Model-Driven, Aspect-Oriented Development for Mobile, Context-Aware Computing

Andrew Thomas Carton, B.A. (Mod)

A dissertation submitted to the University of Dublin, Trinity College,

in partial fulfillment of the requirements for the Degree of

M.Sc. in Computer Science

University of Dublin, Trinity College

September 2007

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Andrew Thomas Carton

September 13, 2007

Permission to Lend and/or Copy

Acknowledgments

I would like to thank my supervisor, Siobhán Clarke, for her guidance and support throughout the year. Her expertise and insights into this work greatly enriched my knowledge on this subject. I also owe a large debt of gratitude to Andrew Jackson for his continuous advice, encouragement and his exceptional insights that contributed to the success of this work. Thanks especially to the NDS class, for an enjoyable and memorial year! Finally, to my family, for their love, support and encouragement.

Andrew Thomas Carton

University of Dublin, Trinity College
September 2007

Model-Driven, Aspect-Oriented Development for Mobile, Context-Aware Computing

Andrew Thomas Carton, M.Sc.
University of Dublin, Trinity College, 2007

Supervisor: Siobhán Clarke

Technological innovation has encouraged the proliferation of small, portable mobile devices such as Personal Digital Assistants (PDAs), mobile phones, laptops and sensors. These devices integrate seamlessly into facets of everyday life, with their requirements presenting new challenges to the software engineer.

One of the requirements of context-aware computing necessitates applications to dynamically adapt their behaviour based on context, such as environmental factors, device limitations, mobility and connectivity. A combination of these context-based concerns are observed to have an adverse impact across application design, manifesting themselves throughout and within other concerns in the forms of scattering and tangling. Aspect-oriented software development (AOSD) presents a solution to modularising these types of crosscutting concerns and promotes improved maintainability, comprehensibility and manageability.

Model-driven development (MDD) is an emerging technology, that emphasises the use of models and powerful abstractions to allow the software engineer the ability to reason about the application semantics separately from the technical specifics of the implementation. While this is an advancement in general for software engineering, the potential range of interacting heterogeneous devices and diverse computing platforms and deployment environments in play for context-aware computing make this an important contribution.

While observing the individual contributions of both these emergent technologies, it seems intuitive that they would mutually complement each other, in a single approach. The motivation of this dissertation is to present such an integrated approach in an attempt to demonstrate, illustrate and evaluate its applicability to the domain of mobile, context-aware computing. The evaluation findings suggest that AOSD and MDD work synergistically to provide a valuable contribution for this domain.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter provides a short and succinct introduction to the field of mobile, context-aware computing; highlightes the problems and challenges that motivate this dissertation, introduces an approach that solves these challenges, summarises the contributions and evaluation and provides an outline for the rest of the dissertation.

1.1 Mobile, Context-Aware Computing

Technological innovation in the areas of wireless technology, protocols, sensors and miniaturisation has lead to the widespread use of mobile devices. Weiser's vision of mobile devices integrating seamlessly and unobtrusively into the environment has now been actualised [56]. Not only are devices such as laptops, mobile phones and PDAs becoming cheaper and more popular, but computers such as embedded devices, wearable computers and smart appliances are now becoming active areas of research that are beginning to filter their way into the fabric of society. This vision includes all of these devices being able to communicate and adapt their behaviour to the requirements and needs of the user.

There are many definitions for context but one of the most suitable for this domain is given by Dey [14]:

> Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the applications themselves.

From this definition, contextual information can be anything from location, computer resources, environmental factors or bandwidth. Context-aware software surveys the environment to detect changes in these numerous contexts and is capable of adapting and reacting to these changes [49].

1.2 Motivation

The previous section described what a context is and what it means for software to be context-aware. Mobile, context-aware software must handle a combination of these interacting context-aware concerns which increase the design complexity and presents new challenges to traditional software engineering approaches. The design of mobile, context-aware applications is difficult for three reasons [13]. First, there are no good software engineering guidelines or principles for handling context. Application behaviour based on context adaptation is observed to cut across the additional application logic, leading to poor concern localisation. Consequently. these context-aware concerns are scattered throughout multiple modules and tangled with other concerns. Second, there is a lack of suitable abstractions to think about context. Approaches that propose to ease development of context-aware applications such as middleware or frameworks tend to only support a single application domain or deployment environment and are too inflexible over designing in an ad-hoc, application specific way [50]. Third, context sensing tends to be distributed, leading to complex designs that span across a wide range of interacting heterogeneous devices. Traditional approaches such as third-generation programming languages and component-based technologies were designed to be deployed on desktop and server machines. A new paradigm shift is needed to be able to cater for context-aware applications that are highly dynamic and mobile. With the wide range of diverse computing platforms and deployment environments available, from PDAs to small embedded wearable devices, there is a strong need for designs to be platform-independent.

With these challenges motivates an approach that attempts to alleviate and solve some of the difficulties that face mobile, context-aware computing.

1.3 Approach

This dissertation proposes a new approach to tackle the challenges faced by mobile, context-aware computing by combining the research on Aspect-Oriented Software Development (AOSD) and Model-Driven Development (MDD).

1.3.1 Aspect-Oriented Software Development

AOSD is a maturing paradigm that extends the decomposition and composition mechanisms of existing paradigms such as object-orientation, in order to effectively modularise crosscutting concerns. As previously discussed, there is a lack of techniques, guidelines and necessary abstractions for software engineers to handle context. As concerns that relate to context adaptation are observed to be crosscutting, AOSD offers a solution by effectively localising these concerns in their own concern space or module called an aspect [39]. Theme/UML [10] is a modelling language that supports the concept of an aspect, allowing concerns that relate to context adaptation to be effectively abstracted and modularised [5].

1.3.2 Model-Driven Development

MDD is an emerging technology, that emphasises the use of models and powerful abstractions to allow the software engineer the ability to reason about the application semantics separately from the technical specifics of the implementation. One of the challenges previously mentioned was the potentially highly distributed, complex designs needed for a wide range of interactive heterogeneous devices. MDD attempts to alleviate these difficulties by emphasising the use of different layers of separation. In effect the business logic of an application can be modelled as a Platform-Independent Model (PIM) with mappings encapsulating the domain-specific knowledge needed to transform it into a Platform-Specific Model (PSM).

1.3.3 A Synergistic Union

The approach this dissertation proposes involves combining both AOSD and MDD to gain the significant benefits of both in the domain of mobile, context-aware computing. The approach involves three steps (c.f. Figure 1.1). The first step is to realise Theme/UML as a PIM to allow context-aware concerns to be modelled. The second step is to devise and implement an algorithm that maps this aspect-oriented PIM into an object-oriented representation. The third step involves transformations mapping this object-oriented design to two mobile environments, J2ME and .NET CF.

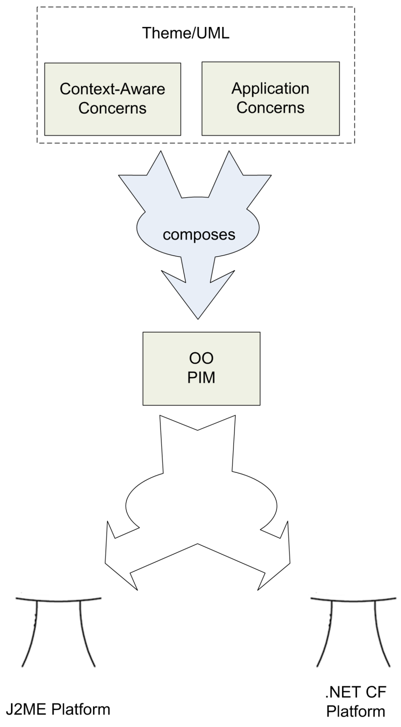


Figure 1.1: Proposed Approach

There are a number of challenges to designing this approach. Firstly, the approach must realise Theme/UML, an aspect-oriented modelling language based on composition patterns [9] and an extension of version 1.3 beta R7 of the Unified Modelling Language (UML). The UML version is currently 2.1.1. Realising Theme/UML with a model-driven solution involves updating its composition semantics to comply to the current version. Secondly, Theme/UML currently conforms to a MOF-based heavyweight graphical extension of the UML, which is infeasible to implement with current tool support. One of the challenges is a means to specify the graphical extensions to Theme/UML with current MDD tools. Thirdly, model-driven based transformation tools are currently in their infancy, especially those involving the UML. While Model-to-Text (M2T) tools are considered mature, Model-to-Model (M2M) tools are still evolving [12].

This approach aims to solve the three problems discussed in the previous section. First, this approach supports the software engineer to be able to effectively modularise context-based concerns, with the use of aspects, significantly improving maintainability, comprehensibility and manageability. Second, the use of aspects and models synergistically compliment each other to provide better abstractions to handle context-aware software. Third, MDD supports and encourages the *"Design once, build it on any platform"* rationality, allowing a context-aware design to be reused on many platforms and deployment environments.

1.4 Contribution

The previous sections presented a synopsis of the difficulties faced by mobile, context-aware computing and introduced a new approach to alleviate some of these difficulties. This section briefly clarifies and summarises the contribution that this dissertation motivates.

1.4.1 Model-Driven Theme/UML

Currently, there is no tool support for Theme/UML that realises the composition or mappings to different aspect languages. Based on the old UML 1.x and MOF standards, Theme/UML was defined before the evolution of model-driven concepts, standards and technologies. The Object Management Group (OMG) has updated its standards since then towards the Model-Driven Architecture (MDA) vision. As such, some of the semantics and constructs used to define Theme/UML may not be fully compatible or suitable with current standards or conventions. One of the contributions of this dissertation is to design a tool for Theme/UML that complies with current MDA standards and conventions, while trying to adhere to the general purpose and intention of its semantics as they were originally defined.

1.4.2 Aspect-Oriented Modelling Weaver

One of the contributions of this dissertation is to design and implement a transformation to compose an aspect-oriented design in Theme/UML to an equivalent object-oriented representation. The transformation involves weaving or composing the aspect themes into the base themes. There are two advantages for composing to an object-oriented representation. First, object-oriented programming languages are more mature than their equivalent aspect-oriented counterparts for mobile deployment environments. Second, the designer gains the powerful decomposition and composition capabilities of AOSD at design time without being tied to an aspect-oriented platform.

1.4.3 Mappings to J2ME and .NET CF

Although Theme/UML mappings exist for aspect-oriented languages HyperJ [11], CME [10] and AspectJ [10], there exists no tool that realise these mappings and they must be transformed manually. Moreover, there only exist mappings for aspect-oriented languages. From the result of the aforementioned model weaver, the object-oriented PIM is used to further transform to the mobile platforms J2ME and .NET CF. These two platforms were designed as a proof-of-concept to illustrate the approach, but many more mappings are possible to different deployment environments.

1.5 Evaluation

A mobile, context-aware case-study was used to evaluate against the three challenges that motivated this work. Through study of this case-study, the evaluation findings suggested that this approach was favourable for designing mobile, context-aware applications offering better guidelines and abstractions for handling context and supporting the need for a single design targeting multiple different mobile devices.

1.6 Dissertation Outline

Chapter 2 - Model-Driven Architecture
This chapter gives background information on Model-Driven Architecture (MDA), a standardised MDD approach. It explains the use of models and metamodels, gives an outline of the MDA process and introduces the terminology and concepts involved with MDA.

Chapter 3 - Aspect-Oriented Software Development
This chapter gives background information on Aspect-Oriented Software Development (AOSD). It discusses the advantages of aspects over traditional paradigms, explains the rationale for the choice of aspect-oriented modelling language, explains the terminology of aspects and gives an introduction to Theme/UML.

Chapter 4 - State-of-the-Art
This chapter gives a review of the state of the art in model-driven aspect-oriented weavers, in order to gain insights into how Theme/UML can be integrated as a model-driven solution.

Chapter 5 - Design
This chapter describes the design of the proposed approach. The first section discusses the initial

design decisions. The second section gives an overview of the whole process. The rest of the chapter is divided into three main sections, detailing each phase of the process - the modelling phase, the composition phase and the transformation phase.

Chapter 6 - Implementation
This chapter discuss the implementation of the different phases. The chapter is split into three main sections, each representing each phase, the modelling phase, the composition phase and the selection, transformation and synthesis phase.

Chapter 7 - Evaluation
This chapter presents a mobile, context-aware case study to evaluate the outlined approach that this dissertation presents. This case study demonstrates how a domain-specific, mobile context-aware concern that emerges from a requirement specification can be modelled separately from the application concerns and then illustrates how it can be composed with them. The motivation is to evaluate and ascertain the suitability of this approach for mobile, context-aware systems. The results highlight the strengths and weaknesses of using models and aspects in such a way for this domain.

Chapter 8 - Conclusion and Future Work
This chapter discusses the objectives of this dissertation and how they were met. It reflects on the work that was done and suggests some suitable future work.

Chapter 2

Model-Driven Architecture

2.1 Introduction

Model-Driven Architecture (MDA) is a proposed MDD framework, standardised by the non-for-profit organisation, Object Management Group[1] (OMG). Traditionally, models were only used as a means

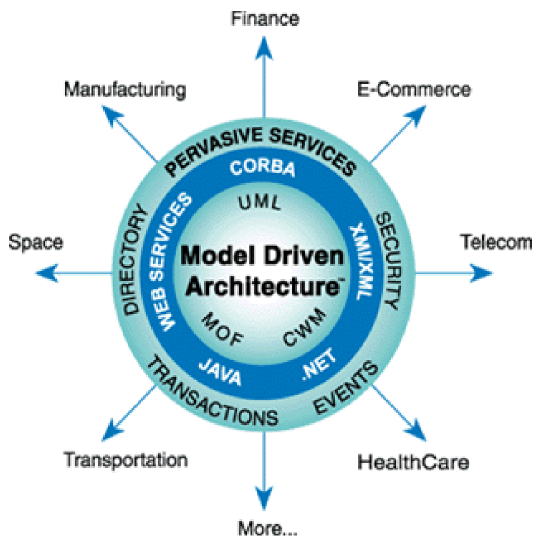


Figure 2.1: MDA Logo

for documentation, providing a guide to implementation. MDA emphasises the use of models as the main artefacts throughout the software development life-cycle. In doing this, it enables a system to be modelled independently from the platform(s) that realise it, allowing automated transformations to map abstract platform-independent models to more concrete ones targeting one or more specific

[1]http://www.omg.com

platforms. With this architectural separation of concerns, the main goals of MDA are to provide portability, interoperability and reusability [37]. The OMG provides guidelines and standards such as MOF, UML and CWM that describe the process.

2.2 Models and Metamodels

OMG gives the definition of a model as:

> A *model* of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language.

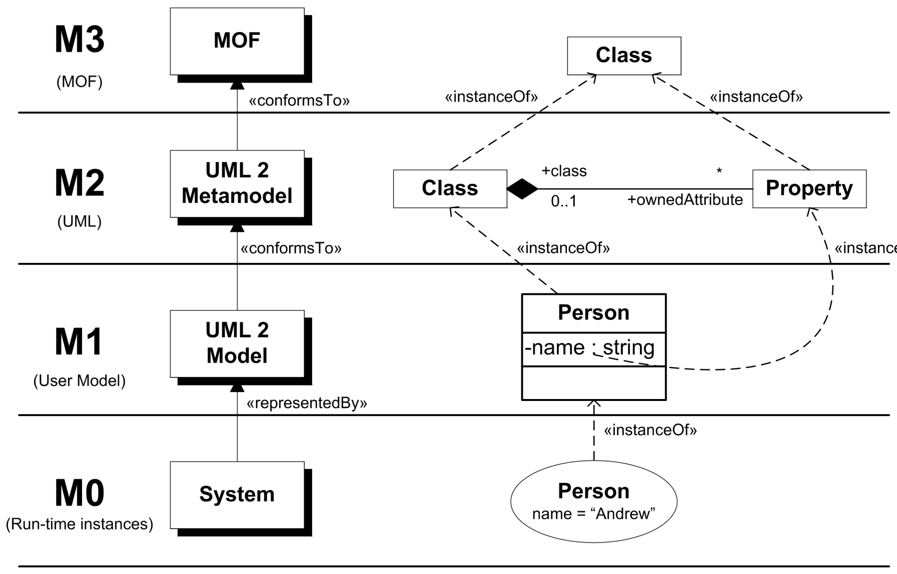


Figure 2.2: Example of Four-tier Metamodel Architecture

From this definition, a model is taken to be any abstract representation that describes a system. A model conforms to a metamodel, in so far as the model is described using the semantics of the metamodel. In standard MDA, the root metamodel (or the meta-metamodel) of all model definitions must conform to the Meta-Object Facility (MOF) standard. The MOF is defined in terms of itself which ends the meta* cycle. Figure 2.2 illustrates a simple example of this four-tier architecture. `M0` exemplifies an object named `Andrew` with type `Person` contained within a system. At this level, artefacts represent run-time instances. Moving up to the next layer, `M1` is the user model which describes the system. In this example, an UML class diagram models the concept of a `Person`. The object at `M0` is an instance of this model. A designer describing a system would work at this layer. The upper layers of the architecture `M2` and `M3` provide definitions for their lower layers counterparts.

M2 is the UML metamodel and the diagram illustrates how the model at M1 is defined. Likewise, MOF at M3 illustrates how the UML metamodel at M2 is defined.

A parallel can be drawn between a model and an object. When object-orientation was advancing in the 80's, the principle "Everything is an object" was used to better understand the technology. An *Object* is an *instanceOf* a *Class* which *inheritsFrom* a *SuperClass*. Likewise the phrase "Everything is a model" can now be used synonymously with object-orientation terminology. A *System* is *representedBy* a *Model* which *conformsTo* a *Metamodel* [3].

2.3 MDA Process

MDA presents three different viewpoints for a model . These different viewpoints are represented individually as a Computation Independent Model (CIM), a Platform Independent Model (PIM) and a Platform Specific Model (PSM). These concepts will be explained with an example. Envision an application programmed in Java with a SQL database backend. Figure 2.3 illustrates how an application like this is designed in MDA from requirements to code.

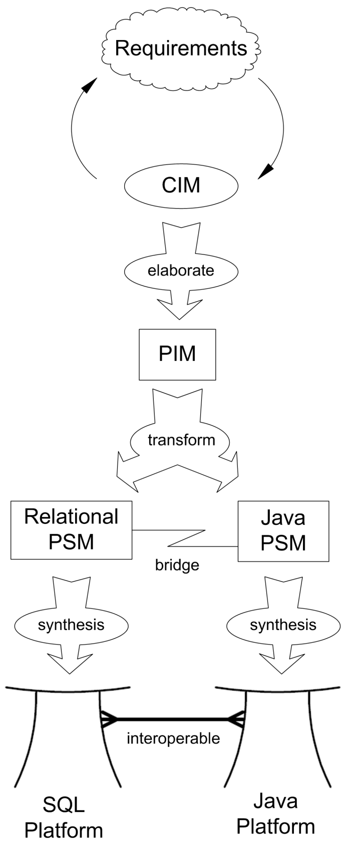


Figure 2.3: MDA Process

Similar to traditional software processes, the designer(s) start off with a set of stakeholder requirements (c.f. Figure 2.3). A Computation Independent Model (CIM) is synonymous to a domain or business model. Requirements and modelling is illustrated as an iterative process, allowing domain experts to better understand them, providing a set of shared concepts and vocabulary for use in other models and illustrating what the system is expected to do. It makes no assumptions about how the system is to be implemented or how the requirements are to be realised. Artefacts of a CIM should be traceable to the PIM and PSM artefacts that realise them.

From the CIM, the designer elaborates a detailed design. A Platform Independent Model (PIM) is a model that is an abstract representation of a system but is not dependent on any one platform. Its artefacts are common enough to provide the structure and behaviour of a system that is suitable for a number of different platforms of the same kind. The UML is a general purpose modelling language defined by the OMG and is suitable for use as a PIM.

Using transformations, a set of PSMs are generated from a PIM. A Platform Specific Model (PSM) is a concrete realisation of a PIM for a specific platform, focussing on how the specification is realised using platform specific constructs. A mapping is a specification that defines rules to transform elements of a model conforming to a specific metamodel into elements of another model that also conform to a metamodel. The metamodels may be the same or different. A PIM may be transformed into one or more PSMs with bridges allowing communication between them. Figure 2.3 demonstrates a bridge between the Java PSM and the Relational PSM, representing communication through some medium, for example an API or a protocol over a network connection.

