. These algorithms were proposed and analyzed in (Clarke & Walker, 2005). There are some reasons that motivated the use of these algorithms in this study. First, evolution scenarios can be decomposed into one (or more) canonical operation supported by these algorithms. Typically, these operations are *additions*, *modifications*, and *removals* (Section 3.3).

Second, these algorithms can be also seen as basic "rules of the thumb" for developers to compose models; they do not need to be strictly realized for each instance of model composition in a software project. They provide general descriptions of how the compositions should be performed and guide developers to combine model elements. For example, these general composition guidelines may be useful to accommodate the specificities of particular model compositions and lead to fewer inconsistencies in the output composed model.

Third, they have been applied in a wide range of model composition scenarios, such as evolution and integration of software product lines (Jayaraman et al., 2007), and composition of design models (Clarke & Baniassad, 2005), and aspect-oriented modeling (Clarke & Baniassad, 2005). They have been recognized as candidate algorithms to compose well-modularized design models, such as aspect-oriented design models, e.g., Theme/UML (Clarke & Baniassad, 2005).

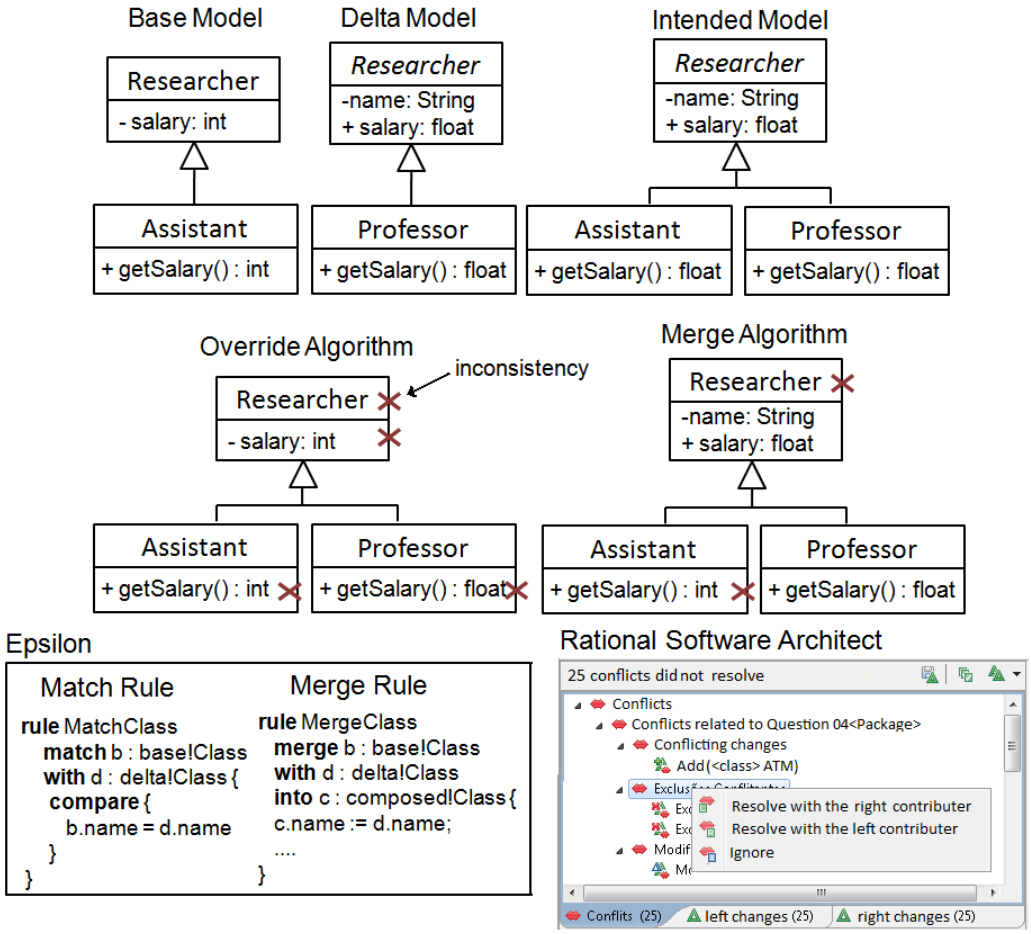

Figure 1: Illustrative example

In the following, we briefly define override, merge, and union algorithms, using a simple example to illustrate them. We assume the presence of two input model, M_A and M_B. We consider that two elements from M_A and M_B are corresponding if they have been identified as equivalent in the matching process.