Synthesis is the terminology used to describe how a model is transformed into code for a specific platform. In this diagram, the Java and Relational PSMs are transformed into their platform specific counterparts with an interoperable line between them illustrating the realisation of the bridge.

2.4 Marking

Marking is a concept illustrated in the MDA Guide that suggests marking elements of a PIM to guide the transformation.

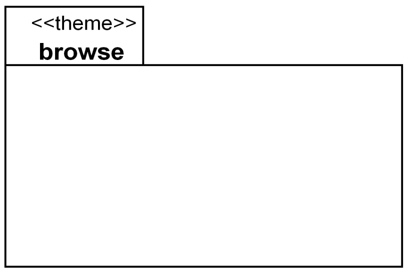


Figure 2.4: Marking Example with an UML Profile

A mark is a simple concept involving tagging an element indicating a certain mapping rule is to be used during the transformation. Elements can be marked several times, indicating that the element is involved in more than one mapping rule. A mapping rule may also involve a set of marks across many elements to support a full transformation specification. A tool may also prompt a user during a transformation, if the source model can not contain enough information for the transformation to complete. Marking is generally considered non-invasive [35]. The UML Profile is suitable means to mark a PIM in the UML. Stereotypes provide definitions of how existing UML metaclasses can be extending. Stereotypes can have properties called tag definitions. When applied these definitions become tagged values.

2.5 Types of Model Transformation

The main approaches suggested by the OMG in the MDA Guide emphasis transformation approaches from PIMs to PSMs. It is also important to illustrate the other types of transformations that are possible in the MDA approach. A taxonomy is provided that distinguishes between the different types of model transformations possible [36]. A transformation may be classified as *endogenous* if the metamodels of both the input and output models are the same and *exogenous* if they are different. Examples of an endogenous transformation include *optimisation* and *refactoring*, while *synthesis* (code generation) and *reverse engineering* are considered exogenous. Furthermore, it is also relevant, in the scope of this dissertation, to distinguish between *horizontal* and *vertical* transformations. A vertical transformation is one where the source and target models are at different levels of abstraction such as *refinement*. A horizontal transformation, such as *refactoring* is one where both the source and target models are at the same level of abstraction. It is important to note here, that a transformation to compose an aspect-oriented model to an object-oriented one is clearly horizontal.

2.6 Methods of Model Transformation

The MDA recognises that all transformations may not be suitable for automation. The standard advises that approaches with a combination of automatic and manual transformation may be necessary. It illustrates four different approaches, full manual transformation, profile based transformations, pattern based transformations using marks and full automatic transformations. While manual transformations offers no automation between two models, full automatic transformations require that the PIM is computationally complete. This means that the full specification is available as a PIM and can be transformed directly to code. In effect, the PSM is the actual code. This distinguishes two types of MDA patterns, elaborational and translational approaches. Elaborational approaches emphasis the incremental elaboration of more and more details in models through different layers of abstraction and refinements. Eventually the final refinement produces code. Translational approaches such as those proposed by Shlaer-Mellor [51], aim to make analysis models so precise that the design can be directly translated to the target environment without elaboration. Executable UML [35] is such an approach using the UML Action Semantics to specify precise model behaviour. The appraoch suggested in this dissertation is an elaborational one.

2.7 MDA Standards

This section will give a description of some of the MDA standards.

2.7.1 UML

The Unified Modelling Language (UML) is the de facto standard modelling language in industry. The current version is 2.1.1. Its generalities and platform-independence make the language suitable for a wide range of application domains. UML has numerous different model types that support different views of a system throughout the life-cycle - from requirements analysis, architectural descriptions, structure and behaviour specifications and deployment scenarios. The UML is defined according to the OMG MOF standard.

2.7.2 MOF

The Meta Object Facility (MOF) is a standard that provides metadata management and provides metadata services for the development and interoperability of model-driven solutions. The MOF is an important MDA standard that is used to define the UML as well as a number of other important MDA metamodels. The MOF supports the definition of other metamodels. MOF is so called the meta-metamodel as it is the ultimate definition of all metamodels and is defined in terms of itself.

2.7.3 XMI

The XML Metadata Interchange (XMI) is a specification that describes how XML documents can be produced from instances of MOF models and also XML schemas to allow validation of these documents. The motivation is to produce an open standard that promotes interoperability between different tools and vendors.

2.7.4 OCL

The Object Constraint Language (OCL) is a side-effect free formal language that can be used to describe expressions on models of the UML. These expressions can be used to specify invariants of a the system being modelled. OCL can also be used to query over objects in a model.

2.7.5 QVT

The Queries/Views/Transfomrations is the standard for defining model transformations. The standard is still currently being finalised. The QVT defines three languages, *Core*, *Relations* and *Operational*. The *Core* and *Relations* languages are declarative and are structured according to a two-level architecture with a mapping between them. The *Core* language is defined as a metamodel that

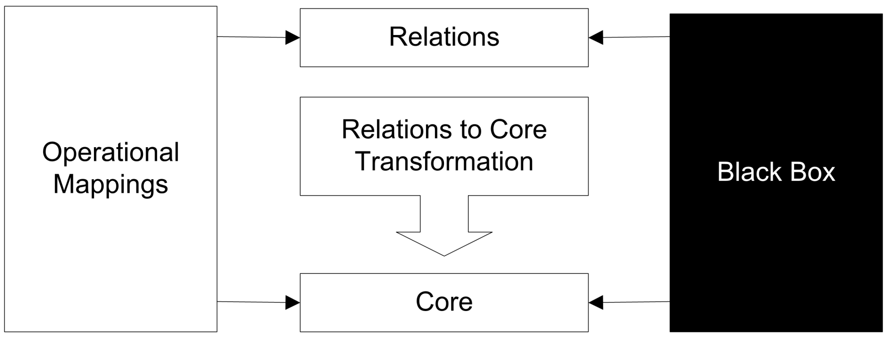


Figure 2.5: QVT architecture

combines both a subsection of the MOF and OCL standards. The *Relations* language is a more user-friendly declarative language, supporting complex pattern matching and object template creation. The *Operational* language extends the other two and provides support for imperative constructs such as loops and conditions. The *Black Box Implementation* is used to support other non-QVT transformation services not defined.

2.8 Summary

This chapter explained MDA, the concepts of models and metamodels, illustrated the MDA process and summarised a number of OMG standards.

Chapter 3

Aspect-Oriented Software Development

3.1 Introduction

Software systems are continuing to increase both in accidental and essential complexity [18]. While essential complexity remains static and inherent to the actual concerns of the problem domain, accidental complexity is that which is introduced through their manifestation in the solution domain. In software engineering, such essential complexities are dealt with by using various degrees of structural and behavioural abstractions to provide a *separation of concerns* [4]. Separation of concerns is a fundamental principle of software engineering [43] [16]. A concern is a specific set of related requirements or interests. Decomposing a software system into a number of orthogonal concerns provides the software engineer with the necessary abstractions to compose a complex system. Each separate concern can be designed individually to each other and then later composed. In the object-oriented paradigm, decomposition is primarily by class, object and method.

However, when a concern can not be decomposed and modularised by the dominant decomposition of that formalism, that concern is forced to become scattered and tangled across and within other concerns [54] [31]. This type of concern is termed to be crosscutting and adversely affects the maintainability, comprehensibility and manageability of a software system.

Aspect-Oriented Software Development (AOSD) [17] is a maturing paradigm, improving on previous paradigms by emphasising advanced separation of concerns at all stages of the development life-cycle. The motivation of this paradigm is to provide improved identification, composition and decomposition techniques and methodologies to better handle concerns, including those which are crosscutting. While aspect-oriented programming languages have reached maturation at this stage, with a plethora of aspect-based extensions for the majority of programming languages, the work on *Early Aspects* [46] is still evolving. Work in the areas of requirements analysis [7] [38] [30], architecture design [44] [1] and detailed design [25] [26] are still areas of active research as well as methodologies to integrate these separate research areas [10] [6] [48]. The scope of this dissertation is concerned with aspects in the design space, particularly those with an existing foundation in models[1].

3.2 Choice of AOM Language

One of the motivations for this work is to provide an integration of both AOSD and MDD. In the design space, there are many Aspect-Oriented Modelling (AOM) languages. Twenty-one such AOM languages were surveyed [27], concluding Theme/UML [10] was unique, providing support for symmetric decomposition and well defined composition specification semantics [29]. Theme/UML was one of the first aspect-oriented approaches which is extensively documented and has withheld the test of time. With the foundation of its evolution based in Subject-Oriented Programming (SOP) [22] and its integration of early AspectJ constructs [32], Theme/UML is semantically defined in the UML to support both symmetric decompositions and hybrid asymmetric/symmetric compositions [8]. This means, it gains the powerful multi-dimensional separation of concerns gained from SOP as well as supporting direct mappings to asymmetric programming languages such as AspectJ.

3.3 Asymmetric Vs. Symmetric

AOSD has evolved from a number of different approaches and technologies such as viewpoints [41], adaptive programming [34], subject-oriented programming [22], AspectJ [32] and Composition Filters [2]. From these, two main forms or notions of an aspect has evolved called asymmetric and symmetric [23]. Paradigms like AspectJ support an asymmetric view, allowing an aspect to be described and composed in relation to other base modules (like a class). Symmetric paradigms like Subject-Oriented Programming have a single composable unit with n-ary relationships dictating how these individual units are to be composed with each other. Theme/UML, evolving from Subject-Oriented Programming supports symmetric decomposition and a hybrid composition, so it is applicable to discuss the terminology of both approaches.

3.3.1 AspectJ terminology

AspectJ is one of the first aspect-oriented programming languages to take fruition. As such, AspectJ terminology is widely used to explain AOSD concepts. It provides constructs to support the modularisation of crosscutting concerns both dynamic and static.

Static crosscutting is supported through *inter-type declarations* allowing for existing types such as classes to be extended. Dynamic crosscutting is supported through AspectJ's join point model. *Join points* are well defined points such as constructor, method calls or exception handlers in the exection of a program. The *pointcut* construct defines a collection of join points. *Advice* declares how a pointcut should be executed, such as *before*, *after* or *around*. Around advices allows the exectuion

[1]For a comprehensive list refer to [27]

of the pointcut to be pre-empted or replaced. An *aspect* is a unit of modularisation that encapsulates a crosscutting concern and has state and behaviour. They are declared in AspectJ like a class with pointcuts and advice defining the where and when of its activation.

3.3.2 Subject-Oriented terminology

In the subject-oriented paradigm [22], each subject is modelled and defined according to its unique, subjective perspective. Each subject has its own state, behaviour and class hierarchy that is programmed with this perspective in mind. Subjectivity makes no other existential assumptions except for this perspective, making the definition of a subject self-contained and declaratively complete. Decomposing a set of requirements is relatively straightforward in this paradigm, where each requirement or similiar set of requirements can have its unique subject. Decomposing in this way means that different subjects may have overlapping concerts. In order to integrate all subjects to a complete application, it is necessary to identify and rectify these overlaps through composition rules. Composition rules include ways to match the unique class hierarchies of each subject to resolve these overlaps.

3.4 Theme/UML

The Theme Approach [10] is an aspect-oriented methodology that supports requirements analysis (Theme/Doc), detailed design (Theme/UML) and mappings to different aspect-oriented programming languages. Theme/UML is a MOF based extension of the UML version 1.3 beta R7 based on composition patterns. It provides a new type of classifier called a *theme*, a *composition relationship* and three integration strategies, *merge*, *override* and *bind*. These extensions enhance existing object-oriented design to support advanced separation of concerns.

3.4.1 Semantics

A *theme* is like a package and is a declaratively complete concern space that encapsulates the structure and behaviour of that concern. Declarative completeness means that the specification of the concern is self-contained and does not reference anything outside the space in which it is defined. An *aspect theme* is one that encapsulates a crosscutting concern and is designed relatively to the abstract templates with sequence diagrams specifying when and how the templates interact with the base themes. A *base theme* is used to design non-crosscutting concerns. Theme/UML supports two different composition mechanisms. Based in the subject-oriented paradigm, the first includes the composition of overlapping or shared concepts. Theme/UML allows a composition specification to be defined to allow these shared concepts to be matched and reconciled. Figure 3.1 shows an example of a merge composition relationship between two themes. A `merge` can exist between two or more themes. The `match[name]` indicates that the elements in the themes are to be matched by name. The `ThemeName` indicates what the name of the theme that results from the merge will be. The `resolve` specifies reconciliation strategies if a collision occurs during a merge. In this diagram, if two attributes match but have

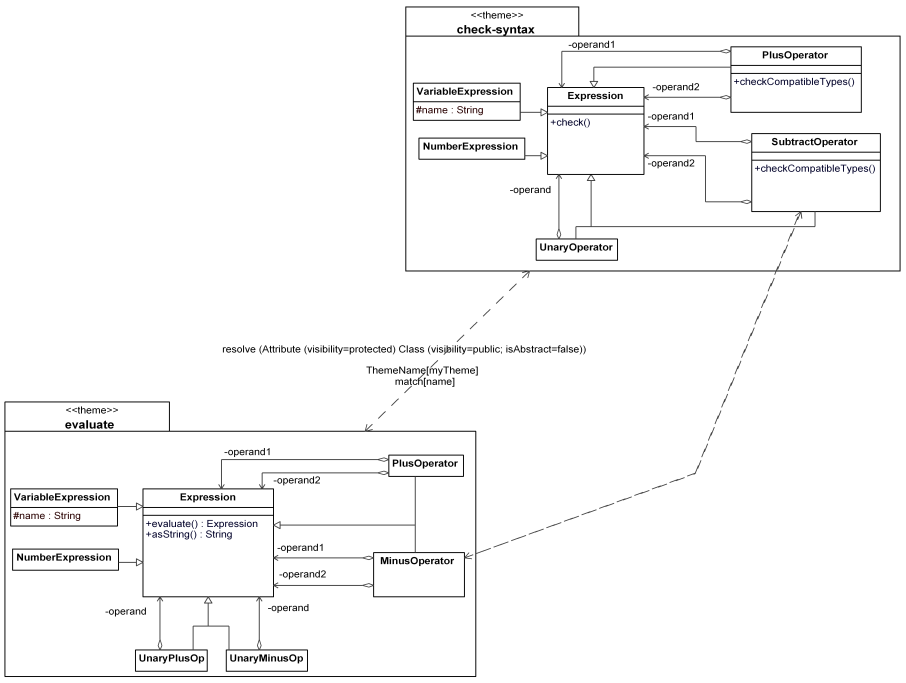


Figure 3.1: Example of Theme/UML merge

different `visibility` the merged result will have a `visibility` of `protected`. Likewise, if a two classes match the default for `visibility` is `public` and `isAbstract` is `false`. The composiiton relationship between the elements `MinusOperator` and `SubtractOperator` indicate that these elements match. Theme/UML also has other support for reconciliation not illustrated here. For example, it is possible to give an `explicit` element a reconciliation if the collision is anticipated. It is also possible to give themes priority, to allow a theme's specification to have a priority over others. Theme/UML also supports an override composition relationship that allows one theme's specification to override the specification of another. The second mechanism for composition allows aspect themes to be merged into base themes. This composition relationship is called a `bind`. Figure 3.2 gives an example of an aspect theme `logging` being bound to a base theme `check-syntax`. On the aspect theme the `loggedOp()` operation is the template that triggers the behaviour in the aspect. This behaviour is illustrated by the sequence diagram. The `_do_loggedOp()` operation indicates that this is the behaviour of the base theme. The sequence diagram is important in representing how and when the crosscutting behaviour is executed with respect to the base themes it is crosscutting. The `bind` represents the instantiation of the theme. The operation `Expression.check()` is now the operation that is being bound to and the template operation is replaced with this upon the aspect's instantiation.

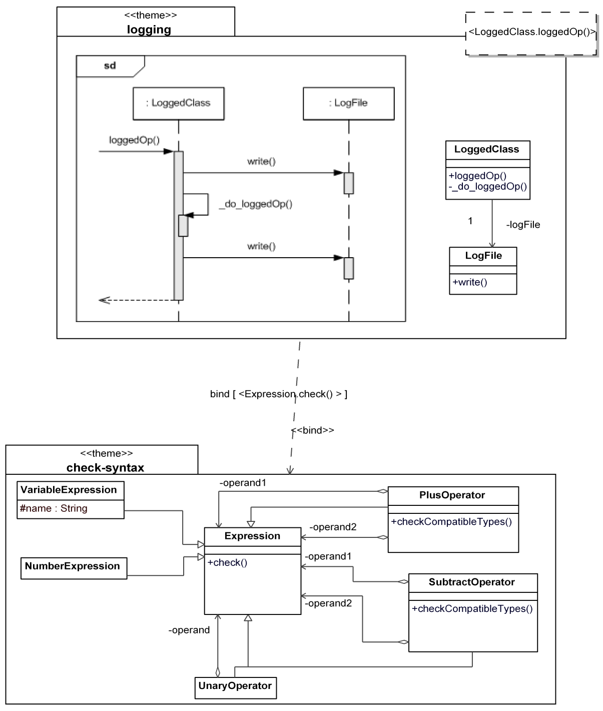


Figure 3.2: Example of Theme/UML bind

3.5 Summary

This chapter gave an introduction to AOSD and introduced some of the concepts. The two main aspect approaches, asymmetric and symmetric paradigms were discussed. The terminology for AspectJ and the Subject-Oriented paradigm were discussed to lay a foundation for an explanation of Theme/UML, the chosen AOM used in this dissertation.

Chapter 4

State of the Art

4.1 Introduction

This chapter gives a review of the state of the art in model-driven aspect-oriented weavers. The insights gained from this review will be used to create a model-driven aspect-oriented weaver that is suitable for the domain of mobile, context-aware computing.

4.2 AMW

The Atlas Model Weave (AMW)[1] is a tool that allows creating links between models [15]. It is based on the Eclipse Modelling Framework (EMF) and is part of the ATLAS Model Management Architecture (AMMA). The links are stored in a weaving model, conforming to a weaving metamodel. This tool can be used to support weaving aspects, although it is not centered specifically around the notion of aspect-orientation. ATL is which is a transformation language and part of the AMMA is used to weave once the links are established.

4.2.1 Weaving Capabilities

To support weaving links between models, AMW defines a metamodel for weaving. It allows models to be visualised in a tree-like way and to associate links between two metamodels or models using the weaving metamodel. It defines the notion of a weaving session in which the weaving metamodel, the models and their metamodels are loaded and the links are defined and woven. Their justification for this approach is that weaving is a purely human task and may be guided by heuristics. Figure 4.1 illustrates their weaving metamodel. `WElement` is the base element and is used to extend other elements through inheritance. `WLink` is used to link them with `WLinkEnd`. Each `WLinkEnd` references a `WElementRef`. `WModel` represents the root elements for all other elements. Aspect-Oriented extensions

[1]http://www.eclipse.org/gmt/amw

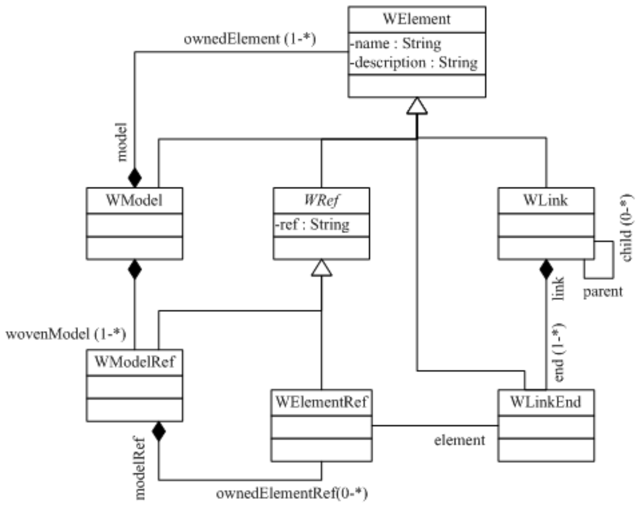


Figure 4.1: AMW Weaving MetaModel

can be defined in terms of this metamodel[2]. The elements that are woven can be defined as extensions to `WElementRef`. The weaving process can be defined in terms of `WLink` elements, and `WModel` allows many modes to be woven. The weaving process does not distinguish between primary and aspect models, therefore it is purely symmetric.

4.2.2 Critique

Although AMW does support aspect-orientation through its weaving metamodel, there is no official publications demonstrating or evaluating the approach for its integration with MDA. It is hard to judge from the usecase how the approach is used to model and design in an aspect-oriented way. Moreover, the graphical extensions is done manually with the tools and it appears to only support establishing links with two models at any one time. There is no support for composition of models at design time or how advice is applied relative to the behaviour of a system.

4.3 C-SAW

Constraint-Specification Aspect Weaver (C-SAW) [20] is a framework that supports the creation of aspect-oriented domain-specific weavers. It is integrated as part of the Generic Modeling Environment (GME)[3], a configurable toolkit for creating domain-specific modelling and program synthesis environments. GME's metamodelling language is similiar to the notation of the UML class diagrams and OCL constraints.

[2]http://www.eclipse.org/gmt/usecases/AOM

[3]http://www.isis.vanderbilt.edu/projects/gme/

4.3.1 Weaving Capabilities

C-SAW supports the creation of new domain weavers by integrating domain-specific strategies into a meta-weaver framework. Strategies support heuristics for a particular modelling paradigm and are specified in a DSL called the Embedded Constraint Language (ECL). A generator translates and instantiates a meta-weaver framework from the strategies and a domain-specific weaver is created. This domain-specific weaver can then be used to model pointcuts. ECL is an extension and subset of the OMG OCL and allows the weaver designer to specify the pointcuts in this language. It is declarative and supports navigating the hierarchical structure of models to specify pointcuts. It also uses introspective operators and reflective information to obtain more precise pointcuts. For domain-specific weavers, pointcuts are specified in ECL with strategies describing how a concern is applied with relation to the node in the model.

4.3.2 Critique

The approach emphasises the creation of domain-specific weavers by combinding AOSD and Model-Integrated Computing (MIC). Although, GME does not implement the MDA standards it has very similar concepts and is similiar to these standards, providing its own metamodel and support for OCL-like constraints. However, a designer is still tied to these environments rather than a set of evolving standards provided by the OMG. The support for pointcut selection is very powerful as it is based in OCL. It appears that these pointcut expressions are text based but suggests integrating them into the modelling environment for future work. Although the strategies determine how the advice is applied, these seem to be implemented in C++, rather than the modelling environment.

4.4 Model Composition Directives

Composition Directives [52] [47] is an approach that supports the composition of both aspect and base models with each other in the UML. In this way, it uses a hybrid symmetry approach for merging i.e. the composition procedure does not distinguish between an aspect and a base model. This approach was designed to deal with inadequacies associated with simple name-based matching. For example, two operations to be merged may have the same name but different argument lists or return values but according to a simple name-based matching strategy it would match. This is an unintended side-affect as they should not match. This approach supports different elements having unique matching strategies, according to their syntactic properties and for what makes sense for that element. To support this a composition metamodel is created. Composition directives can also be applied to resolve inconsistencies with the composition procedure.

4.4.1 Weaving Capabilities

The composition procedure (c.f. Figure 4.2) is illustrated by this activity diagram taken from [47]. Taking the primary model, the instantiated aspect models and the composition directives, the tool

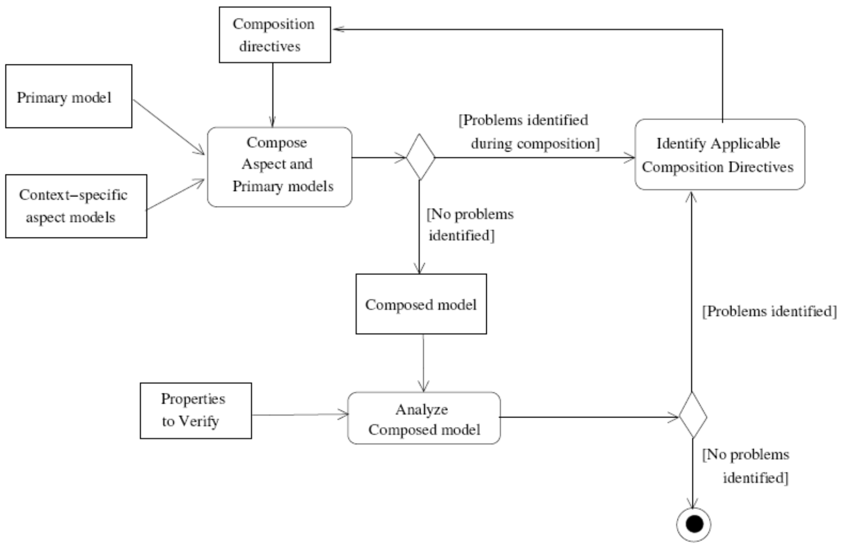


Figure 4.2: Model Composition Directives Procedure

composes them. If there is problems identified during composition, composition directives are applied to help resolve them. The models are then re-merged and checked again until the composition result is as intended. The composed model is then analysed. To define the behaviour of the composition,

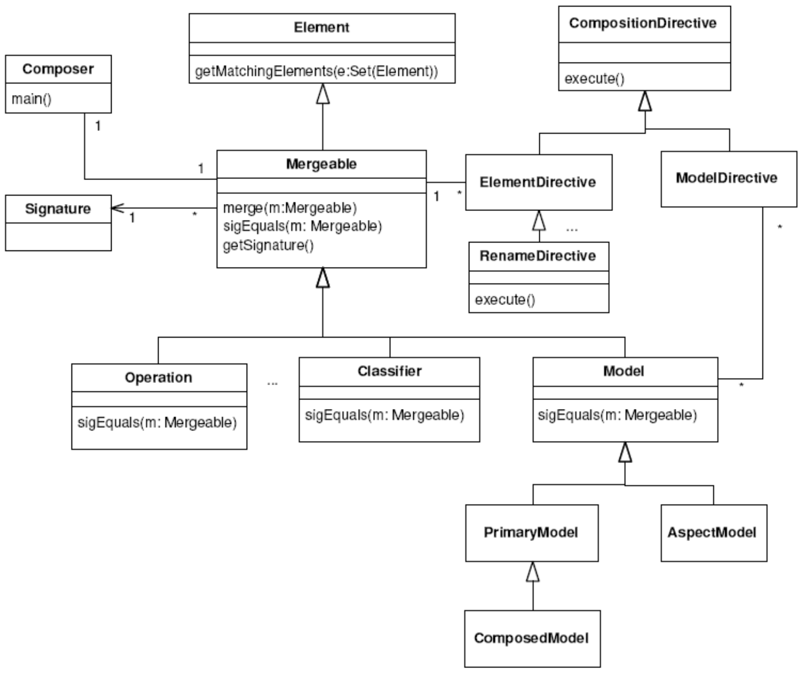


Figure 4.3: Model Composition Directives MetaModel

a composition metamodel was created. The metamodel was implemented using Kermeta[4]. Kermeta extends the EMOF with an action language. This allows one to define the behaviour of operations in a metaclass. Figure 4.3 shows the metamodel. The metamodel extends the elements of the UML metamodel with composition behaviour. As illustrated, the `sigEquals()` operation defines the signature-based matching criterion for each element individual. The merge is then done recursively. Merge is called on each element and in return the child elements are called too. The composition directives defined in the metamodel include, creating, moving, removing, changing, replacing and overriding model elements.

4.4.2 Critique

The combined approach of extending the UML metamodel with composition behaviour and directives seems very powerful. Each UML element can have its own matching strategy, which is claimed is better over other approaches that use matching on name only. The composition metamodel seems like a good idea. However, it appears to extend the UML metamodel at M2 (c.f. Figure 2.2 and as such, is deemed a heavyweight extension. This means an existing UML 2 tool would need to be re-factored to accomodate with the extra extensions in the metamodel. The composition directives appear to be textual based. It does not suggest any graphical tool that would aid the user in applying the composition directives directly at the model level.

4.5 WEAVR

The Motorola WEAVR [53] is an add-in to the Telelogic TAU tool and is designed for use in the telecom system engineering industry. It is a translation-oriented approach that includes a joinpoint model for state machines. It uses the Specification and Description Languages (SDL) and UML standards to fully model reactive discrete systems and produce executable code. Unlike other approaches, it uses transition-oriented state machines and an action language to specify fully the application logic at the model level.

4.5.1 Weaving Capabilities

The add-in includes a profile to allow developers to define Aspects in the UML and provides a means to visualise the joinpoints and the effects of applying them throughout the model. Furthermore, it allows aspects to be validated at the model level. It also performs full aspect weaving at the model level, allowing simulation of the behaviour with aspects. To illustrate an aspect, a class is extended with the `aspect` stereotype, allowing tagged definitions in the forms of attributes, operations, signal definitions and ports which are treated like inter-type declarations. It also allows the pointcuts and connectors to be specified using their relevant stereotypes. Connectors are bound to pointcuts which

[4]http://www.kermeta.org

are state machine based. It is also possible to use order the precedence of connectors applying to the same pointcut and in effect allows aspect interference to be handled by the designer [24].

4.5.2 Critique

This aspect-oriented model weaver seems to be the most mature at the moment. It provides a means to both structurally and behaviorally specify crosscutting. It supports the visualisation, analysis, simulation, weaving and execution of aspects at the model level. It also supports full transformation to code. However, the tool, Telelogic TAU, used to design the add-in for aspects is an expensive industrial tool, making it inaccessible for the researcher or average designer.

4.6 XWeave

XWeave [21] is a model weaver that supports the composition of different architectural viewpoints, supporting software product-line engineering and allowing for variable parts of architectural models to be woven according to a specific product configuration. It supports the weaving of any metamodel or model defined in Ecore, the Eclipse Modelling Framework's (EMF) meta-metamodel. XWeave uses a form of asymmetric AO.

4.6.1 Weaving Capabilities

Aspect models are woven into a base model using two strategies, name matching and explicit pointcut expressions. Name matching supports weaving through equivalence of elements in the base and aspect models if both have the same name and type. Pointcuts are defined according to the oAW[5] expression language and are external to both aspect and base models. This expression language is statically typed and is based on OCL, supporting path expressions, collections and selection/filtering. Path expressions uses the dot-notation to navigate over child elements. Collections provide the common set operations such as union and intersection and allow for predicate queries. Selection allows an expression to be queried and filtered. XWeave distinguishes between homogeneous and hetereogeneous aspect models. A homogenous aspect is one that applies the same piece of advice to several places, while a heterogenous aspect adds different pieces of advice to different places.

4.6.2 Critique

As discussed in their evaluation, their support for advice is limited. Base model elements cannot be removed, changed or overridden by aspect models and hence only supports additive weaving. Support is also only available for weaving asymmetrically. It also appears to only support structural weaving, with no support for how and when the advice is applied. This is important for capturing stateful aspects. The aspect structure is copied into the base models according to the pointcuts. It also doesn't appear to have graphical support for indicating the pointcuts, it seems to be text based.

[5]http://www.openarchitectureware.org

However, as its expression language is OCL-like, its support for the definition of pointcuts appears to be very powerful and flexible.

4.7 Summary

This chapter gave an overview of the state of the art in model-driven aspect-oriented weavers. Each tool was discussed in regard to its weaving capabilities and a critique was given.

Chapter 5

Design

5.1 Introduction

This chapter describes the design of the proposed approach. The first section discusses the initial design decisions. The second section gives an overview of the whole process. The rest of the chapter is divided into three main sections, detailing each phase of the process - the modelling phase, the composition phase and the transformation phase.

5.2 Initial Design Decisions

As discussed in the introduction, Theme/UML was defined before the evolution of model-driven concepts, standards and technologies. As such, the definition of Theme/UML was not fully compatible or suitable with current standards or conventions. One of the contributions of this dissertation was to implement Theme/UML as to be in-line with the MDA vision. The first objective was to investigate the different MDA strategies that may be suitable as a means to integrate Theme/UML. The later sections will then discuss the challenges and design of the chosen strategy. Three alternative suitable strategies evolved over time. This section will discuss the alternative strategies, highlighting the advantages and disadvantages of each and why the third strategy proved to be more suitable than the other two.

5.2.1 Strategy 1 : Heavyweight Extension

A *Heavyweight Extension* is a term that defines extending the UML by modifying the underlying UML metamodel (MOF) at M2 (c.f. Figure 2.2). Theme/UML was originally defined as a heavyweight extension [9] in the UML 1.3 beta R7[1].

The heavyweight extension mechanism in the UML 2.1 involves reusing part of the `InfrastructureLibrary` package and augmenting it with appropriate metaclasses and

[1]ftp://ftp.omg.org/pub/docs/ad/99-06-03.pdf

metarelationships[1]. An initial attempt was made to investigate how the syntax and semantics of the composition relationship extensions could be ported over to the current UML version. It proved quite challenging as the two standards were largely incompatible with each other. While sharing some of the same concepts, such as `Package`, `Operation` and `Classifier`, the underlying metamodels (and indeed the meta-metamodels) that defined these constructs meant that while named the same and conceptually meant to be used in the same way, they were semantically different.

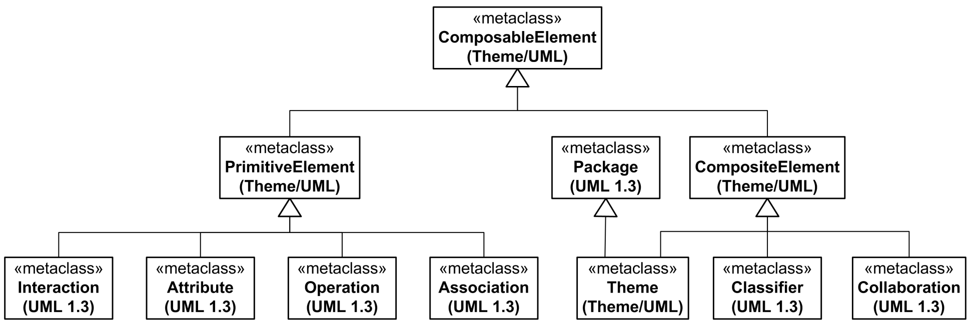


Figure 5.1: Theme/UML Composable Elements

For example, illustrated in Figure 5.1 the `ComposableElement`, `PrimitiveElement`, `CompositeElement` and `Theme` metaclasses are used to indicate which existing UML metaclasses are extended for composition. The goal is to incorporate these new metaclasses into metaclass hierarchy of the new UML 2.1.1 metamodel. The first challenge can be observed while trying to integrate with the `Classifier` metaclass. In UML 2.1 (c.f. Figure 5.3) the definition of a `Classifer` is different from a `Classifier` in UML 1.3 (c.f. Figure 5.2). The metaclass hierarchy and metarelationships between them are completely different. Even while disregarding the semantic foundation on which the metaclasses are defined and assuming that a `Classifier` in both metamodels are conceptually the same, more challenges were encountered. An `Association` in UML 2.1 has been changed to be a `Classifier`. As illustrated in Figure 5.1, an `Association` is defined in Theme/UML to be a `PrimitiveElement` while a `Classifier` is defined as a `CompositeElement`. Attempting to use the same definition of Theme/UML with the current UML standard, an `Association` in UML 2.1 would then be a `PrimitiveElement` as well as a `CompositeElement`, which is not the original intended semantics. Likewise, the element definition `Attribute` in UML 1.3 no longer exists in UML 2.1 and has been updated to the concept of a `Property`. A `Property` is not directly analogous to an `Attribute` either, as it can also be used to represent `Association` ends.

This example was only used as an illustration of the difficulties involved with updating Theme/UML for heavyweight extension. The rest of the extensions would have to be updated in a similiar way, with similiar effort. These difficulties meant that Theme/UML would have to be redefined completely from scratch using UML 2.1 and the intention of the original semantics.

[1]http://www.omg.org/docs/ptc/03-09-15.pdf

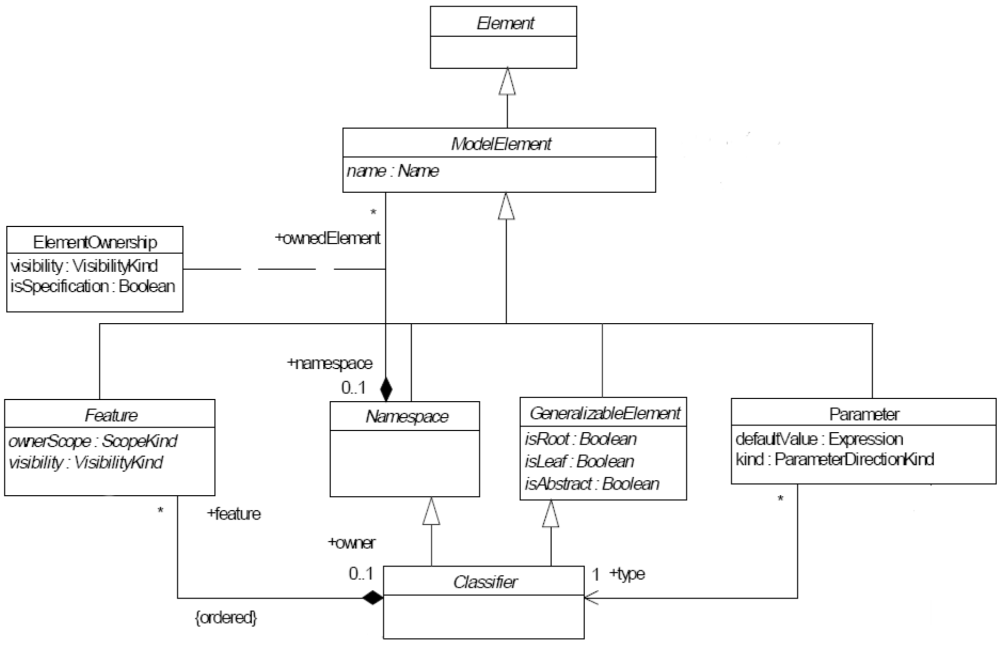


Figure 5.2: UML 1.3 Hierarchy for Classifier

Tool support was also another major disadvantage for choosing this strategy. The *compliance levels* set forth in the UML standard does not mention this type of extension to be a requirement for vendors who wish to implement the standard. In effect, these invasive changes that deviation from the standard, make existing tools inoperable for a user wishing to use heavyweight extension, as well as a developer that wishes to implement these extensions.

5.2.2 Strategy 2 : Lightweight UML Extension with UML Package Merge

Ruling out the option of a heavyweight extension, which is a first-class extension mechanism as explained in the previous subsection, a lightweight extension is a mechanism that is in-built in the UML and is called a profile. A profile is a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular method or platform.

One of the changes with the adoption of the UML 2 standard was the introduction of a Package Merge.

> A package merge defines how the contents of one package are extended by the contents of another package.

As a `Theme` is an extension of an UML Package (c.f. Figure 5.1), it is worth investigating if this merge can be redefined or replaced in terms of the existing Theme/UML merge. In [27], an investigation was made to redefine Theme/UML's integration merge strategy in terms of Package Merge. This

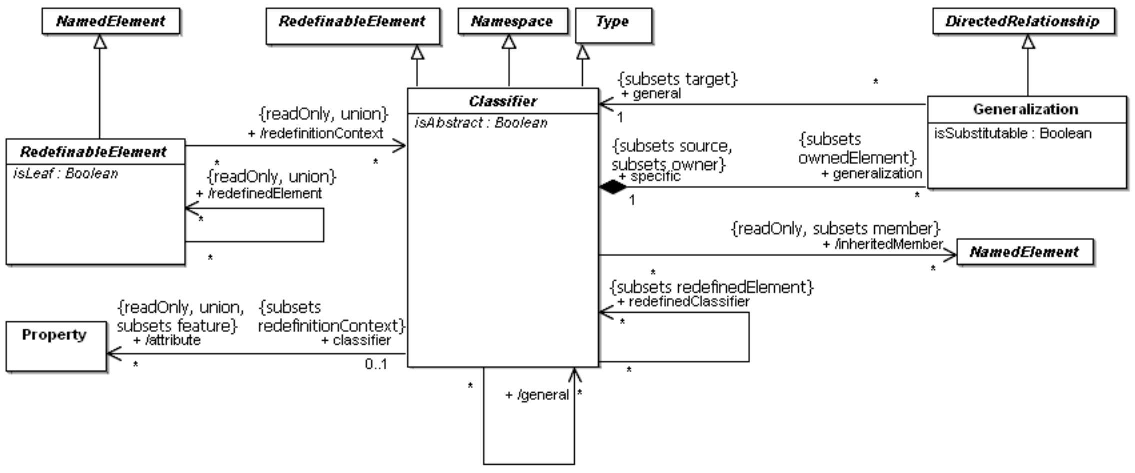


Figure 5.3: Metaclass Classifier for UML 2.1

work outlines both the differences and similarities between Theme/UML and Package Merge. The original intention was assumed to extend Package Merge at the metamodel level to accommodate the unification of Theme/UML composition semantics. This type of merge was implemented in [28] to support execution and testing of themes. However, it uses kermeta, which is a tool that adds executability at the M2 layer (c.f. Figure 2.2), making any addition a heavy weight extension. As discussed in the last subsection, heavyweight extensions have been ruled out for this approach. However, Package Merge still needs to evaluated to decide if it is suitable as a replacement at the M1 layer.