Override (direction: M_A to M_B). For all pairs of corresponding elements in the base model (M_A) should override its similar element in the delta model (M_B). Elements not involved in the correspondence remain unchanged. They are then inserted into the output model. Figure 1 shows the application of this algorithm. The concrete class *Researcher* (*isAbstract* = false) overrides the abstract class *Researcher* (*isAbstract* = true), and the concrete classes *Assistant* and *Professor* were just inserted into the output composed model. However, the intended model was not produced. Rather, the output composed model has three inconsistencies. This implies that the algorithm was not able to properly accommodate the changes

from the delta into the base model, as would be expected. Note that the algorithm was applied in the direction from the base model to the delta model.

Merge. For all corresponding elements in M_A and M_B, such elements should be composed instead of overridden as in the override algorithm. The composition depends on the element type. Elements in M_A and M_B that are not involved in a correspondence match remain unchanged and, consequently, are inserted into the output model directly. In Figure 1, the merge algorithm is applied from the base model to the delta model; hence, a composed model is produced with two inconsistencies. Again, the intended model is not produced. Although the attribute *Researcher.name* has been correctly inserted into the class *Researcher*, it is a concrete class (*isAbstract* = false) instead of abstract (*isAbstract* = true), as it would be expected (according to the intended model). This problem affects the method *Assistant.getSalary():int* as a ripple effect. To produce the intended model, the merge algorithm should be applied from the delta model to the base model. Given this inverse order on the application of the algorithm, the changes in the delta model will predominate over the model elements in the base model.

Union. For all elements in the base and delta model that are corresponding elements, they should be manipulated in order to preserve their distinguished identification. It means that they should coexist in the output models with different identifiers; elements in the M_A and M_B that are not involved in a correspondence match remain unchanged, and they are inserted into the output model, M_{AB}. For example, we will have two classes *Researcher* in the composed model. However, both classes will carry identifiers that somehow preserve their original identities e.g., *BaseModel.Reseacher* and *DeltaModel.Researcher*.

2.4.2. IBM Rational Software Architect

IBM RSA is a comprehensive modeling and development environment that relies on the UML language artefacts (Norris & Letkeman, 2011). We choose IBM RSA due to some reasons.

First, it is the most robust composition techniques adopted in industry (Norris & Letkeman, 2011). In (Altmanninger et al., 2009), this superior quality is supported by empirical studies. Second, IBM RSA's model validation mechanism

allows us to the automated identification of syntactic inconsistencies. This means that developers are expected to localize inconsistencies more quickly than manually, minimizing the detection effort. Third, it provides an adequate composition environment to report the conflicting between the input model elements.

Fourth, it allows creating all thirteen UML diagrams and executing some important operations such as model transformation and reverse engineering. In particular, it supports model-to-code (e.g., UML to Java) and code-to-model (e.g., Java to UML) transformations. In addition, it supports reverse transformations go from Java to UML, C++ to UML, and .NET to UML. IBM RSA is designed on top of the open-source Eclipse development platform. Therefore, it gives the developers a complete IDE for model-driven software development. In addition, it provides a disciplined control of shared design models in evolving software projects. Finally, empirical studies (Altmanninger et al., 2009) indicate that IBM RSA's composition technique has a considerable level of precision compared with other related technologies such as Subversion (SVN, 2012), EMF compare (EMF, 2012), and UNICASE (Unicase, 2012). More importantly, it enables model management in collaborative software development e.g., splitting, comparing and composing models created in cooperation.

Although IBM RSA implements a robust and precise model composition technique, it does not ensure that the intended model will be always produced. This means that developers should necessarily interact with models via the tool facilities to produce an output composed model. Figure 1 depicts an example of conflict report produced by RSA. For example, when conflicting changes emerge, developers should decide which changes — from the base model (*Researcher.isAbstract* = false) or from the delta model (*Researcher.isAbstract* = true) — will be inserted into the output composed model.

2.4.3. Epsilon

Epsilon is a *flexible* platform for model management (Kolovos et al., 2011) defined as an Eclipse Plug-in. This flexibility is achieved by providing a set of consistent task-specific languages for developers so that they can perform some

tasks such as model comparison and model composition. To date, seven interoperable, but with different purposes, languages have been proposed to help developers to manage design models. Although there is a wide diversity of modeling languages, we put our attention on two specific languages: the epsilon comparison language (ECL, 2012) and the epsilon merge language (EML, 2012). They are two hybrid, rule-based languages used to compare and merge design models, respectively (EML, 2012). These two languages were chosen because three reasons.