The difficulty with comparing Theme/UML with Package Merge is that they each address different problems. Package Merge was designed to accommodate the interoperability of tools by allowing a higher level of compliance to be merged with a lower level one [57]. Theme/UML's merge was developed in order to accommodate the composition of overlapping concepts in base themes. As discussed in the previous subsection, both metamodels have different structural and semantic foundations, making comparison possible only on a conceptual level. For example, an Association in UML 1.3 is conceptually similar than an Association in UML 2.0 but the semantic and structural differences make comparing them inane. Rather than attempt to explicitly illustrate such differences in order to do a comparison (and extend the efforts of [27]), an evaluation shall be performed against a set of criterion for the intended application of the upgraded Theme/UML in the context of MDA. In this way, both Package Merge and the existing Theme/UML merge can be evaluated against these criteria in order to investigate if Package Merge is a suitable alternative.

To be implemented as a transformation, the merge algorithm must to be correct, well-formed and complete. Since Theme/UML's composition semantics relies heavily on the merge algorithm, a fault in it has the possibility to render composition incorrect, consequently having an adverse chain reaction on all further transformations. Another requirement is that the algorithm should be flexible enough to accommodate the extension of existing or new elements for composition. For example, currently Theme/UML's merge integration does not allow explicit behavioural merging of sequence diagrams [28] but if this extension is required at a later date, both the the merge algorithm and the ability to add new composition elements should be available.

The basis for this evaluation is a comparison against model merge methodologies. Model merging techniques may be decomposed into four distinct phases; comparison, conformance checking, merging and reconciliation [33][45]. Package Merge and the Theme/UML shall be compared in these phases.

Comparison Phase

The comparison phase of a merge algorithm determines the equivalence of elements based on semantics of the merge.

In Theme/UML two types of comparisons are possible to match elements, `matchByName` and `dontMatch`. As this is an enumeration, additional matching strategies can be defined and attached to the merge making it extensible. Theme/UML also defines placing these relationships explicitly between elements in a merge. This allows different elements to be matched against different matching strategies in the same merge. Theme/UML also matches based on composite and primitive elements.

Package Merge, as a static specification does not allow any customisability in terms of comparison. The result of merging two elements with matching names and meta-types that are exact copies of each other is the receiving element. There is no way to define or attach matching strategies to elements. Any extension would have to be done at the UML metamodel level, which has been ruled out as infeasible.

Conformance Checking Phase

After the comparison phase, the matching elements are examined to determine if they conform to each other. In this phase, a check is done to see if the elements are of conforming types. This phase is irrelevant to both Theme/UML and Package Merge as both match based on type and/or name and their constraints. Any elements that are not matched are assumed to be of the same meta-model and are deep copied.

Merging Phase

The merging phase is the algorithm used to merge the matching elements. A merging algorithm must be extensible and flexible enough to handle the addition of new meta-model types.

Theme/UML supports the addition of new composition types through the extension of the primitive and composite hierarchies. It is also flexible enough to allow a new merge to be redefined or sub classed using the Integration abstract meta-class as a basis.

Package Merge, as a static specification, does not offer the same flexibility or extensibility. Although it states, that the default deep copied rule can be superseded for specific meta-types through profiles, it fails to demonstrate how this is possible. Furthermore, UML internals rely heavily on the specific implementation of this merge. Even if a heavyweight extension was feasible, these redefinitions of Package Merge may have unforeseen ramifications on the rest of the metamodel that depend on the current static specification of Package Merge [57]. The merging rules are defined by a set of transformations on specific meta-types. Other meta-types are not supported as they are considered "complex and domain specific".

Reconciliation

The reconciliation phase allows conflicts in the merge model to be resolved or fixed.

Theme/UML has very extensible reconciliation strategies in order to resolve the conflicts needed in overlapping design elements. To accommodate this requirement, Theme/UML specifies a number of features. Firstly, reconciliation can have precedence. This allows different themes to merge according to their precedence. Explicit Values allow a designer to resolve conflicts on specific conflicting elements. Default Values allow a designer to specify a default merge if no existing reconciliation rules exist. Each of these features can be attached based on the designers needs, allowing a very extensible reconciliation specification.

Package Merge does not have any reconciliation features.

Summary

One of the original requirements for the integration of Theme/UML into UML 2.1 was that the original composition specification was at least as good as it had previously been defined. In its evaluation, Package Merge does not appear to be suitable as a replacement for the original Theme/UML merge. It lacks the necessary features. Using Package Merge as a foundation for merge is also unsuitable for several reasons. In [27], it is suggested that the extra reconciliation, merging and comparison features in these phases be transformed to a suitable medium for Package Merge. However, as Package Merge merge algorithm lacks the ability to support additional meta-types [57]; it is unsuitable based within the context of these requirements. The final requirement is that the merge algorithm be well-formed and correct. The existing Theme/UML documents well-defined rules in its specification. Although Package Merge appears to be well defined in the UML super specification, research has proven that it is not yet ready for use by either UML users or meta-model builders [40].

5.2.3 Strategy 3 : Lightweight UML Extension with Composition Metamodel

This strategy, like the previous subsection, uses profiling to indicate how elements are marked for composition. Ruling out PackageMerge as a foundation for the definition of the composition semantics, the final strategy is to define a separate *composition metamodel* in MOF to define Theme/UML's composition semantics. The process involves marking a model to indicate the composition, and then mapping this into an instance of the composition metamodel. The composition metamodel defines the structure and behaviour of Theme/UML's integration strategies.

The first advantage is the distinct separation of the graphical extensions in the UML and the definition of the composition semantics. The UML is a general purpose modelling language and is not suitable to be used in defining transformations on itself. This was conveyed with PackageMerge in which the semantics of the transformations itself could only be expressed in textual form. The second advantage is that a composition metamodel can better convey the structure and behaviour of a transformation (in this case a horizontal transformation) in a more well-defined formal manner.

5.3 The Process

This section gives an overview of how the process works from the perspective of a software designer that wishes to develop a mobile, context-aware application. The three main sections of this chapter gives a more in-depth explanation of each phase and the design decisions made.

The process is illustrated in Figure 5.4. The three phases are represented by swim-lines in the activity diagram. The first step, the designer(s) begins by modelling the mobile, context-aware application in Theme/UML. The domain-specific context-aware concerns that exhibit crosscutting behaviour can be separated and designed independently from the main application concerns. The fork in the activity diagram emphasises that these activities can be done both orthogonally and concurrently with each other. After all these concerns are designed, represented by the join in the activity diagram, the designer must then apply the composition relationships. This means that the designer can specify how the context-aware concerns are to be composed with the application concerns. Furthermore, if the application concerns were further decomposed into multiple themes, composition relationships can also be applied to these.

The second step is the composition phase in which the designer uses the tool to automatically compose the model. Composition of the model to an object-oriented representation is required because the target environments are object-oriented. After viewing the model, the designer can then choose to do one of three things, illustrated by the decision point in the activity diagram. This first choice allows the designer to choose to go back to phase one if the composition relationships need to be re-applied or adjusted because the composed model was incorrect or incomplete. The second choice available to the designer is to go back to the start of phase one and edit the model again. Otherwise, the designer can choose that the composed model is complete and move on. Before moving to phase three, the designer may need to re-factor the composed model. Re-factoring is necessary at this stage, due to semantic ambiguities that may occur while merging with the assumption of disjoint class hierarchies. These ambiguities emerge as a result of the composition process and need human intervention to reconcile. This will be further explained in a later section.

The third step, the designer can choose to select a mapping of a technology for transformation. Two mappings are possible in this approach, as a proof-of-concept, are J2ME and .NET CF, but choosing any pre-existing mapping is possible. Using the mapping and the model, the tool adds in the platform-specific details and produces a PSM. This PSM is open for re-factoring if the designer wishes to add additional platform-specific structure and behaviour. As discussed in Chapter 2, an

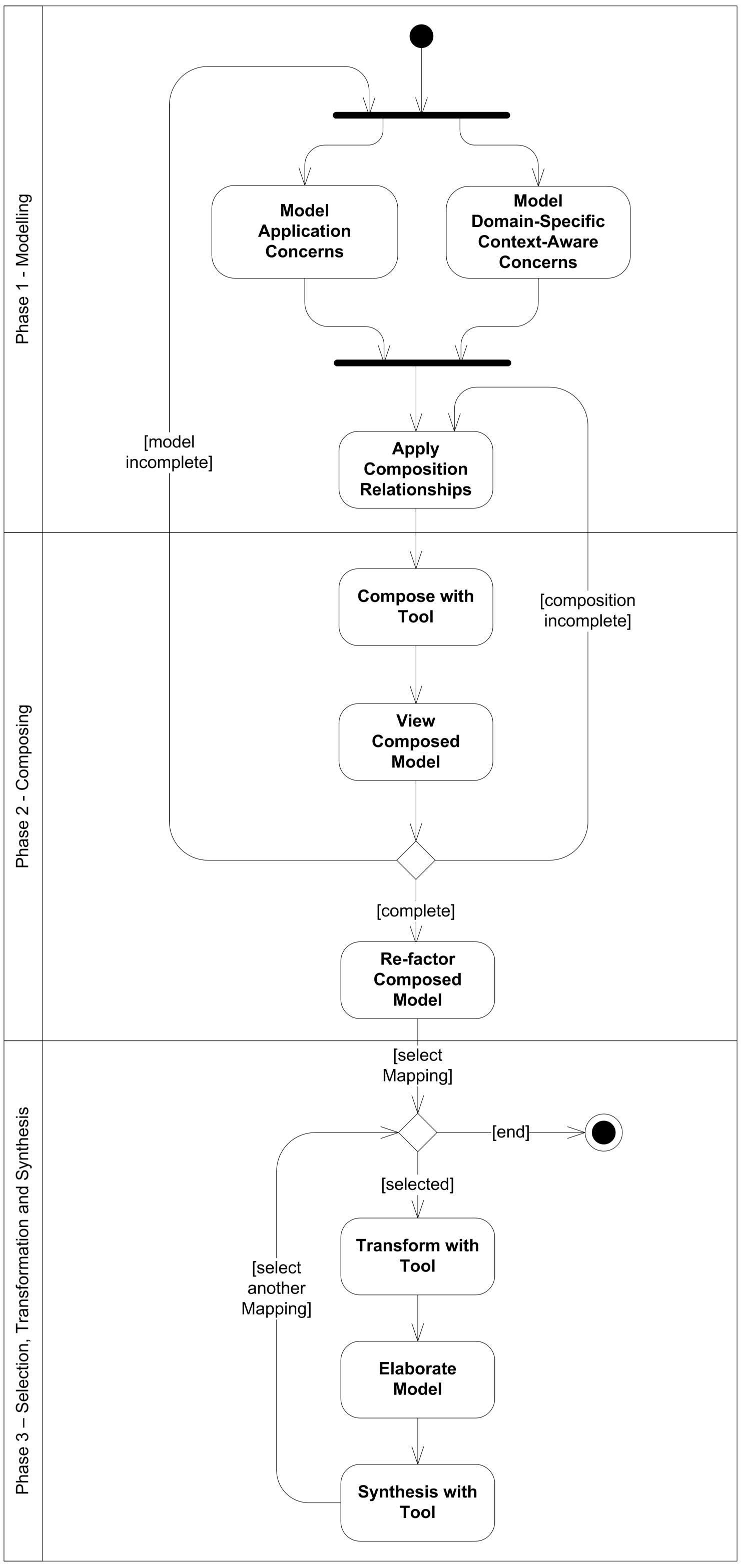


Figure 5.4: The Process

elaborational approach was chosen. This means that the PIM is **not** computationally complete and the PSM needs to be open for modelling after its transformation. After this, the PSM can be synthesised into code. The designer can then choose another platform and repeat phase 3.

5.4 Phase One - Modelling

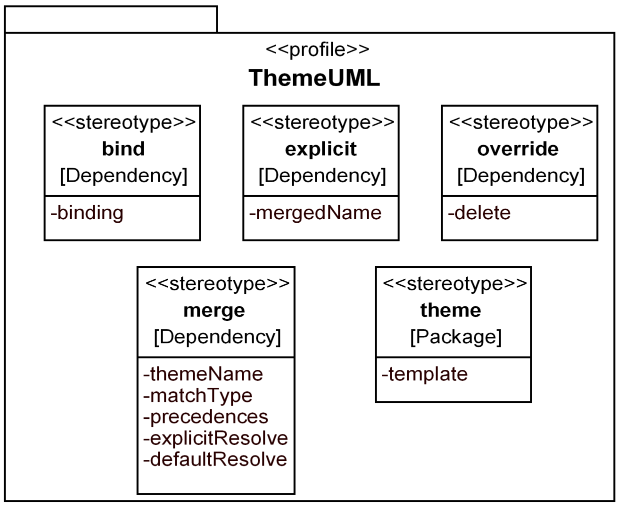


Figure 5.5: Theme/UML Marking Profile

Using a standard UML 2.1 compatible tool, the designer imports the Theme/UML marking profile. As explained in Chapter 2, a profile is a lightweight extension of the UML. Marking is a technique that allows elements in an UML model to be indicated in a non-invasive way to allow the specification of a transformation. With this marking profile, the designer can model according to the Theme/UML guidelines [10], using the marks to indicate how the themes and `PackageableElements` within are to be composed.

5.4.1 Modelling Concerns

Before the designer begins modelling, it is necessary and assumed that the requirements of the application have been correctly analysed and a number of concerns have been decomposed to be compatible with Theme/UML. Although the MDA Guide [37] does not explicitly indicate transformations from a CIM to a PIM, it does indicate that those artefacts modelled in a CIM be traceable to their PIM and PSM artefact realisations, which suggests some kind of mapping. Whether this mapping is realised as a manual transformation, indicated by completely elaborating the PIM, or an (semi-)automatic transformation, where some artefacts are generated in the PIM, such as the that presented in [48] is not pertinent to the outlined approach. In the later case, the designer would simply begin with a set of UML artefacts that have been pre-generated and then further elaborate and re-factor this.

The modelling phase as indicated by Figure 5.4 illustrates the two activities used to model both the application concerns and the domain-specific mobile, context-aware concerns. The application concerns are modelling using base themes and the context-aware concerns are modelling using aspect themes. Theme/UML's support for symmetric decomposition and declarative complete concern spaces allow for these two activities to be done both concurrently and independently from each other. Furthermore, each theme can be designed in isolation, either by an individual or a team of designers. Each theme must be modelled inside a `Package` and must not reference any element outside that package. This ensures that the concern space is declaratively complete. The `Package` must have the stereotype `theme` applied from the Theme/UML Profile. As aspect themes are modelled relative to their abstract templates, it is necessary for the designer to indicate this using the tagged value `template` from the Theme/UML Profile. Each theme must have a sequence diagram for each sequence of templates. This sequence diagram illustrates the interaction of the templates with the behaviour of the theme itself.

UML 2.1 Gates

In Theme/UML, the initial triggering message and returning message of a sequence diagram that illustrated a crosscutting specification were unconnected to any lifeline at its origin and destination respectively. Figure 5.6 illustrates how a crosscutting sequence diagram was designed with Theme/UML. The `loggedOp()` operation is not connected. Likewise the returning message is not connected. In the context of MDA, all UML models must be valid according to the UML

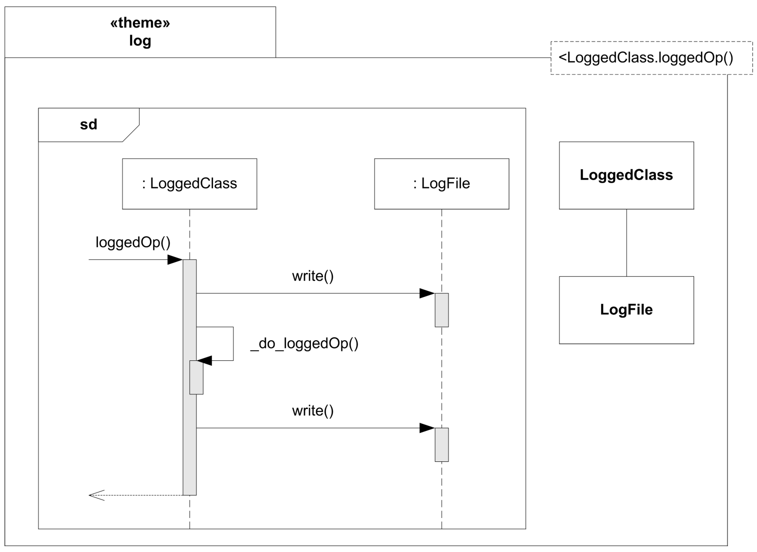


Figure 5.6: UML 1.3 Sequence Diagram

specification to ensure correct transformation. Messages cannot come or go from nowhere, therefore it is necessary to come up with an alternate solution. In UML 2.1, there is a notion of a gate.

> A Gate is a connection point for relating a Message outside an InteractionFragment with a Message inside the InteractionFragment.

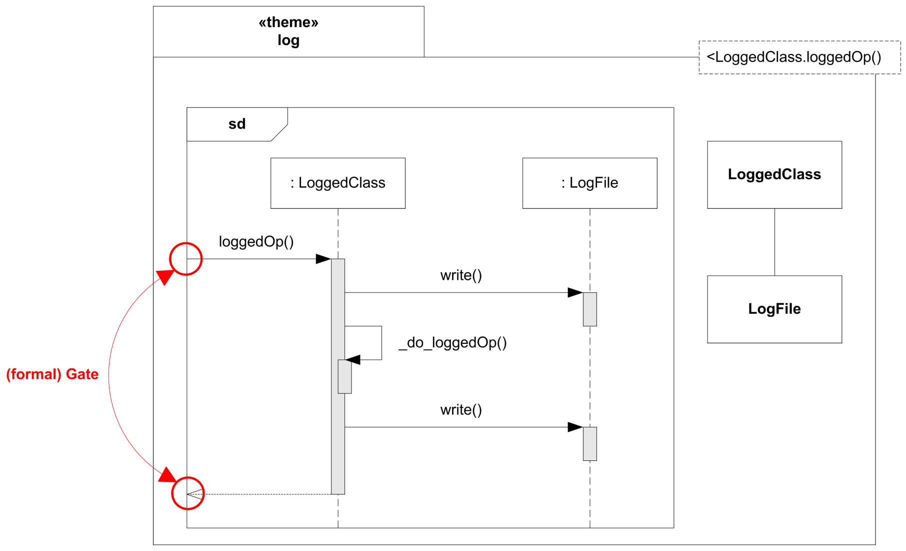


Figure 5.7: UML 2.1 Sequence Diagram with Gates

A gate is a new feature of the UML 2.0. It was created to allow for the decomposition of large sequence diagrams into smaller fragments. The fragments could then be composed together with the help of gates. For this approach, it nicely solves the problem where the origin of the initial triggering message is unspecified. Figure 5.7 illustrates the updated semantics of Theme/UML with UML 2.1, where the triggering message `loggedOp()` and its returning message are connected to formal gates.