First, they are responsible for executing the two most common tasks in model composition: comparison and composition of models. Second, these languages define a set of constructs expressive enough to seamlessly specify how the input model elements are going to be compared and integrated. Third, by using these languages, developers can master the complexity of dealing with inherent composition problems, i.e., the imprecise specifications of commonalities and differences between the input model elements. Lastly, they are intuitive and expressive enough so that we empirically investigate the effort of developers invest to compose two design models (Kolovos et al., 2011).

Additionally, the Epsilon platform also presents some interesting characteristics to support the use of those two languages. To begin with, the feature of syntax highlighting differs in colors and fonts the language constructs improving the readability of the composition specifications. Next, the code completion steeps the learning curve, i.e., the learning related to composition specification may be achieved more quickly. This resource can improve the quality of the composition specification by decreasing the initial difficulty of creating and editing the composition specifications. Developers can become more familiar with the languages; hence, improving the definition of the correspondence and composition relations. Thirdly, the syntax highlighting and code completion are two crucial characteristics, for example, to foster the use of model composition by novices. To sum up, the Epsilon is an Eclipse-based IDE provides important resources to developers, so that the comparison and composition rules can be carefully created and edited. Figure 1 shows an example of these rules. The *MatchRule* determines that there can be correspondence relations between the input classes if their names are similar. The *MergeRule*

specifies that the name of the output composed classes should be equal to the name of the input class of the delta model, i.e., *c.name* := *d.name*.

To sum up, these three techniques (i.e., Epsilon, IBM RSA and Traditional Algorithms) are good candidates for comparisons because: (1) they are robust and usable tools, which are two prerequisites for an experiment like this; (2) IBM RSA is an industry leading model composition tool; and (3) traditional algorithms such as merge/override are well mentioned in the academic literature as a technique and have been used to build tools.

2.4.4. Limitations of Related Work on Model Composition Techniques

Model composition is a very active research field in many research areas, such as merging of state charts (Whittle & Jayaraman, 2010), composition of software product lines (Clarke, 2001), aspect-oriented modeling (Clarke & Walker, 2005), and mainly composition of UML design models (Farias et al., 2011a). In doing so, there has been more research on proposing model composition techniques or even creating innovative model composition techniques, such as traditional composition algorithms (Clarke, 2001; Clarke et al., 2005), IBM RSA (IBM RSA, 2011), Epsilon (Kolovos et al., 2011), MATA (Whittle & Jayaraman, 2011), Kompose (Kompose, 2011) rather than evaluating them.

Clarke and colleagues (Clarke, 2001; Clarke et al., 2005) propose three conventional algorithms, namely override, merge, and union, to compose UML design models such as UML class diagrams. These algorithms are the basis for other composition techniques such as Epsilon (Kolovos et al., 2011), Araxis Merge (Araxis, 2011), KDiff3 (KDiff3, 2011), and MergePlant (MergePlant, 2011). Araxis Merge is a 2/3-way file comparison, merging and folder synchronization for Windows and Mac OS X. The focus of the techniques is on synthesizing text-like files rather than design models (Araxis Merge, 2011). KDiff3 (KDiff3, 2011), MergePlant (MergePlant, 2011). They are useful for determining what has changed between versions, and then merging changes between versions.

Kolovos and colleagues (Kolovos et al., 2011) propose the Epsilon Platform in order to compose homogenous and heterogeneous design models. That is, the tool is able to combine input design models that are instanced from a particular

metamodel or from different metamodels. Epsilon offers an innovative, flexible platform to promote compositions of design models.

However, none of these approaches has investigated the effort that developers should invest to compose design models. As a matter of fact, the current literature in composition techniques points out the absence of empirical studies and does highlight the importance of empirical evidence (Dingel et al., 2008; Apel et al., 2011; Uhl, 2006; Mens, 2006; France & Rumpe, 2007). This absence of knowledge may cause serious consequences. First, it is not possible to grasp if the effort invested by developers is cost-effective (or not). Cost-benefits analysis in terms of effort is crucial before applying any technique in practice. If the effort of applying a particular technique is high, then developers will not use in practice. Second, the composition techniques are improperly used due to the influential factors that directly (or indirectly) affect the use of the techniques are unknown.