5.4.2 Applying Composition Relationships

After the themes have been modelled, the designer must apply the composition relationships to the themes, specifying how they are to be composed[2]. At the coarsest level of granularity, the individual themes themselves are marked for composition, but support is also available to indicate finer compositions that deviate from the composition specification of the composite container. Base themes use the `merge` stereotype, applied to a `Dependency` from the Theme/UML marking profile. The tagged values for the `merge` stereotype are `themeName`, `matchType`, `precedences`, `explicitResolve` and `defaultResolve`. The `themeName` stereotype indicates the name of the final composed theme. The `matchType` allows a matching strategy to be applied to the merge, with the `precedences` stereotypes indicating the ascending order of the merge and the `defaultResolve` and `explicitResolve` as the reconciliation options if a conflict arises. An `explicit` stereotype, applied to a `Dependency` explicitly

[2]Refer to [10] for a more detailed explanation of Theme/UML's composition relationships and their semantics

indicates a deviation from the default composition of a `merge`. The `bind` stereotype is used similarly to the merge but indicates how aspect themes are composed with the base themes. The composition of the aspect theme is indicated using a `binding` tagged value to show how the templates are instantiated to the elements of the base themes. Once the composition relationships have been applied the design can then proceed to phase two as indicated in Figure 5.4.

5.5 Phase Two - Composition

5.5.1 Composing

At phase two, the model is composed. The tool takes the UML model with the marks indicating the composition specification and generates a composition model that is an instance of the composition metamodel. The UML model is composed into a single package. The designer can then view this composed model.

5.5.2 Iterative strategies

After viewing the model, the designer can then choose to do one of three things, illustrated by the decision point in the activity diagram (c.f. Figure 5.4). This first choice allows the designer to choose to go back to phase one if the composition relationships need to be re-applied or adjusted because the composed model was incorrect or incomplete. The second choice available to the designer is to go back to the start of phase one and edit the model again. Otherwise, the designer can choose that the composed model is complete and move on. This iterative approach is important. Jackson et. al [27], discuss partitioning the AO design process along two dimensions, horizontal and vertical. The two main activities suggested are concern design and composition specification.

Three iterative strategies are suggested depending on the design methodology used (c.f. Figure 5.8). `Continual cumulative composition` indicates a per-iteration strategy where the composition specifications are designed at the end of the iterations. `Once off composition` indicates the composition specification be done in a separate iteration. `Hierarchical composition` emphasises the integration of composites in each separate iteration. The outlined approach conforms to all of these strategies, depending on the needs of the designer or design team.

5.5.3 Re-factoring the Composed Model

This subsection will demonstrate the reasons the composed model is open for re-factoring. Illustrated in Figure 5.9 is an example merge, taking a small subset of the expression evaluation system (EES) example [10]. The `define` theme illustrates the structural specification of the grammar needed to define expressions. The design has been pruned to one class hierarchy for clarity. In this example, a `BinaryOperator` is a type of `Expression` with two operands indicated by the aggregation relationships. The `MinusOperator` and `PlusOperator` classes inherit from `BinaryOperator` which inherits from `Expression`. The `check-syntax` is modelled from the perspective of providing the

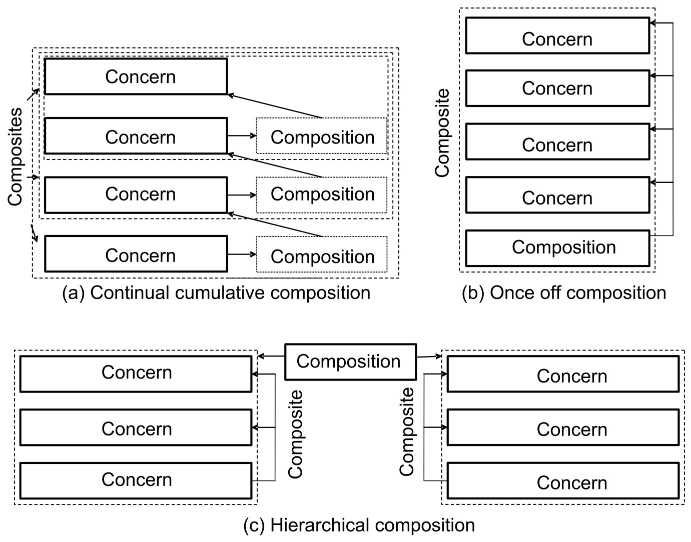


Figure 5.8: Iterative Strategies

functionality that ensures the syntax of the grammar is correct. Again, this theme has been pruned to one class hierarchy. The `Expression` class provides a `check()` method for this functionality. The `MinusOperator` and `PlusOperator` classes inherit from the `Expression` class and have aggregates `operand1` and `operand2` each with it. Note the distinct differences in the way the class hierarchies have been modelled. The `BinaryOperator` in the `define` theme is lacking from the `check-syntax` theme.

The ramifications of modelling class hierarchies disjointly in each theme leads to ambiguities that currently cannot be reconciled easily with Theme/UML constructs. The merged result is illustrated in Figure 5.9. Compared to the semantics of the Theme/UML merge, the result is as expected. However, the semantics of the design is in question. The first observation that questions the design is the redundant generalisations. An example of this is that `MinusOperator` inherits both from `BinaryOperator` and `Expression` and `BinaryOperator` inherits from `Expression` also. The obvious correction is to remove the generalisation from the most concrete to the most abstract if there exists one or more elements in between them in the class hierarchy. This is a structural problem that renders a UML model invalid and as such there is an algorithmic solution. This has been added into the composition process. The next observation that can be made is the redundant aggregations between the elements in the hierarchy. `BinaryOperator` has aggregations with `Expression` named `operand1` and `operand2`. Likewise the classes `MinusOperator` and `PlusOperator` also have `operand1` and `operand2` aggregations with `Expression.` The result is that `MinusOperator` has doubly two aggregations of the same kind. In this case, the solution is not as obvious. As the themes were originally designed as declaratively complete, assumptions cannot be made about naming conventions. This merge may be a perfectly acceptable design specification or the extra operands may need to be deleted manually. Either way, the decision is in the hands of the designer.

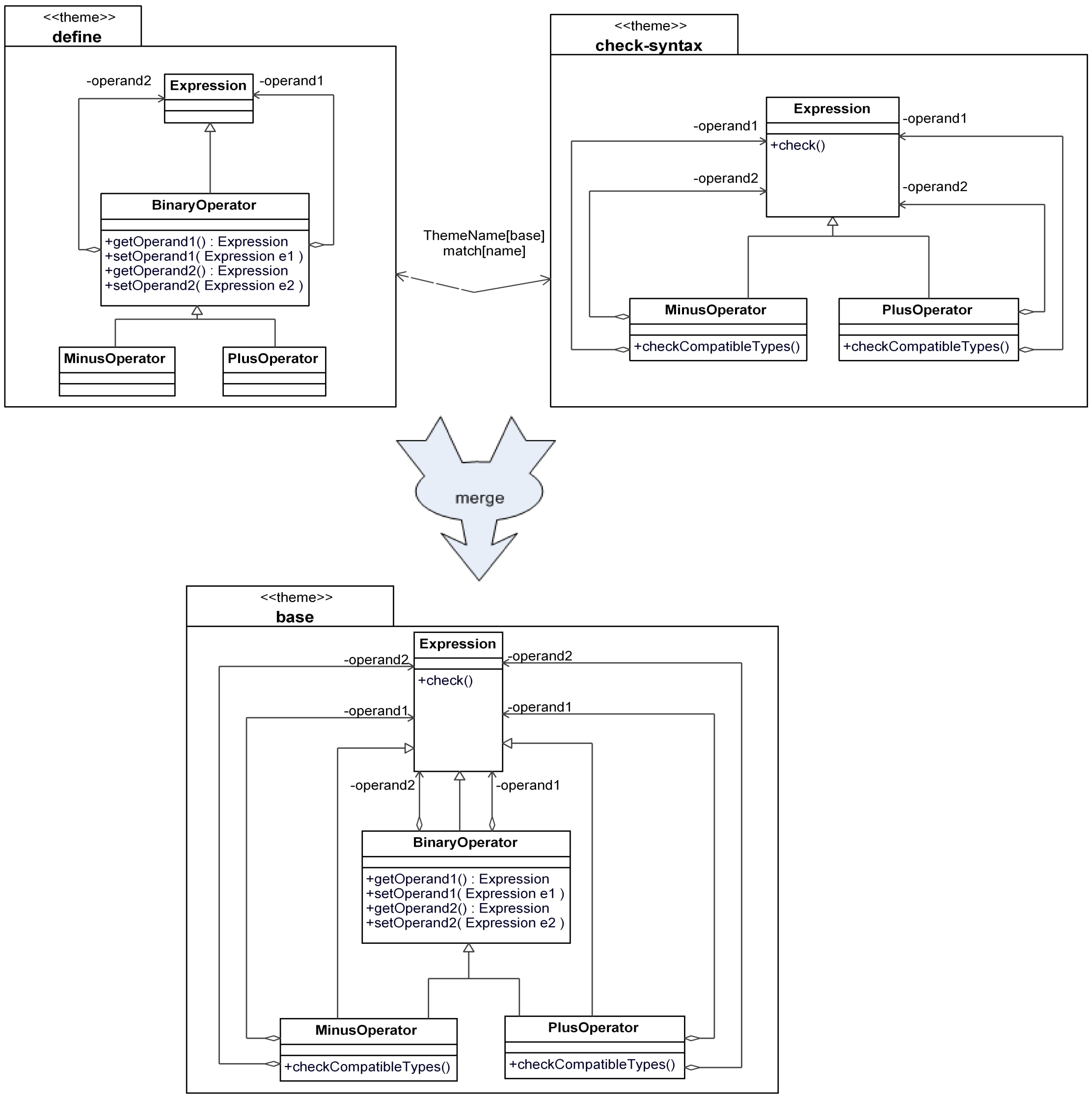


Figure 5.9: Re-factoring the Composed Model

Although not illustrated in this example, another reason the composed model is open for re-factoring is as a result of cycles in generalisations between classes as a result of the merge. This problem has been tackled in [55]. The proposed solution suggests a process in subject-oriented programming known as `flattening` [42]. The subject-oriented compositor flattens the class hierarchies and makes each class complete in itself. The solution involves removing the cycles from the flattened hierarchy then un-flattening. This solution is not implemented in this approach, but the designer can correct the problem if it emerges at this stage. However, the process of flattening is an interesting one in the context of MDA. One of the advantages of flattening is that it can cater for languages with different types of inheritance. UML supports multiple inheritance but the PIM design may not be suitable for PSMs that support single inheritance if it is used. For example C++ supports multiple inheritance while Java supports single-inheritance. Flattening would be an interesting transformation to integrate into a PSM transformation to re-adjust the class hierarchies depending on the OO capabilities of the language.

5.5.4 Design of Composition Metamodel

The composition metamodel describes the structure and behaviour of Theme/UML's composition semantics and is defined in MOF. Figure 5.10 shows the metaelements involved in the composition. These elements represent the elements of the UML metamodel that are used in the composition. The composition metamodel works on top of the UML 2.1. The composition classes are directly linked with their underlying UML 2.1 elements.

Matchable: This metaclass abstracts the notion of matching and comparing metaclasses. Metaclasses that realise `Matchable` need to provide their own matching strategy for that specific type.

- ***Matchable::matches(Matchable) : Boolean***: This operation takes as another `Matchable`. If the matching strategy provided by that element deems the two match it returns true.

ComposableElement: This metaclass realises the `Matchable` metaclass. It provides behaviour to query and set the properties of the UML element it is linked with through the MOF reflection facilities.

- ***ComposableElement::precedence : Integer***: This property is set as part of the precedence reconciliation. Each class in a composition specification has a specific precedence associated with it.

- ***ComposableElement::isComposite() : Boolean***: This operation indicates whether the element is a composite container. Each metaclass realising this class must define this behaviour. The semantics are directly linked with the UML 2.1 metaclass it represents. If this is a container than this returns true.

- ***ComposableElement::getProperty(Property) : ReflectiveSequence***: This operation retrieves the property of the instance of the UML 2.1 metaclass that the `ComposableElement` is linked with, through the MOF reflection facilities.

- ***ComposableElement::setProperty(Property, ReflectiveSequence)***: This operation sets the property of the instance of the UML 2.1 metaclass that the `ComposableElement` is linked with, through the MOF reflection facilities.

Theme: This metaclass is a composite and can not be contained by any other element. All `ComposableElements` must be contained inside a `Theme` (c.f. Figure 5.10).

- ***Theme::crosscutting : Boolean***: A `Theme` can be an aspect theme or a base theme. This property is set to true if it is an aspect theme.

Composable*: These metaclasses that directly realise the `ComposableElement` (c.f. Figure 5.10) metaclass are represent the elements that can be involved in a composition specification. These metaclasses are directly linked to their UML 2.1 classes when instantiated.

Figure 5.11 illustrates the structure and behaviour of Theme/UML's integration strategies and reconciliation properties.

Integration: This metaclass represents the abstract notion of an Integration Strategy for Theme/UML. An `Integration` must contain more or two themes.

- ***Integration::matchKind : MatchKind***: This property represents the matching specification that the integration strategy uses.

- ***Integration::integrate() : Theme***: This operation provides an abstract notion of an integration. The result of the operation provides an integrated Theme. The contents of the Theme is provided the metaclasses that realise this metaclass.

MatchKind: This enumeration represents the matching specification for an integration strategy.

- ***match***: This indicates that the elements are to be matched.

- ***dontMatch***: This indicates that the elements are not to be matched.

ThemeMerge: This metaclass represents an integration strategy on base themes. It is used to merge the models of different requirement specifications that were designed in separate base themes. `ThemeMerge` can contain a number of `Reconciliations` that take place during the merge.
Reconciliation: This metaclass represents the abstract notion of a reconciliation specification. A reconciliation can be specified on a `ThemeMerge` if conflicts arise in different themes.

- ***resolve(ComposableElement, ComposableElement)***: This operation takes two `ComposableElements` and resolves the conflict according to the behaviour of the specific metaclass that realises it.

ExplicitReconciliation: This type of reconciliation represents anticipated conflicts that are indicated to be reconciled. An `ExplicitReconciliation` can have a number of individual `ExplicitValues` indicating what is to be reconciled.

ExplicitValue: This metaclass represents the individual container to indicate an anticipated conflict.

- ***construct : String***: This represents the construct or class instance that the reconciliation is to be executed on.

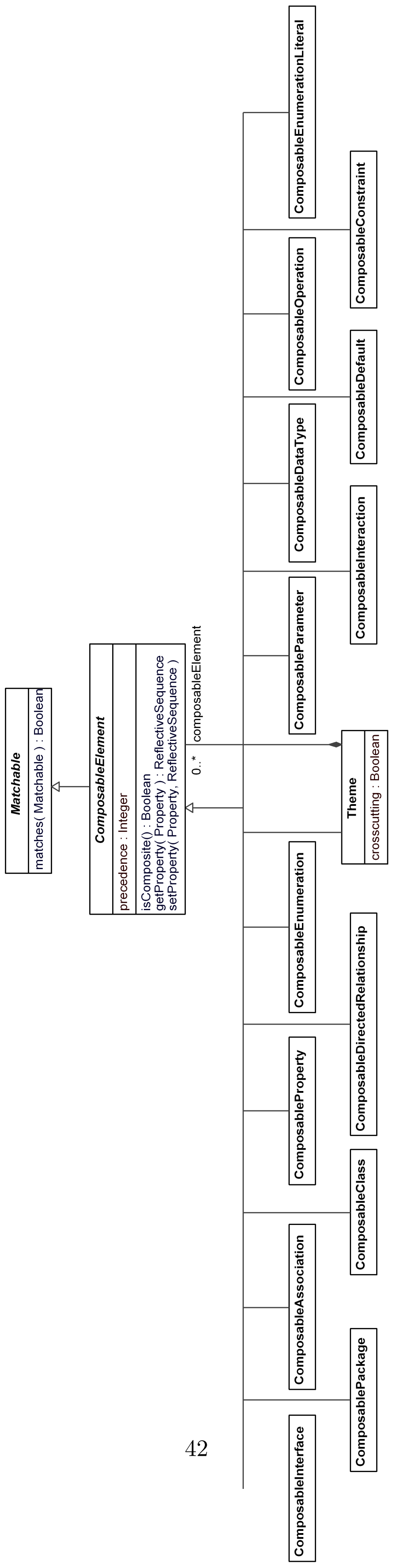


Figure 5.10: Composition Metamodel Elements

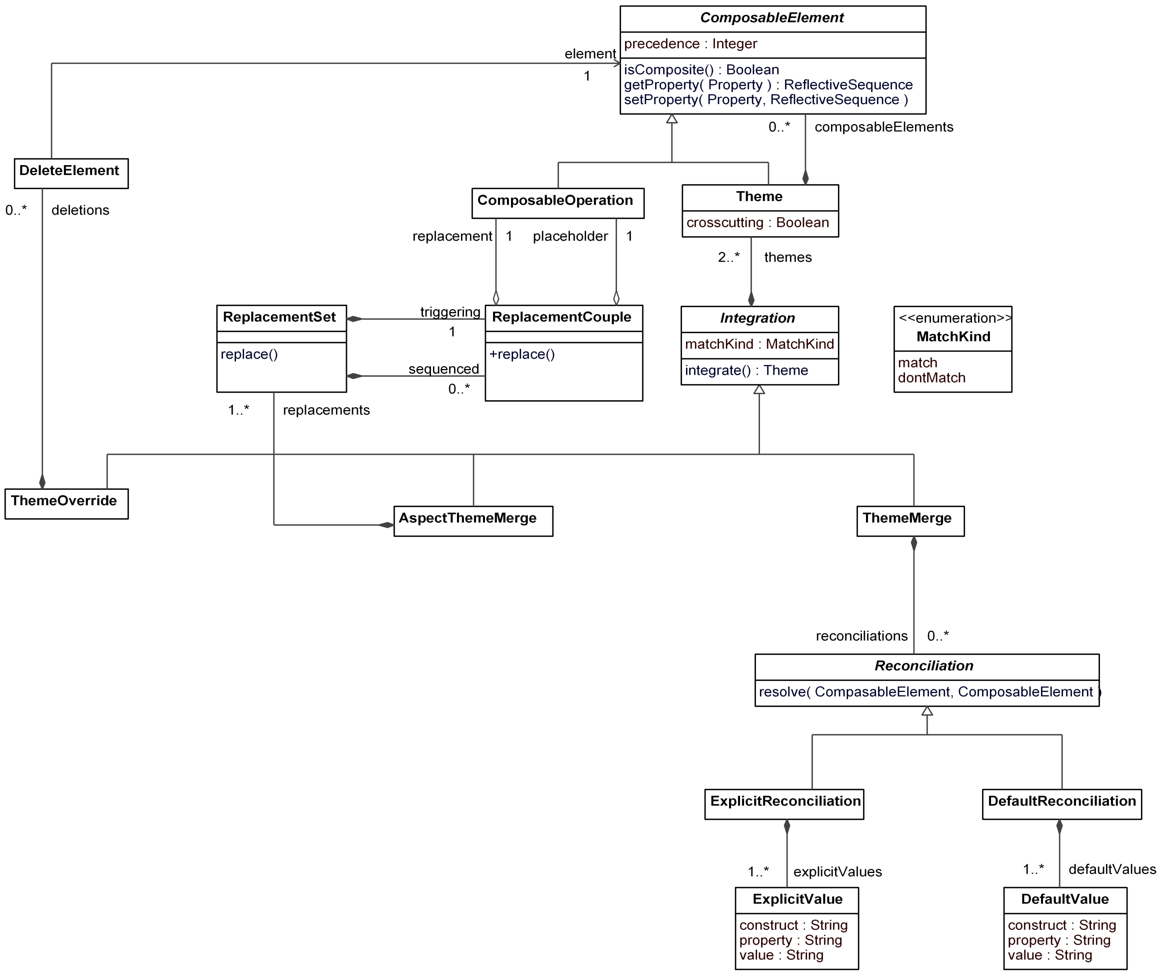


Figure 5.11: Composition Metamodel Integration Strategies

- ***property : String***: This represents the property of that class.
- ***value : String***: This represents the new value for that property.

DefaultReconciliation: This type of reconciliation represents a default resolution in case a conflict arises while merging. An `DefaultReconciliation` can have a number of individual `DefaultValue`s indicating what is to be reconciled.