The current works have notably aimed at evaluating the use of design models rather than the consequences of the application of composition techniques on them. In fact, there existing studies concentrate on investigating UML models in terms of quality attributes such as comprehensibility (Ricca et al., 2010) and completeness (Langes & Chaudron, 2004). These works are very important, as the current standard modeling language is the UML.

In addition, we have also observed that most of the research on the interplay of effort and composition techniques rests on subjective assessment criteria (France & Rumpe, 2007). Even worse, they depend on the expert judgments, who have built up an arsenal of mentally held indicators to analyze the growing complexity of models and then evaluate the effort on composing them. Therefore, to date, developers rely on feedback from experts to determine “how good” the input models and their compositions are.

According to (France & Rumpe, 2007), the state of the practice in assessing model quality provides evidence that modeling is still in the craftsmanship era and when we assess model composition the problem be aggravated. More specifically, to the best of our knowledge, our results are the first to empirically investigate the research questions in a controlled way by using specification-based and heuristic-based techniques.

To sum up, there are two critical gaps in the literature. First, practical knowledge about the relative effort of composing design models is lacking. That is, developers do not know very little about what they invest in terms of effort to apply the composition techniques as well as detecting and resolving inconsistencies. Second, insight about the potential influential factors is also lacking. Hence, developers are unable to improve the composition process (i.e., the execution of the composition activities) once they do not know which, in fact, jeopardize the execution of the activities. Second, the lack of empirical evidence about the correctness of the output models produced using these techniques in practice.

2.5. Design Modeling Languages

In this research, we focus our investigations on the Unified Modeling Language (UML) (OMG, 2011) and one of its extensions to Aspect-Oriented Modeling (AOM) (Clarke & Walker, 2005).

2.5.1. Unified Modeling Language

The Unified Modeling Language (UML) is a general-purpose modeling language adopted as the standard modeling language in practice (OMG, 2011). The UML models are by far the most widely used in object-oriented software engineering (OMG, 2011; Dobing & Parsons, 2006). In fact, most of its diagrams are primarily tailored to support object-oriented software development. It is used to specify, communicate, and document the artifacts of software-intensive systems under development.

UML is defined using a metamodeling approach, i.e., a metamodel is used to specify the models that comprise UML. The UML metamodel is defined based on a 4-layer metamodel pattern. While this approach lacks some of the rigor of formal specification techniques, it offers the advantages of being more pragmatic for most researchers and developers (OMG, 2011). The UML metamodel defines thirteen diagrams, such as the component diagram, the class diagram, the sequence diagram, and the use case diagram (OMG, 2011). Together the UML

diagrams represent two different views of a system model: (1) *structural view*: it emphasizes the static structure of the system using objects, attributes, operations, and relationships. Examples of these diagrams are the class diagram and component diagram, and (2) *behavioral view:* it emphasizes the behavior of the system by showing collaborations among objects and changes to the internal states of objects. Examples of these diagrams are the sequence diagram, the activity diagram, and the state machine diagram.

In this research, we use three UML diagrams: class, sequence, and component diagrams. This choice is not an arbitrary choice, but based on observations drawn on empirical studies reported by Dobing and Parsons in (Dobing & Parsons, 2006). These researchers conducted an OMG-supported survey to investigate which UML diagrams are used in real-world projects more frequently. The survey identified the frequency of use of UML diagrams. The main result of the study was that class diagram is the most-used UML diagram used followed by use case diagram and sequence diagram. Consequently, these diagrams tend to be the diagrams that developers compose.

Additionally, developers usually compose these diagrams in practice (Norris & Letkeman, 2011). The key reason for using these diagram types is their usefulness and adequacy of information as perceived by the models' users. Their selection for this research is also motivated for the fact that there are aspect-oriented counterparts for these diagrams. The aspect-oriented versions of these diagrams are also used in some of our studies. Aspect-oriented modeling is discussed in the following subsection.

2.5.2. Aspect-Oriented Modeling

Separation of concerns is a fundamental principle that addresses the limitations of human cognition for dealing with complexity. Dijkstra advocates to master complexity, one should deal with one relevant concern at a time (Dijkstra, 1976). Parnas reinforces that complexity of software systems should be tamed by decomposing their modules into smaller, clearly separated modular units, each dealing with a single concern (Parnas, 1972). The principle of separation of concerns is employed through the decomposition and modularization of software

systems. The expected benefits are an improved understandability and reuse in complex software systems. In software modeling, the achievement of separation of concerns depends largely on the suitability of abstractions and notations of modeling languages to represent these concerns. Typically, components, classes, and methods are examples of modular units in object-oriented modeling languages, such as UML and its profiles.