DefaultValue: This metaclass represents the individual container to indicate the default behaviour if a construct arises on a specific construct.

- ***construct : String***: This represents the metaclass that the default resolution is attached to.
- ***property : String***: This represents the property that may conflict.

- ***value : String***: This represents the default value if that conflict occurs.

ThemeOverride: This metaclass represents an `Integration` Strategy when one theme's design specification takes priority over another, one theme can override another.

DeleteElement: This represents the `ComposableElement`s that are to be deleted from the overridden specification before the integration executes.

AspectThemeMerge: This `Integration` strategy represents a merge from an aspect theme to a base theme. Each template sequence is represented by an instance of a `ReplacementSet`.

ReplacementSet: A `ReplacementSet` represents a single template sequence of a crosscutting specification. It contains the triggering `ReplacementCouple` and a set of `ReplacementCouple`s representing the other items in the template sequence.

- ***replace()***: This operation defines the replacement behaviour.

ReplacementCouple: This metaclass represents the individual items in a `ReplacementSet`. Each instance of a `ReplacementCouple` contains two `ComposableOperation`s, the replacement and the placeholder.

- ***replace()***: This operation defines the behaviour that replaces the referenced `ComposableOperations`

5.6 Phase Three - Selection, Transformation and Synthesis

After re-factoring, the object-oriented PIM is ready to be transformed into a specific platform. The first step, the designer chooses the platform the object-oriented PIM is to be transformed to. The tool takes the PIM and using the mapping for the target environment transforms it and produces a PSM representing the domain-specific extensions of the PIM for that environment.

5.6.1 PIM-to-PSM Transformation

Rather than go straight from a PIM to code, the decision was made to go to an intermediate PSM. The first reason for this is that the proposed approach is elaborational. This means that the PIM is not computationally complete and does not contain the full executable application specification. Rather than pollute the PIM with platform extensions such as I/O, threading and network support (and make it less reusable), the PSM is open for re-factoring and elaboration of these low-level details by the designer. As a point of interest, MDA emphasises the use of powerful abstractions, with models representing different abstractions and transformations refining the platform commonalities between these models. Remember, the definition of a PIM is a model whose artefacts are common enough to provide the structure and behaviour of a system that is suitable for a number of different platforms *of the same kind*. The PIM of this approach is a very abstract PIM, whose representation is common enough for all kinds of object-oriented mobile target environments. As discussed by the MDA Guide, a PIM is defined from the perspective of a mapping specification. A model may be a PIM from the perspective of one mapping and a PSM from the perspective of another. The MDA Guide also suggests chaining mappings in such a way, with each refinement adding bit by bit more commonalities of the target environments. For example, .NET CF and J2ME may support I/O streams, which would be a commonality for a more refined PIM. However, a target environment such as C does not support streams and this PIM would not be suitable. It is not hard to imagine a hierarchy of refinements gradually being reduced to a subset of families of the same kind. With these vertical separation of concerns through refinements, it is not difficult to observe why the concept of MDA is so powerful. The proposed approach only uses one PIM as a proof-of-concept, but other PIM and their corresponding refinements are possible and indeed advised.

5.6.2 PSM-to-Code Transformation

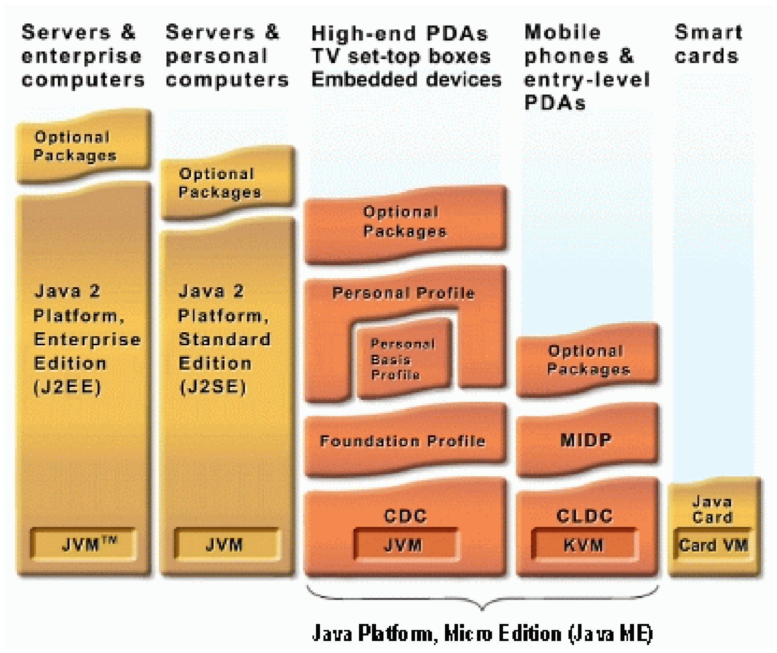
After elaborating the design of the PSM, the designer can decide to transform the model to code. The transformation from a PSM to code is known as *synthesis*. This kind of transformation is generally very straightforward as the PSM is a direct representation of the underlying platform, modelling the exact library support and features of that environment. From a pragmatic and productivity point of view, it is probably not suitable to model the full specification in the PSM. For example, one could imagine that programming a complex algorithm would be much more productive to be done in code than tediously modelling it with an UML activity diagram. If the full structural and behavioural specification is not modelling in the PSM, then it can be specified in code. However, in order to keep the model and code synchronised there is need for round-trip engineering, which represents a bi-directional transformation, synthesis as its forward and reverse-engineering as its inverse.

5.6.3 Mobile target platforms

J2ME

Java Platform, Micro Edition (J2ME)[3] is a subset specification of the Java platform from Sun, that is targeted for small and embedded devices such as mobile phones, PDAs, set-up boxes and printers. J2ME devices implement a profile, which are subsets of configurations (c.f. Figure 5.12). There are two profiles called the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC). CLDC contains a minimum subset of the Java class libraries for the Java virtual machine to operate. The CDC has a much richer set of libraries, mostly originating from the J2SE standard except the GUI related ones, The Personal Profile contains the AWT library and applet support. This approach will use the Personal Profile as a target platform.

[3]http://java.sun.com/javame/index.jsp



Servers & enterprise computers
Servers & personal computers
High-end PDAs TV set-top boxes Embedded devices
Mobile phones & entry-level PDAs
Smart cards
Optional Packages
Optional Packages
Optional Packages
Java 2 Platform, Enterprise Edition (J2EE)
Java 2 Platform, Standard Edition (J2SE)
Personal Profile
Personal Basis Profile
Optional Packages
Foundation Profile
MIDP
Java Card
CDC
CLDC
JVM™
JVM
JVM
KVM
Card VM
Java Platform, Micro Edition (Java ME)

Figure 5.12: J2ME Architecture

.NET CF

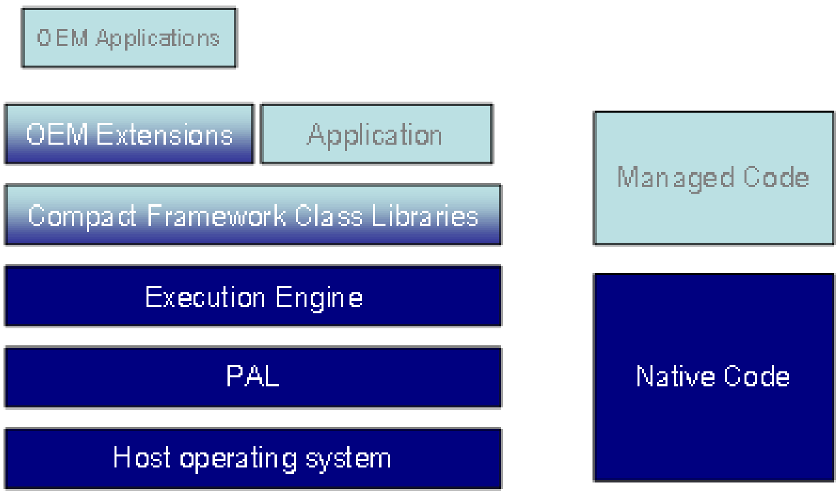


Figure 5.13: .NET Compact Framework Class Architecture

.NET CF[4], is a development framework from Microsoft that provides a rich subset of the

[4]http://msdn.microsoft.com/mobility/netcf/

.NET framework for resource-constrained devices, such as PDAs and mobile phones. It consists of a compact framework of class libraries from the .NET framework as well as additional libraries that support the development of mobile devices (c.f. Figure 5.13). The Common Language Runtime (CLR) is built specifically for the .NET Compact Framework, supporting requirements of small devices that have limited memory, resources and battery power. Managed code is compiled down to the Microsoft Intermediate Language (MSIL) and metadata. These are merged to a Pre Execution Environment (PE) file which is capable of being executed on a device that supports that specific CLR. When the executable is run, the JIT compiles the Intermediate Language (IL) to native code. The .NET Framework supports many different programming language that is compiled to the IL. This approach uses the .NET C# programming language (ECMA-334).

5.7 Summary

This chapter described the design of the proposed approach and the design decisions leading to its implementation. The approach was separated into three phases - the modelling phase, the composition phase and the transformation phase and the design of each was discussed in succession.

Chapter 6

Implementation

6.1 Introduction

This chapter discuss the implementation of the different phases. The chapter is split into three main sections, each representing each phase, the modelling phase, the composition phase and the selection, transformation and synthesis phase.

6.2 Phase one - Modelling

At this stage the designer is preoccupied with modelling the separate concerns and applying the composition relationships that implement the Theme/UML marking profile for composition.

6.2.1 UML 2.1 Graphical Modelling Tool

For a modelling tool there is a number of requirements. The first requirement is that the tool has sufficient support to model in the UML 2.1. The second requirement is that the tool should be able to support creating or importing an UML profile, to allow the model to be marked with the Theme/UML marking profile. The third requirement is that the modelling tool should be able to export to XMI, in particular an XMI format that is compatible with the transformations at the later phases (EMF XMI). The UML 2.1 is a massive specification. The majority of tools that claimed to be UML 2.1 compliant were either proprietary and expensive or only implemented a subset of the UML specification. Specifically, it was the goal to find a tool that supported structural and behavioural modelling with class diagrams and sequence diagrams respectively. It is well to note at this stage, the approach is not confined to any one tool. Any UML tool that supports the above requirements is suitable for this approach.

6.2.2 EMF

The Eclipse Modelling Framework (EMF) is a modelling framework for Eclipse., that unifies Java, XML and the UML. The metametamodel in EMF is called Ecore and is synonymous to MOF, although it has some slight variations. Ecore defines all EMF based models. The EMF implements the UML 2.1 standard called UML2 through Ecore. It also defines its own own XMI schema that allows the libraries to read and write any EMF based model.

6.2.3 MagicDraw

At the time, a survey of tools was done to select a tool that met the requirements and was free for personal or academic use. The modelling at this phase was decided to be supported by an UML tool called MagicDraw Academic edition[1]. This tool appeared to be the best candidate for the job. It claimed to be UML 2 compliant, allowed UML profiles to be defined, supported class and sequence diagrams and exported to the EMF XMI format. However, there were a few difficulties with the tool that were not discovered until later. A `Dependency` was not working according to the UML 2.1 specification, which was required as a major extension point for the Theme/UML marking profile (c.f. Figure 5.5). Likewise, it was discovered that the EMF XMI Export that Magicdraw provided was faulty for Sequence Diagrams. The next two subsections will discuss the work-arounds for these two problems.

6.2.4 Workaround for modelling composition relationships

Theme/UML defines an n-ary composition relationship for elements that are to be composed by its merge. As a profile extension can only mark the existing UML elements, two candidates are possible for profile extensions for n-ary relationships, `Association` and `Dependency`. An `Association` is restricted as a relationship between certain types and is unsuitable as a marking profile extension for Theme/UML, as those types are not the same for Theme/UML. Therefore a `Dependency` was the only option left. In the UML 2.1 superspecification,

> A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation.

This relationship is ideal because it is a n-ary to n-ary relationship that targets `NamedElements` and is suitable for extension by the Theme/UML marking profile.

However, Magicdraw only supports one-to-one relationships with a `Dependency` and as such deviates from the standard. For this reason, those relationships can not be n-ary. To workaround this, n-ary dependencies are emulated by drawing an additional dependency on the dependency that was drawn between two model elements. Since a dependency can be drawn between any `NamedElement` and a `Dependency` itself is a `NamedElement` this workaround was successfully implemented. However, this lead to extra parsing logic to determine all the elements participating in a composition relationship.

[1]http://www.magicdraw.com

6.2.5 Workaround for modelling sequence diagrams

The faulty EMF XMI Export rendered using Magicdraw for sequence diagrams useless. To workaround this, the EMF UML2 provides a tree-based graphical tool to view the XMI of a model. The workaround entails creating the sequence diagrams by hand using the tree-based graphical tool. This tool offers the designer a little more abstraction than working with the raw XMI files directly. Note here, that writing to the UML 2 is very difficult and tedious. It requires detailed knowledge of the specification and without the support of a direct graphical tool that supports standard UML graphical conventions, it is very error-prone, even for an expert. The other option would be to create a hardcoded transformation that would take the EMF XMI file, read it and add in the expected sequence diagrams using the EMF UML2 library. This option would be only be slightly better as the sequence diagrams would be preserved in a transformation rather than in a disposable model. No other tool surveyed was capable of viewing or writing sequence diagrams to EMF XMI correctly. This workaround is only necessary until a tool that supports sequence diagrams becomes available.

6.2.6 Implementation

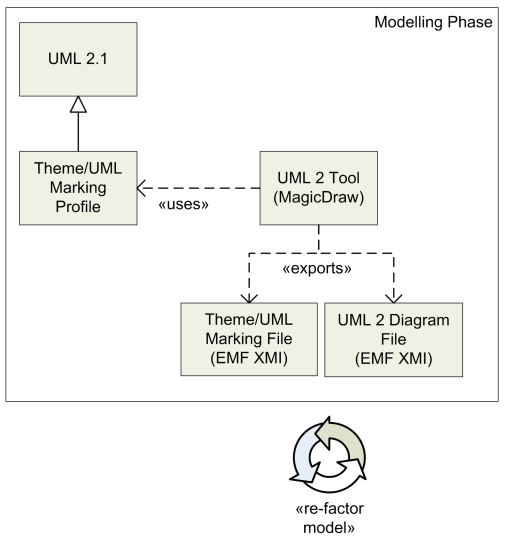

Modelling Phase
UML 2.1
Theme/UML Marking Profile
«uses»
UML 2 Tool (MagicDraw)
«exports»
Theme/UML Marking File (EMF XMI)
UML 2 Diagram File (EMF XMI)
«re-factor model»

Figure 6.1: Implementation of Modelling Phase

Figure 6.1 indicates the final implementation of phase one. The figure illustrates the Theme/UML Marking Profile extending the UML 2.1. A UML 2 Tool (in this case Magicdraw), uses this marking profile to design and apply the composition relationships. This tool then exports two files, the profile and the model in EMF XMI format. The circular arrows indicate that the model then needs to be re-factored further if need be. In this case, this indicates the sequence diagrams be added with the UML2 tree-editor as discussed in the previous section as a workaround.

6.3 Phase two - Composition

At this phase, the designer composes the model with the composition relationships applied. As illustrated in Figure 6.2, the tool takes the two EMF XMI files, the Theme/UML Marking Profile and the UML 2 model with the design. Using these two files, it maps and creates a composition model that is an instance of the composition metamodel illustrated in the previous chapter (c.f. Figures 5.10 and 5.11). The composition model is executed and an EMF XMI file is produced that holds the Object-Oriented PIM. The transformation here is illustrated as horizontal, as the level of abstraction is not changed. At this point, as discussed in the Design Chapter, the model can then be re-factored, illustrated by the circular arrows.

This section discusses the implementation of the composition metamodel and the two transformations illustrated in Figure 6.2. The first transformation maps the UML 2 Diagram File to a composition model using the Theme/UML Marking File and composition metamodel. The second transformation is the composition process.

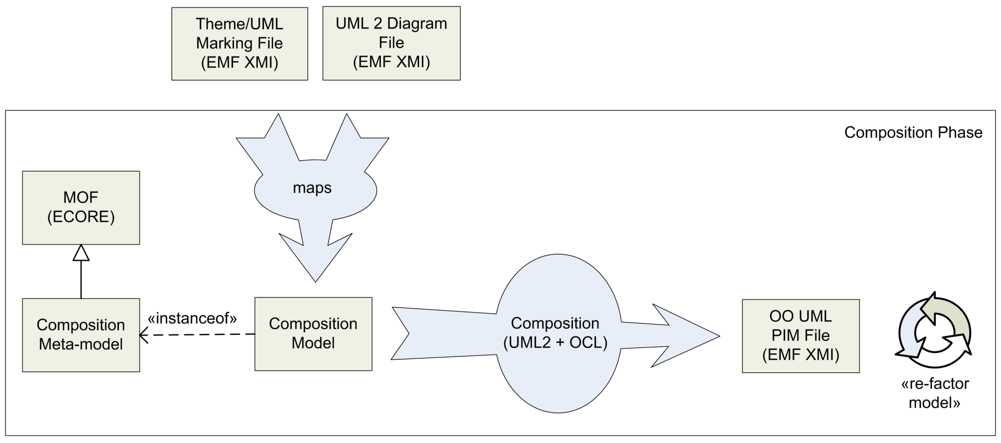


Figure 6.2: Implementation of Composition Phase

6.3.1 Mapping from Theme/UML Marking Profile to Composition Metamodel

As illustrated in Figure 6.2, the tool parses the two EMF XMI files, the Theme/UML Marking Profile and the design in UML using the EMF and UML2 libraries to read the XMI.

6.3.2 Step 1 - Resolving Explicit Composition Relationships

The first step is to resolve the `explicit` composition relationships. From the Theme/UML marking profile (c.f. Figure 5.5) `explicit` has a tagged value called `mergedName`. An `explicit` composition relationship between two or more elements with a `mergedName` indicates that these elements represent the same concept and are designated as `mergedName`. Therefore it makes sense that these are resolved first before the composition metamodel is created.

6.3.3 Step 2 - Parsing and generating AspectThemeMerges

The second step involves parsing the `bind` composition relationships from the aspect themes to the base themes. The tagged value `binding` on each `bind` corresponds to the `template` of each theme. However, as these are tagged values, they are strings. These strings represent the model elements. The composition metamodel deals with real elements and not string representations, therefore the model must be queried to extract the real elements using their string counterparts. A `binding` in Theme/UML has a specific format and a parser is needed to flatten out the bindings to a set of absolute sequences. For example the bind sequence:

```
<Location.{takeCrystals(), stealCrystals()}, Player.crystalsTouched()>
```

would be flattened to

```
<Location.takeCrystals(), Player.crystalsTouched()>
<Location.stealCrystals(), Player.crystalsTouched()>
```

Binding Grammar

```
<bindings> ::= <BIND> <LBRAK> <LCHEVRON> <binding> <RCHEVRON> [ {<LCHEVRON> <binding> <RCHEVRON> } ] <RBRAK>
<binding> ::= <nestedBinding> [ { <LCHEVRON> <nestedBinding> <RCHEVRON> } ]
<nestedBinding> ::= <matchGroup> [ { <COMMA> <matchSingle> } ]
<matchGroup> ::= <matchSingle> | <LCURLY> <matchSingle> [ { <COMMA> <matchSingle> } ] <RCURLY>
<matchSingleTrigger> ::= <STRING> <PERIOD> <matchSingleOp> | <LCURLY> <matchSingleOp> [ { <COMMA> <matchSingleOp> } ] <RCURLY>
<matchSingle> ::= <STRING> <PERIOD> <matchSingleOp>
<matchSingleOp> ::= <STRING> <LPAREN> [ <matchParams> ] <RPAREN>
<matchParams> ::= <STRING> | [ { <COMMA> <STRING> } ]

Tokens
<EOF> ::= 0x00
<BIND> ::= ('b' | 'B') ('i' | 'I') ('d' | 'D')
<LCURLY> ::= '{'
<RCURLY> ::= '}'
<LCHEVRON> ::= '<'
<RCHEVRON> ::= '>'
<LBRAK> ::= '['
<RBRAK> ::= ']'
<PERIOD> ::= '.'
<STRING> ::= ('a' ... 'z' | 'A' ... 'Z' | '0' ... '9' | '_' | '(' | ')' | '*' | '=' | ':' )  [ { <STRING> } ]
Whitespace is skipped
```

Querying Model Elements

When the operations of the `binding`s are extracted, it is necessary to query the model to identify the model elements that the string represented for the composition metamodel. However, Theme/UML also uses wildcards to specify multiple elements in a `binding`. These are mapped into an OCL expression. OCL is a querying language and is suitable for this kind of task.