Unfortunately, object-orientation has some limitations in dealing with concerns that address global constraints and widely scoped functionalities, such as persistence, error handling, logging, among many others (Sant'Anna, 2008). These concerns have been commonly called *crosscutting concerns* since they naturally crosscut the boundaries of modular units that implement other concerns. Without proper means for separation and modularization in the UML, crosscutting concerns tend to be scattered over a number of modular units (e.g., components and classes) and tangled up with other concerns. Consequently, the cohesion in the modular units tends to decrease, while the coupling between them tends to increase. This can jeopardize the comprehensibility and evolvability of design models. Aspect-orientation (Kiczales et al., 1997) is an approach that supports a new flavor of separation of concerns. It introduces new modularization abstractions and composition mechanisms to improve separation of crosscutting concerns at different levels of abstraction. Aspect-orientation defines a new modular unit, called *aspect*, for separating crosscutting concerns, and provides new mechanisms for composing aspects with other modular units at well-defined points. In the following, we briefly describe the main aspect-oriented abstractions and mechanisms. After that, we illustrate the use of aspect-oriented modeling in the light of an example.

Aspects

Aspect is the term used to denote the abstraction that aims at supporting improved isolation of crosscutting concerns (Kiczales et al., 1997). Aspects are modular units of crosscutting concerns that crosscut a set of modular units — i.e., components, classes, interface, and so on (Sant'Anna, 2008). An aspect can affect, or crosscut, one or more modular units in different ways. Thus, aspect-oriented design models can be decomposed into components, classes, interfaces, and aspects. While aspects modularize crosscutting concerns and the other modular

unit modularize non-crosscutting concerns. In addition to conventional attributes and methods, an aspect includes pointcuts and pieces of advice as described as follows.

Join Points and Pointcuts

Essential to the process of composing aspects and classes is the concept of *join points*, the elements that specify where aspects and other modular units are related. Join points are well-defined points in the dynamic execution of a system (Kiczales et al., 1997). Examples of join points are method calls, method executions, attributes sets and reads, and object initialization. Each aspect defines one or more first-order logic expressions, called *pointcut expressions* (or just pointcuts), to select the join points that will be affected by the aspect's crosscutting behavior (Kiczales et al., 1997).

Advice

When execution of the system reaches a join point, selected by some pointcut expression, an *advice*, can be executed before, after or around it (Filman et al., 2005). Advice is a special method-like construct attached to pointcuts (Kiczales et al., 1997). There are three basic forms of advice supported by most aspect-oriented languages (Kiczales et al., 1997): (i) a before advice runs whenever a join point is reached and before the actual computation proceeds, (ii) an after advice runs after the computation under the join point finishes, i.e., after the method body has run, and just before control is returned to the caller, and (iii) an around advice runs whenever a join point is reached, and has explicit control whether and when the computation under the join point is allowed to run at all.

Therefore, aspect-oriented (AO) modeling languages aim at improving the modularity of design models by providing a range of notations to represent these concepts. It is important to highlight that there are many approaches proposed for AO modeling. Most of them are aimed at representing basic AO concepts also supported by most aspect-oriented programming models. Approaches that are more conservative propose UML profiles (Losavio et al., 2009; Clarke & Banaissad, 2005; Chavez & Lucena, 2002) for supporting AO modeling (Losavio et al., 2009; Clarke & Banaissad, 2005). These techniques are more aligned to classic AO programming models, such as the one realized by AspectJ (Laddad &

Johnson, 2009) and dialects. In these profiles, the modularization of crosscutting concerns, for instance, is achieved by the definition of a new model element, called *aspect*. In general, the notation enables to explicitly distinguish between *aspects* and *classes*. An aspect can crosscut several classes in a system. These relations between aspects and other modules are then called *crosscutting relationships*. Typically, these relationships are motivated by crosscutting concerns.

Having the goal of this work in mind (Chapter 1), we opted for carrying out our investigation regarding UML profiles. Another reason for using AO UML profiles is that the real developers will participate in the empirical studies and these subjects tend to have previous experience with AspectJ (Laddad & Johnson, 2009) rather than with any other AO modeling approach. Thus, the UML profile for aspect-orientated tends to be the best choice for this typical characteristic of aspect-oriented software developers.

These profiles have the advantage of supporting classical AOP concepts at a higher abstraction level. This means that AO key concepts are usually represented via conventional extension mechanisms of the UML such as UML stereotypes. This alternative followed in our studies prevented, for example, classical side effects related to the learning curve in empirical studies. Otherwise, it would not be possible to investigate any causal relationships between design model languages and composition effort without any high overhead to the subjects involved.

It is also important to highlight that UML is the standard for designing software systems. The use of stereotypes reduces the gap between subjects with low and high skilled (or experienced) subjects (Ricca et al., 2010). The other consequence of using UML profiles for AO modeling is that the model reading technique used by the subjects would not be much influenced by new notation issues. Therefore, the use and interpretation of the models are exclusively influenced by the use of the concepts in object-oriented and aspect-oriented modeling. As UML profiles are supported by academic and commercial modeling tools, such as IBM Rational Software Modeling (IBM RSA, 2011), developers are familiar with stereotype notations. Additionally, learning the current state-of-the-art of AO modeling is not a trivial task for developers in early adoption of aspect-oriented programming. Finally, UML profiles for aspect-oriented design is the

approach more common for structural and behavioral diagrams. Based on these reasons, the AOM language used in our study is a UML profile described in (Losavio et al., 2009; Clarke et al., 2005; Chavez & Lucena, 2002).

Figure 2 presents illustrative examples of some aspect-oriented models used in our study: class and sequence diagrams. The notation supports the visual representation of aspects, crosscutting relationships and other aspect-oriented modeling concepts. The stereotype <<aspect>> represents an aspect, while the dashed arrow decorated with the stereotype <<crosscut>> represents a crosscutting relationship. Inner elements of an aspect are also represented, such as pointcut (<<pointcut>>) and advice. An advice adds behavior before, after, or around the selected join points (Losavio et al., 2009; Clarke & Walker, 2005). The stereotype associated with an advice indicates when (<<before>>, <<after>> or <<around>>) a join point is affected by the aspect. The join point is a point in the base element where the advice specified in a specific pointcut is applied.

With this in mind, we discuss the limitations of the related work regarding the effort of detecting inconsistencies and empirical studies on software modeling.

2.5.3. Limitations of Related Work on Design Modeling Languages

Many design modeling languages have been proposed in recent years, such as UML and its extensions (OMG, 2011). Some empirical studies have also been performed with these languages in order to understand their usefulness in different contexts. For instance, AOM languages will be considered useful compared to traditional modeling techniques if the claimed improved modularity of aspectual design decompositions actually leads to practical benefits, such as reduction of inconsistency detection effort and misinterpretations. Unfortunately, it is well known that empirical studies of AO modeling are rare in the current literature, which confirms that it is still in the craftsmanship era (France & Rumpe, 2007).

Research has been mainly carried out in two areas: (1) defining new AOM techniques, and (2) proposing new weaving mechanisms for design models. First, several authors have proposed new modeling languages, focusing on the definition of constructs, such as <<aspect>> and <<crosscut>>. These constructs represent concepts of aspect-orientation as UML based extensions (Losavio et al., 2009;

Chavez & Lucena, 2002). In addition, Clarke and Baniassad (Clarke & Banaissad, 2005) make use of UML templates to specify aspect models.

On the other hand, the chief motivation of some works is to provide a systematic method for weaving aspect and base models (e.g., (Whittle & Jayaraman, 2010; Jézéquel, 2008; Klein et al., 2006). For example, Klein and colleagues in (Klein et al., 2006) present a semantic-based aspect-weaving algorithm for hierarchical message sequence charts (HMSC). They use a set of transformations to weave an initial HMSC and a behavioral aspect expressed with scenarios. Moreover, the algorithm takes into account the compositional semantics of HMSCs.

Unfortunately, most of empirical studies on aspect-orientation are focused on assessing implementation techniques. For example, Hanenberg and colleagues (Hanenberg et al., 2009) compare the time invested by developers to implement crosscutting concerns using object-oriented and aspect oriented programming techniques. Other studies focus on the assessment of aspect-oriented programming under different perspectives, such as software stability (Ferrari et al., 2010; (Greenwood et al., 2007) and fault-proneness (Burrows et al., 2010). However, empirical studies about AO modeling have not been conducted in particular in the context of modeling inconsistencies (or defects). Only the literature on OO modeling does highlight that empirical studies have been done on identifying defects in design models (Langes & Chaudron, 2004). Lange (Langes & Chaudron, 2006a) investigates the effects of defects in UML models. The two central contributions were: (1) the description of the effects of undetected defects in the interpretation of UML models, and (2) the finding that developers usually detect more certain kinds of defects than others do.