For example

```
Location.add*()
```

would be mapped into this OCL expression

```
self->collect(packagedElement)
->select(p: PackageableElement | p.oclIsTypeOf(Class))
->iterate(p; cs : Set(Class) = Set{} | cs->including(p.oclAsType(Class)))
->select(c : Class | c.name = 'Location')
->collect(ownedOperation)
->iterate(o; os : Set(Operation) = Set{} | os->including(o.oclAsType(Operation)))
->select(o : Operation
          | ('add'.size() <= o.name.size() and o.name.substring(1, 'add'.size()) = 'add'))
```

6.3.4 Step 3 - Creating ThemeMerge and ThemeOveride

The next step is to generate the ThemeMerges and ThemeOverrides from the two files. The reconciliations for the merge also need to be parsed and a grammar is provided for parsing this below. An example of this is:

```
resolve ( Player.crystals (isQuery=true, isOrdered=false) )
```

Merge Grammar

```
<resolver> ::= <RESOLVE> <LPAREN> <constructs> <RPAREN>
<constructs> ::= <construct> [ { <construct> } ]
<construct> ::= <STRING> <LPAREN> <pvpair> [ { <SEMICOLON> <pvpair> } ] <RPAREN>
<pvpair> ::= <STRING> <EQ> <STRING>


Tokens
<EOF> ::= 0x00
<RESOLVE> ::= ('r' | 'R') ('e' | 'E') ('s' | 'S') ('o' | 'O') ('l' | 'L') ('v' | 'V') ('e' | 'E')
<LPAREN> ::= '('
<RPAREN> ::= ')'
<SEMICOLON> ::= ';'
<EQ> ::= '='
<STRING> ::= ('a' ... 'z' | 'A' ... 'Z' | '0' ... '9' | '_') [ { <STRING> } ]
Whitespace is skipped
```

6.3.5 Realisation of the Composition Metamodel

In the previous chapter, the design of the composition metamodel was discussed. The Design chapter discussed the design of the composition metamodel in MOF. ECORE is similar to MOF, providing a metametamodel for the EMF. It has similar features and capabilities as MOF so realising the composition metamodel in ECORE was straightforward. The EMF provides a visual class-diagram-like tool that eases the development of designing metamodels in ECORE.

6.3.6 Aspect-Oriented Weaving Procedure

In order to weave the aspect themes to the base, it is necessary to merge the base themes. The first step is to create a `ReferenceManager` that takes care of correcting the references of elements while merging. For example, when base themes get merged, there may be a binding on an element that gets resolved. The reference for this would need to get updated as the base occurs. After the bases are merged, the next step is to weave the aspect themes. For each sequence being bound, a new instance is created of the aspect theme and it is redefined in terms of a `ThemeMerge`. This instance then gets composed into the merged base theme.

6.4 Phase three - Selection, Transformation and Synthesis

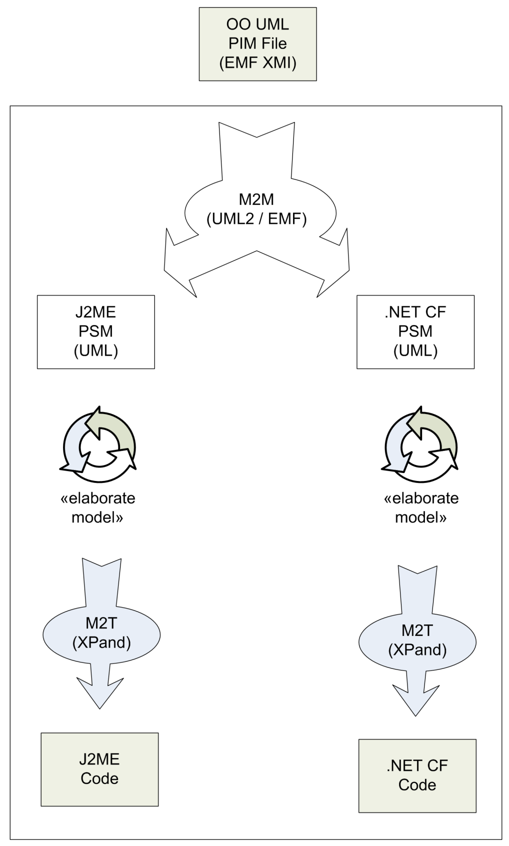


Figure 6.3: Implementation of Transformation Phase

At this phase, the designer needs to choose which platform to transform into. Figure 6.3 illustrates the two separate platforms that this approach implements, J2ME and .NET CF. After choosing the platform, a transformation adds in the additional platform specific extensions. This transformation uses the EMF UML2 library. Once the PSM is generated, this model can be elaborated further using the UML2Tools graphical tool. XPand is a template-based Model-to-Text tool which is part of the oAW framework that supports synthesis of a model to code.

6.4.1 Choice of Model-to-Model Language

A Model-to-Model Language (M2M) is a domain-specific language that is used to write model-driven mappings. The OMG provides the QVT standard, as discussed in Chapter 2, that allows mappings to be defined. However, the standard is still in revision and as of writing there is no tool that is stable and fully supports all of QVT's features. Some additional M2M languages that were surveyed for suitability include ATL[2], Kermeta[3] and oAW XPand[4]. ATL is probably the most mature QVT-like language and supports a declarative programming style with some imperative constructs. Kermeta is an imperative language that does not directly support transformations in the traditional style but supports it through its capabilities to add executability at the metamodel level. oAW XPand is a functional language that supports transformations. One of its features includes writing plugins that use the Java libraries which make it pretty powerful. However the language is quite new and it seemed difficult to write mappings to and from UML. All of the surveyed M2M tools are implemented on top of the Java EMF and UML2 libraries. It was decided that it was better to use Java and these libraries rather than one of these domain-specific languages. M2M languages are still evolving and at the moment are not mature [12]. These transformation languages are suitable for writing straightforward mappings based on simple metamodels. The UML 2 is an extremely large and complex specification and writing valid transformations with UML as a target is deemed "very intricate, non-intuitive and complicated."[5] Initially, an attempt was made to learn these new domain-specific languages but they proved too inflexible, time-consuming and often had bugs or missing features.. The direct approach using Java was more convenient, more powerful and flexible. For each PSM, an UML profile was created extending the standard UML datatypes with those that were specific to the language and platform. The profile can also include the namespaces and datatypes needed to use to elaborate the PSM further. The transformation from the OO PIM involves transforming the UML and datatypes provided by the Magicdraw environment into those equivalent for the PSM.

6.4.2 UML2Tools

The UML2Tools[6] is a graphical tool extension in Eclipse using the Graphical Modelling Framework (GMF) allowing some support to view and model in the UML 2. This tool can be used to view the composition result and the PSM models to further elaborate. At the time of design, this tool was not mature enough to use, preferring Magicdraw that had better support. However, during the course of the dissertation, this project has evolved further and the class diagrams are now quite stable. The other diagrams, for example sequence diagrams, are still not supported though. At some point, this free, opensource tool can be used as a replacement for Magicdraw, if it evolves further.

6.4.3 Choice of Model-to-Text Language

Model-To-Text (M2T) transformations involve methods to transform models to code. There are two approaches for M2T transformations, visitor-based approaches and template-based approaches [12].

[2]http://www.eclipse.org/m2m/atl/

[3]http://www.kermeta.org

[4]http://www.openarchitectureware.org

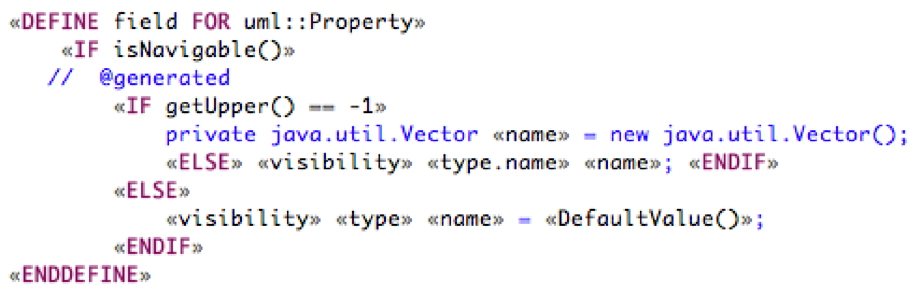
[5]http://www.eclipse.org/gmt/oaw/doc/4.1/52_m2mWithUML2Example.pdf

[6]http://www.eclipse.org/modeling/mdt/?project=uml2tools

Visitor-based approaches use the visitor pattern [19] to traverse the elements of the model and write the code to the text stream. Template-based approaches use a text-based declarative language as a means for selection of model nodes and iterative expansion. It was decided to use oAW XPand to transform the UML class diagrams to code. XPand is a template-based approach. It supports the ability to write extensions in Java. For the sequence diagrams, a visitor-based approach was used in Java. XPand allowed these two approaches to be combined to transform both the structure and behaviour of the PSM models.

6.4.4 XPand

XPand[7] is a template based transformation language used to generate code and is part of the oAW. Its IDE is integrated to Eclipse. It allows for Java based extensions that can be called from the templates. Figure 6.4 illustrates an example of XPand. This example demonstrates how endpoints of associations in the UML class diagrams are mapped to code.



```
«DEFINE field FOR uml::Property»
    «IF isNavigable()»
   //  @generated
        «IF getUpper() == -1»
            private java.util.Vector «name» = new java.util.Vector();
            «ELSE» «visibility» «type.name» «name»; «ENDIF»
        «ELSE»
            «visibility» «type» «name» = «DefaultValue()»;
        «ENDIF»
«ENDDEFINE»
```

Figure 6.4: XPand Example

6.4.5 Sequence Diagrams - Generating Code

A visitor-based approach in Java was designed to generate code from the sequence diagrams. Sequence diagrams are written in the UML in-order so a visitor-based approach is more desirable than a template one, where the visitor can step through the full trace in-order and generate code on the fly. Only a small subsection of the UML specification was used to generate code for the case study as the full spec was too much for such a short time limit. It was revealed that sequence diagrams in the UML 2.1 specification are currently problematic and currently unsuitable for code generation purposes. The OMG Revision Task Force for UML [8] currently lists a number of issues with sequence diagrams that need to be resolved. For example, the standard is ambiguous about how to create and reference variables belong to structure. There is also no way to reference variables as arguments of a `Message`. To get around this, the arguments were sent in textual format. However, this is undesirable. These issues need to be resolved before code generation using sequence diagrams can become a reality.

[7]http://www.eclipse.org/gmt/oaw/doc/4.1/r20_xPandReference.pdf
[8]http://www.omg.org/issues/

6.5 Summary

This chapter discussed the implementation of this approach in the different phases, the modelling phase, the composition phase and the selection, transformation and synthesis phase.

Chapter 7

Case Study and Evaluation

7.1 Introduction

This chapter presents a mobile, context-aware case study to evaluate the outlined approach (c.f. Figure 5.4) that this dissertation presents. This case study demonstrates how a domain-specific, mobile context-aware concern that emerges from a requirement specification can be modelled separately from the application concerns and then illustrates how it can be composed with them. The motivation is to evaluate and ascertain the suitability of this approach for mobile, context-aware systems. The results highlight the strengths and weaknesses of using models and aspects in such a way for this domain.

7.2 Requirements Specification

The case-study presented is a distributed auction system[1] that has been adapted for use with mobile devices. Like a traditional auction system, it presents a number of different features for the user who wishes to browse bids, write comments, place bids, manage an account and buy goods. As the auction system is distributed and mobile, extra features are added to make the application context-aware. These include allowing the user notification of auction events that may be interesting to her, ensuring the user is in a valid location before proceeding with a transaction and customising the user interface to changes in the environment. The full requirements of the auction system have been analysed with Theme/Doc and decomposed to a subset of requirements to be designed by individual themes. The base themes in this example represent the main application concerns, while the context-aware concerns are represented by aspect themes.

7.2.1 Enrolling with the System

Before using the auction system, a user must first enroll with the system by providing her name, age, credit-card details and address. Enrolling succeeds if the bank validates the credit-card number with

[1] http://lgl.epfl.ch/research/fondue/case-studies/auction/problem-description.html

the name. After validating the credit-card details, the system creates an account and stores the users details with the account. After enrolling, a customer can log-on to the system to start a session by providing their details. A user can logout and leave the session, once logged into a session.

7.2.2 Transferring Credit

Before the system closes a customers account, the system can transfer credit back to the customers credit-card if the credit amount is greater than zero. A customer can also request the system to transfer credit between the credit-card and system account. The system can transfer credit with a customers credit-card only if the bank validates the transfer. The system can freeze or unfreeze credit in a customers account. If an account is frozen, the system rejects all requests by the customer to transfer credit. After an auction is terminated, the system must transfer credit from the account of the customer that has the auctions highest bidder to the account of the customer that owned the auction.

7.2.3 Auction Management

During a session, a customer can request the system to create an auction by providing the item name, description, minimum price and duration in hours of the auction. Each auction stores the highest bid amount offered along with the customer who placed the bid. Each auction has a private bulletin board that the customer who created an auction can view. After an auction is created it becomes active and it is terminated when the duration of the auction expires. When an auction is terminated, all customers are forced to leave the auction.

7.2.4 Joining an Auction

During a session, a customer can join an auction if it is active. After joining an auction, a customer can leave that auction only if that auction has not been terminated and if the customer is not the highest bidder on that auction. After joining an auction, a customer can post on the bulletin board and can place a bid.

7.2.5 Placing a Bid

A customer can only place a bid if the bid amount is greater than the auctions highest bid, and the customers account credit is equal to or greater than the auctions highest bid. After a customer places a bid, if the auction has a customer that holds the current highest bid, that customers account is unfrozen. The current customer becomes the new highest bidder and his bid offer becomes the auctions highest bid amount.

7.2.6 Browsing Auctions

During a session, a customer can request the system for all auctions to browse and be displayed on the customers device. During a session, a customer can search the system for auctions by specifying a description of the item to be searched and / or indicate the location.

7.2.7 Adapting the User Interface

Each user with a mobile device connected to the auction system has a background light as part of the user interface. When a customer logs on or off to the system the background light is activated to let the user know that something has happened. Likewise when a customer joins, leaves or posts a message on an auction the light is activated.

7.2.8 Adapting to Location

It is required by the auction system that a user that is to make an important transaction within the system must be in a safe location. An important transaction is for example withdrawing or depositing money from an account, placing a bid or enrolling in the system.

7.2.9 Auction advertising

The system keeps track of all the auction activity by all users in the system. If an auction event occurs such as a customer posting a message or placing a bid, the user is notified according to his or her past activities.

7.3 Designing Themes

The previous section presented the requirements for the auction system and decomposed them into a set of manageable themes. As illustrated in Figure 5.4, the process starts off in phase one by modelling both the application and context-aware concerns. Theme/UML supports that each theme can be delegated to an individual or team to design in parallel. However, with such a small case-study, an individual designer is capable of doing them all in succession. It is also interesting to note how the context-aware concerns can be designed orthogonally with the other application concerns without the need for collaborating on the design between themes. The previous subsections will give an outline of the design decisions made for each theme.

7.3.1 Enroll Theme

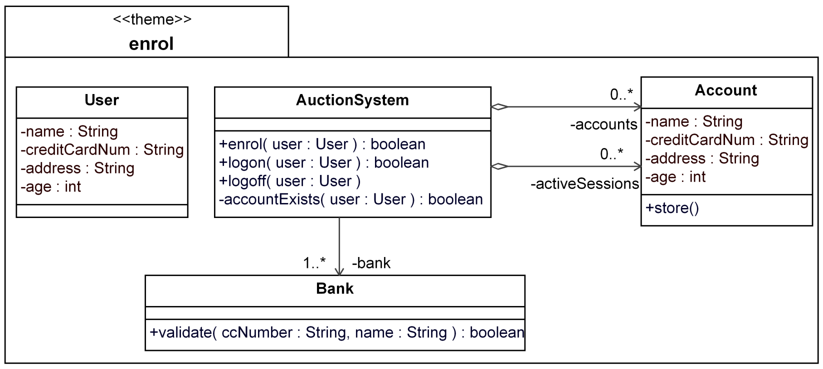
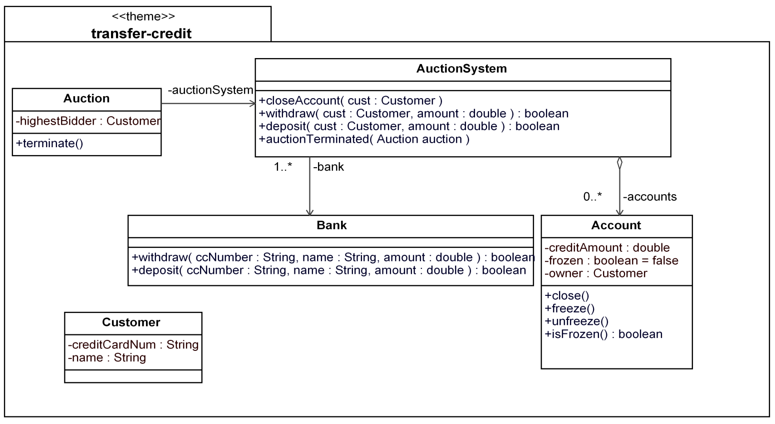


Figure 7.1: Enroll Theme

Figure 7.1 illustrates the design of the enroll theme, from the perspective of enrolling users and managing their sessions. This responsibility is given to the `AuctionSystem` class. When a user initially enrolls with the system, the name of the user and the credit-card number is validated with the `Bank` and an `Account` is created and stored, illustrated by the `accounts` composite aggregation. The user can then `logon` and `logoff` at will with the system. All logged on users' accounts are stored by the `AuctionSystem` as `activeSessions`.

7.3.2 Transfer Credit Theme



<<theme>>
transfer-credit
Auction
-highestBidder : Customer
+terminate()
-auctionSystem
AuctionSystem
+closeAccount(cust : Customer)
+withdraw(cust : Customer, amount : double) : boolean
+deposit(cust : Customer, amount : double) : boolean
+auctionTerminated(Auction auction)
1..*
-bank
0..*
-accounts
Bank
+withdraw(ccNumber : String, name : String, amount : double) : boolean
+deposit(ccNumber : String, name : String, amount : double) : boolean
Account
-creditAmount : double
-frozen : boolean = false
-owner : Customer
+close()
+freeze()
+unfreeze()
+isFrozen() : boolean
Customer
-creditCardNum : String
-name : String

Figure 7.2: Transfer Credit Theme

The transfer credit theme manages all money transactions in the auction system (c.f. Figure 7.2). The `AuctionSystem` provides the `Customer` the ability to `deposit` or `withdraw` money from his Account to or from the Bank respectively. This is done through the credit-card details. When an Auction is terminated, this theme manages the transfer of money between the buyer and the highest bidder. The boolean return values in the Bank methods return true if the transaction with the Bank was successful. This ensures all transactions are validated. An Account can also be frozen which indicates that the customer of that account is involved in a bid. The AuctionSystem is responsible for managing the status of Accounts.