In particular, in this thesis, we aim at studying certain effects on model composition from one of the most prominent and recently proposed approaches to achieve separation of concerns at design level: aspect-oriented modeling language (Clark & Walker, 2005; Losavio et al., 2009). In addition, our other focus is on analyzing the empirical studies on UML and AO modeling. We reinforce that aspect-oriented modeling supports early separation of otherwise crosscutting concerns in software design. An improved modularization may ameliorate one of the main purposes of using of design models: communication. If developers communicate properly, so the interpretation of the models is also proper. Thus, we

analyze empirical studies investigating the side effects of inconsistencies on the interpretation of the design models and the effort invested by developers to detect them. In conclusion, there are two critical gaps in the current understanding about AOM that are addressed in this thesis: (1) the lack of practical knowledge about the developers' effort to localize inconsistencies, and (2) the lack of empirical evidence about the detection rate and misinterpretations when understanding AO and OO models.

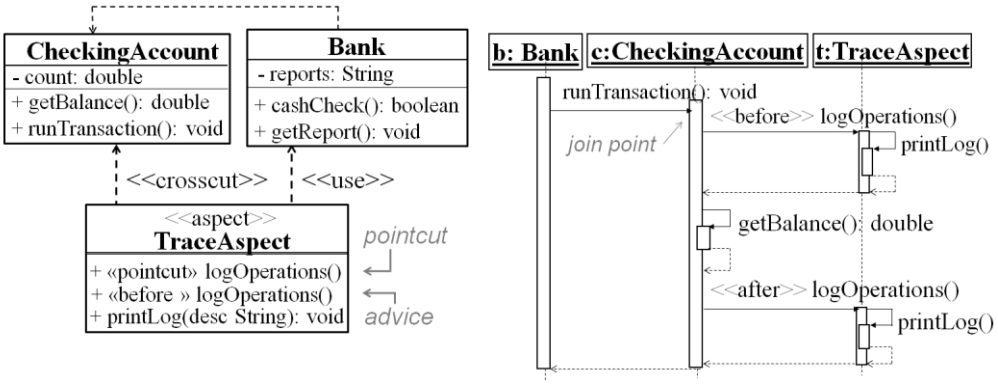

Figure 2: An illustrative example of AO models used in our study.

2.6. Design Characteristics

Researcher investigates how design characteristics, such as design stability, can influence the evolution of software artifacts (Kelly, 2006; Martin, 2003). In this thesis, we study whether the model stability can affect the composition effort. In the next section, we discuss how model stability is addressed in this thesis.

2.6.1. Model Stability

Developers need an indicator to identify the most severe composition cases in which the output composed models produced have a high number of inconsistencies and require a great deal of the developers' effort to be transformed into an output intended model. Without this indicator, it is particularly challenging for developers to exam hundreds of output composed models produced in a

collaborative software development environment. In this thesis, we investigate if the model stability can be this indicator.

In practice, the stability of the output composed model can be computed based on the internal design characteristics of (evolving) models. According to (Kelly, 2006), a design characteristic (e.g., coupling and cohesion) is stable if, when observed over two or more versions of the software, the differences in the metric associated with that characteristic are considered small. With this in mind, we can consider the output composed model as stable if its design characteristics have a low variation regarding the characteristics of the output intended model.

In our study, we define low variation as being equal to (or less than) 20 percent. This choice is based on previous empirical studies (Kelly, 2006) on software stability that has demonstrated the usefulness of this threshold. For example, if the measure of a particular characteristic (e.g., coupling and cohesion) of the output composed model is equal to nine, and the measure of the output intended model is equal to 11. So the output composed model is considered stable in relation to the output intended model (because nine is 18% lower than 11) with respect to the measure under analysis. Following this stability threshold, we can systematically identify whether (or not) the output composed model remains stable in a particular evolution scenario or not. This threshold has been used more as a reference value rather than a final decision maker. Although its effectiveness has been demonstrated in (Kelly, 2006), we will also analyze in our empirical studies if this threshold can be, in fact, used to indicate the most severe composition cases in which an elevated number of inconsistencies and require a great deal of the developers' effort to resolve these inconsistencies. This investigation is realized in Chapter 6.

We will carry out this new analysis because this threshold plays a crucial role in the identification of the output composed models that will be reviewed by the developers. The identification of stable and unstable output composed models is based on the study of the differences between the measures of the design characteristics of the output composed model and the output intended model. These differences are calculated comparing the measures of each characteristic of the design models. We use a suite of design metrics to quantify such characteristics of the models used in our study. The metrics can be seen in the next Chapter 3 (Table 5, Table 6, and Table 7), and Chapter 6.

These metrics were used because they are conventional metrics and they have been used previous works e.g., (Martin, 2003; Kelly, 2006; Fenton & Pfleeger, 1997), which have tested the effectiveness of these indicators for the quantification of design characteristics. We are also interested in identifying evolution scenarios where composition techniques are able to effectively accommodate changes from the delta model in the base model. The quantification method of model stability is presented later in Section 6.1.2.4. With this in mind, the next step is to discuss the limitations of related works considering the subject.

2.6.2. Limitations of Related Work on Design Characteristics

The current literature in software design has defined a set of characteristics that can be used to measure the quality of a design in terms of the interdependence between the modules of that design (Martin, 2003). A pivotal example of such characteristics is the software stability as previously mentioned in Section 2.6.1. According to (Martin, 2003), when we design software, we strive to make it stable in the presence of change. In fact, stability is at the very heart of all software design discipline.

Some works about design stability have been conducted in recent years such as (Kelly, 2006; Martin, 2003). Kelly has demonstrated the usefulness of stability to software maintenance. For this, she presents a method for examining software systems that have been actively maintained and used over the long term. The method relies on a criterion of stability and a definition of distance to flag design characteristics that have potentially contributed to the software maintenance (Kelly, 2006). The main contribution is the demonstration that the method is useful to provide insight into the relative importance of individual elements of a set of design characteristics for the long-term evolution of software. On the other hand, Martin (Martin, 2003) provides a definition of software stability and shows how the characteristic can be applied.

Unfortunately, we have observed that the existing literature in model composition and software design has failed to provide metrics or studies for empirically revealing the effects of stability on model composition effort. Thus, we see our work as the first step to investigate empirically the interplay between

stability and model composition effort. In other words, nothing has been done to investigate the use of stability as an indicator of severe cases of composition effort.

The absence of studies exploring this relationship prevents developers from understanding the influence of stability on the developers' effort. Without this knowledge, developers end up relying on the evangelist feedback, rather than empirical data, to comprehend how well the composition effort can be. In conclusion, these works differ in their aims to the work presented in this thesis. This thesis does not propose how to come up with a good guidance to design software, neither proposes any particular method to quantify stability. Rather, we empirically evaluate how stability influences the developers' effort when composing models (Section 6.1). We defer further consideration about this topic to Section 6.2.4.

2.7. Concluding Remarks

In this chapter, we have presented the main concepts discussed throughout this thesis. To begin with, we describe the three purposes of using model composition. After that, we analyzed the characteristics of design modeling languages that can affect the use of model composition. Three characteristics are discussed: the lack of a rigorous definition, the multi-view design modeling languages, and the complexity of the design modeling languages.

We also revisit the purpose of using design models. The empirical studies use design models for different particular purposes. This happens because we need to investigate the effort of composing design models from alternative perspectives. More specifically, we study the use of design models for three purposes: communication, comprehension, and documentation for maintenance.

Moreover, following the description of the basic terminology used in this thesis, we present the concepts associated with three key factors potentially influencing mode composition effort: composition techniques, design modeling languages, and design characteristics. After mentioning these three factors, we try to discuss how each factor can affect the effort of composing design models in practice.

Observing the related works, the major conclusion is that nothing has been done to evaluate the impact of such three influential factors on model composition effort. In fact, some works such as (France & Rumpe, 2007) emphasize the need for further researches in order to generate a clear understanding about the effects of these factors on model composition effort. For example, several composition techniques have been proposed and used in practice. However, little has been done to quantify the effort invested by developers to compose design models. Without studies that evaluate whether the effort invested is worthwhile or not, it is not possible to recognize the benefits of using composition techniques. This lack of knowledge about the effects of the composition on the developers' effort is also extended as to the other two factors: design modeling languages and design characteristics. To date, the literature fails to provide insight on the influence of these two factors on the composition effort. For example, researchers and developers do not know if by using a particular design modeling language, they will minimize the composition effort on the parts of the design model created in parallel by different software development teams.