7.3.3 Manage Auction Theme

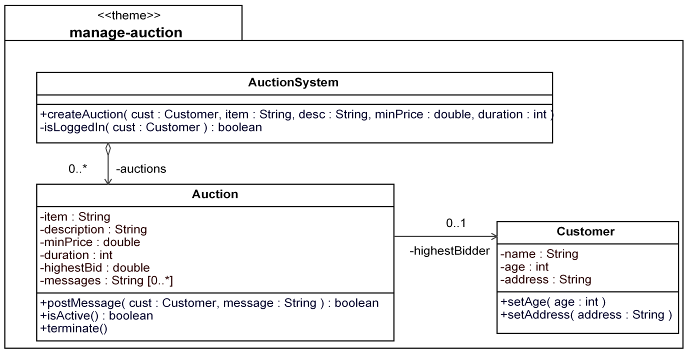
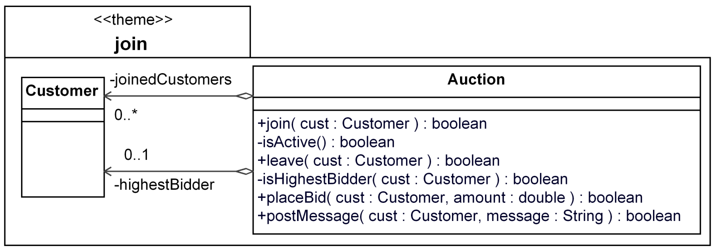


Figure 7.3: Manage Auction Theme

This theme as illustrated in Figure 7.3 is responsible for the maintenance operations of auctions in the system. The AuctionSystem class handles creating an Auction on behalf of a user and stores the ongoing auctions. The Auction class stores its highestBidder along with the highestBid, the item for sale and its description. Each Auction also has a bulletin board illustrated by messages attribute. A Customer is permitted to post to the board if he is participating in the auction.

7.3.4 Join Theme



<<theme>>
join
Customer
-joinedCustomers
0..*
0..1
-highestBidder
Auction
+join(cust : Customer) : boolean
-isActive() : boolean
+leave(cust : Customer) : boolean
-isHighestBidder(cust : Customer) : boolean
+placeBid(cust : Customer, amount : double) : boolean
+postMessage(cust : Customer, message : String) : boolean

Figure 7.4: Join Theme

The join theme (c.f. Figure 7.4) is modelled from the perspective of customers joining an auction and the functionality associated with this. An Customer can join an Auction if it is active.

After joining, the `Customer` can also place a bid or post a message to the bulletin board. The `Customer` can also `leave` at any time provided he is not the `highestbidder` in the `Auction`.

7.3.5 Bid Theme

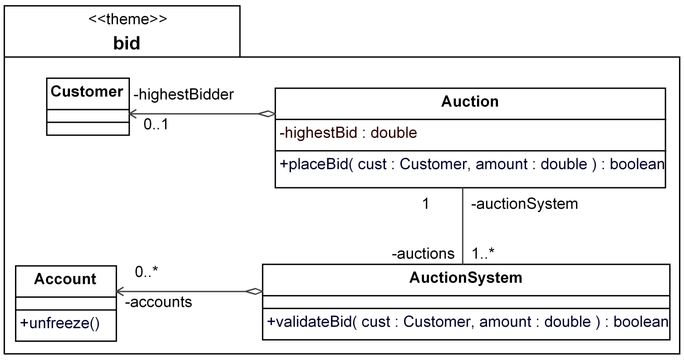
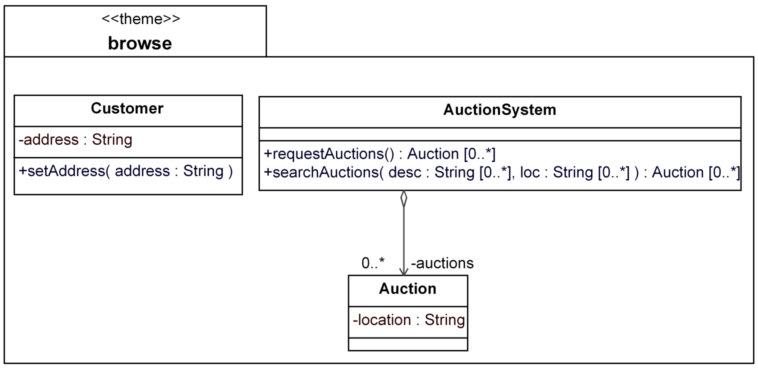


Figure 7.5: Bid Theme

Figure 7.5 illustrates the bid theme which is responsible for modelling the functionality of bidding in the auction system. An `Customer` is allowed place a bid in an `Auction` represented by the `placeBid` method. The `Auction` checks with the `AuctionSystem` to make sure the bid is solvent. Success in these methods is represented by the `booleans`. The `AuctionSystem` will `unfreeze` an `Account` if that account is no longer the `highestBidder`.

7.3.6 Browse Theme



<<theme>>
browse
Customer
-address : String
+setAddress(address : String)
AuctionSystem
+requestAuctions() : Auction [0..*]
+searchAuctions(desc : String [0..*], loc : String [0..*]) : Auction [0..*]
0..*
-auctions
Auction
-location : String

Figure 7.6: Browse Theme

The browse theme (c.f. Figure 7.6) shows how a `Customer` can browse auctions using different criteria. This is illustrated by the `searchAuctions` and `requestAuctions` methods provided by the `AuctionSystem`. The `Customer` can search according to description and location. Since the `AuctionSystem` contains all `auctions` it can simply iterate through all of them and return the results to the customer.

7.3.7 Adapt UI Theme

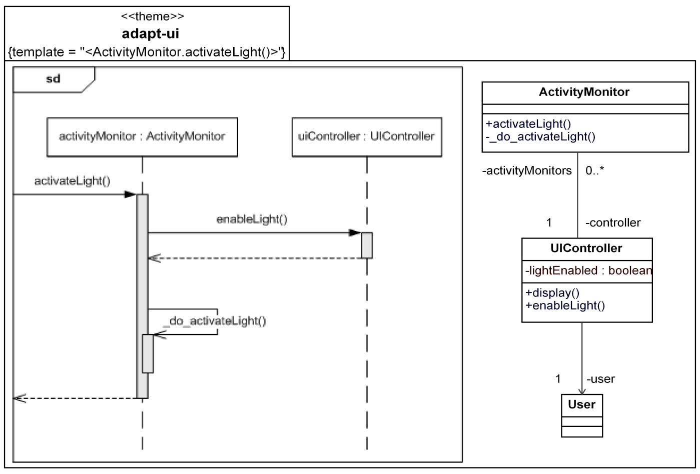


Figure 7.7: Adapt UI Theme

The adapt-ui theme (c.f. Figure 7.7) is a context-aware aspect theme specifying when the background light on a device is to be activated. The adapt-ui theme is designed independently of the activities that trigger the background light, which are postponed until composition. Here a clear separation is demonstrated between the specification of this context-aware concern and how that theme affects the rest of the system. Normally the ramifications of designing such a crosscutting context-aware concern with traditional formalisms would cause scattering and tangling with other themes. As indicated in Figure 7.7, `ActivityMonitor.activateLight()` is the triggering template that will later be bound to at composition. In AspectJ parlance this is similiar to a join point. The contained sequence diagram illustrates the invocation of this triggering operation originating from the gate, demonstrating that the designer is not concerned with what or how this gets invoked. The rest of the sequence diagram shows the behaviour of the context-aware concern that gets executed as a result, with the operation `ActivityMonitor._do_activateLight()` indicating when the behaviour of the triggering operation gets executed. The rest of the theme is designed as normal. The `UIController` represents the controller for the user interface and `ActivityMonitor` is the class that represents the placeholder for the operation that is being bound to at composition time.

7.3.8 Location Theme

The location theme (c.f. Figure 7.8) is another context-aware aspect theme. This theme is responsible for providing the functionality for producing and querying the location of a user. The

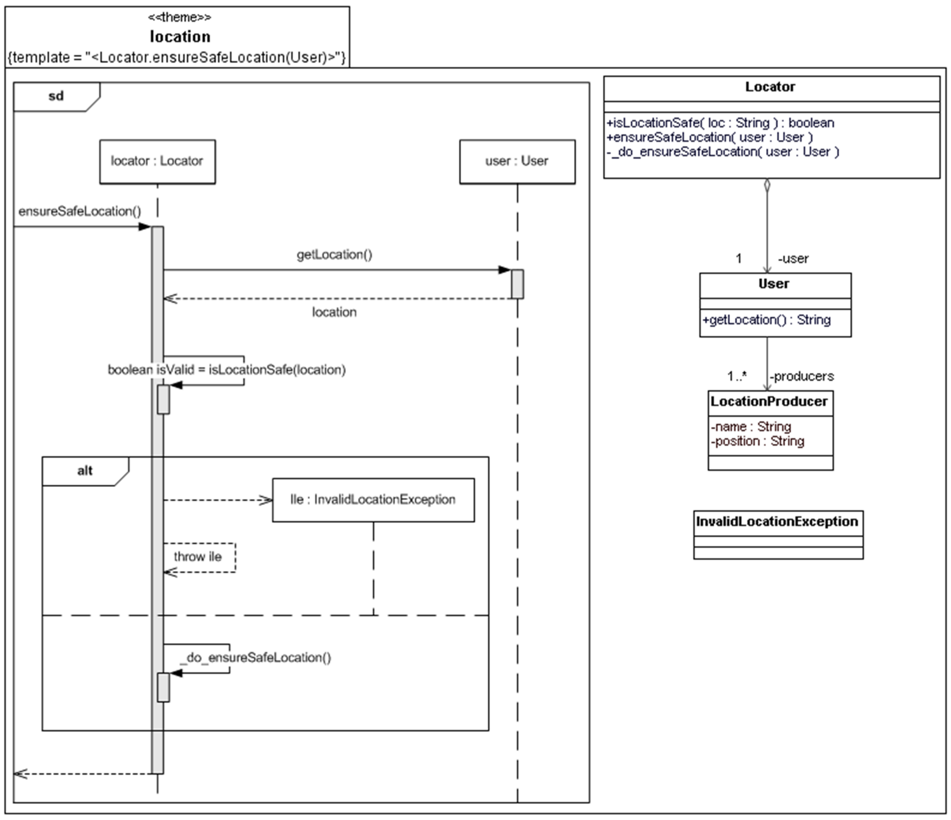
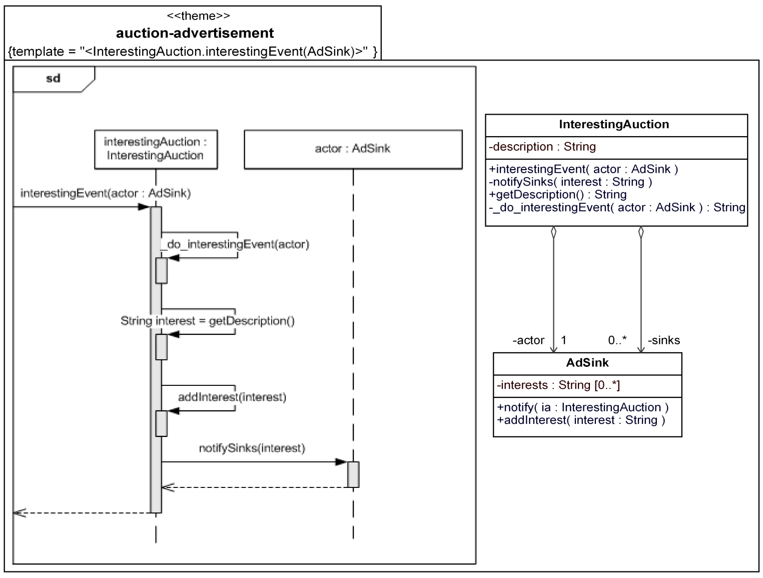


Figure 7.8: Location Theme

`LocationProducer` is a class that will be further elaborated at the PSM, but illustrates the requirement that the device should have some kind of functionality that provides location information. At this time, location is just specified as a `String`, as it seems to be the most generic data type that would be suitable for representing a location. The `User` can have one or more of these `LocationProducer`s depending on the capabilities of the device. The class `Locator` represents the placeholder for the template and will later be bound to. Like the previous context-aware aspect theme, there is a sequence diagram being triggered by an operation `Locator.ensureSafeLocation(User)` from the gate. A check is performed against the location retrieved from the user and if it is unsafe, an `InvalidLocationException` is thrown. The bindings specify the operations that the templates get bound to. On a side note, the UML 2 specification does not have any means to throw an exception in sequence diagrams, as it is probably considered too platform specific. In this scenario, the message used to indicate throwing an exception is annotated with `throw` and extra parsing logic can provide the additional logic to transform this to a platform-specific construct that is pretty much common in object-oriented technologies.



<<theme>>
auction-advertisement
{template = "<InterestingAuction.interestingEvent(AdSink)>" }
sd
interestingAuction : InterestingAuction
actor : AdSink
interestingEvent(actor : AdSink)
_do_interestingEvent(actor)
String interest = getDescription()
addInterest(interest)
notifySinks(interest)
InterestingAuction
-description : String
+interestingEvent(actor : AdSink)
-notifySinks(interest : String)
+getDescription() : String
-_do_interestingEvent(actor : AdSink) : String
-actor 1
0..* -sinks
AdSink
-interests : String [0..*]
+notify(ia : InterestingAuction)
+addInterest(interest : String)

Figure 7.9: Auction Advertisement Theme

7.3.9 Auction Advertisement Theme

Figure 7.9 illustrates the auction advertisement context-aware aspect theme. This theme provides the functionality of accumulating and tracking activity associated with a user in order to notify them of auctions that may be of interest to that user. The class `AdSink` represents the sink for the notification. An `InterestingAuction` has a `description` which is used to accumulate the interest of the `actor` of the interesting activity. These `interests` are collected in the `AdSink` class. The sequence diagram illustrates how the aspect theme is executed in relation to the triggering template `interestingEvent()`. When an interesting event occurs, the description is retrieved and that interest is accumulated. The `InterestingAuction` then notifies the other `AdSink`s if that auction is of interest to them based on a calculation of their previous interests. This context-aware theme is modelled in such a way that the interesting event is not yet bound to. This takes place at the composition phase.

7.4 Composition Specification

After all the themes have been decomposed, designed and modelled, the designer needs to compose them, as illustrated in Figure 5.4 This section discusses the use of Theme/UML's composition relationships to specify how the individual themes can be unified to create one individual amalgamated theme representing the full structure and behaviour of the auction system.

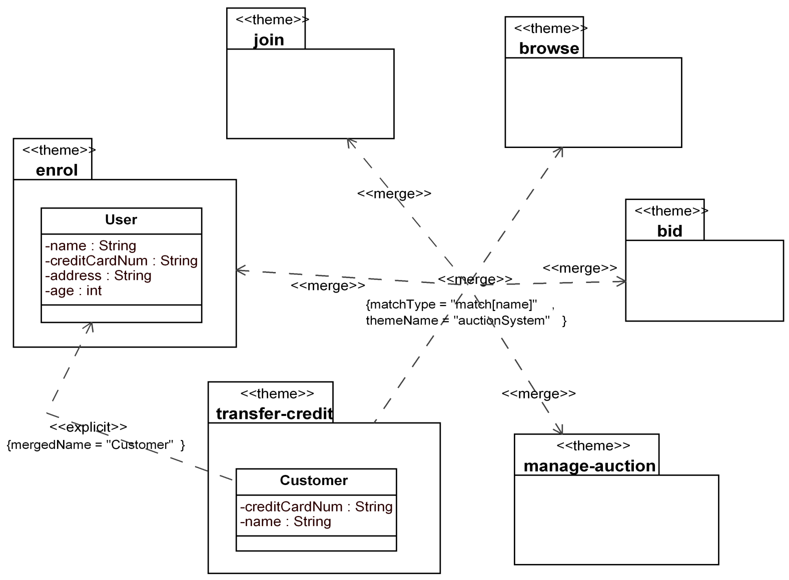


Figure 7.10: Base Merge

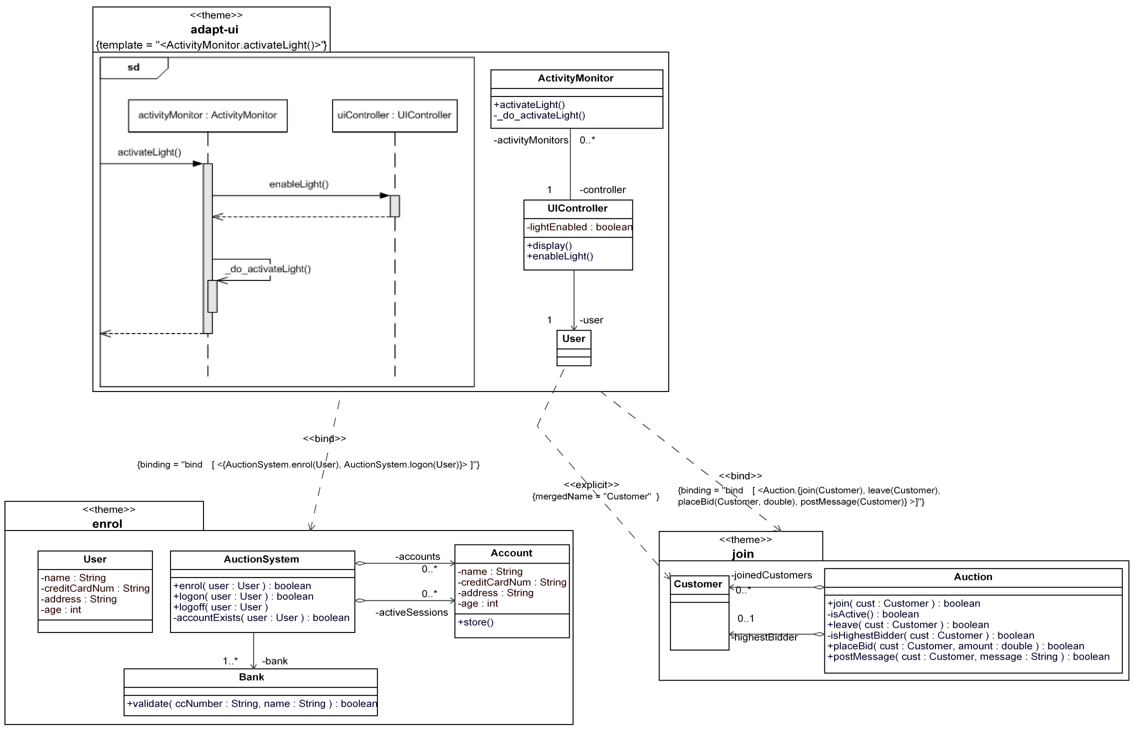
7.4.1 Merging Application Concerns

The first composition relationship that is applied in this case-study is a `merge`. A merge integration strategy allows multiple base themes that have been designed independently to be unified, with reconciliations specifying the resolution of overlapping or shared concepts. As illustrated in Figure 7.10, the themes that participate in this merge are `enrol`, `join`, `browse`, `bid`, `manage-auction` and `transfer-credit`. The name of the merged theme is labeled as `auctionSystem` through the `themeName` tag definition. For clarity, the rest of the structure and behaviour of the themes have been removed except for the overlapping concepts. It is possible in most graphical uml editors to create multiple views of the same underlying model. Observing the different designs of the separate base themes reveals designers used the same vocabulary to model the same concepts, so a `match[name]` matching criterion is attached to match elements that are the same name and type. In the MDA approach, requirements specifications are modelled with a CIM. One of the purposes of a CIM is to establish shared domain vocabulary for the PIM and PSMs. This simplifies the task of composition if agreed upon conventions have been established between individual designers or teams before hand. There is one interesting overlap that has occurred between the two themes `enrol` and `transfer-credit`. In the `enrol` theme the concept of a `User` is the same as a `Customer` in the `transfer-credit` theme, indicated by the explicit composition relationship. Looking at the requirements specification, the `enrol` theme deals with `User`s who have not yet registered with the system. A `User` once registered with system becomes a `Customer`. These concepts however are the same upon composition and so the overlap is reconciled.

7.4.2 Binding of Context-Aware Aspect Themes

At this stage, there are a number of base themes that have been indicated with composition and a number of aspect themes that have yet to be composed. To compose these aspect themes, it is necessary to apply another composition relationship defined by Theme/UML. A `bind` is defined as a specialisation of a merge integration and supports the merging of structure and behaviour of an aspect theme with a base theme in an asymmetric way. The sequence diagrams in an aspect theme specifies *h*ow (advice) and *w*hen (joinpoint) the crosscutting takes place. Upon composition, the binding indicates *w*hat is being crosscut i.e. the pointcuts. The binding identifies base operations directly on name matching or through name selection with wildcards. An aspect theme can be instantiated multiple times through numerous bindings on multiple themes. In such a way, a context-aware concern can have effect throughout the whole application logic but its main functionality, separated in a theme, is both better localised and more cohesive.

7.4.3 Adapt UI Theme Binding

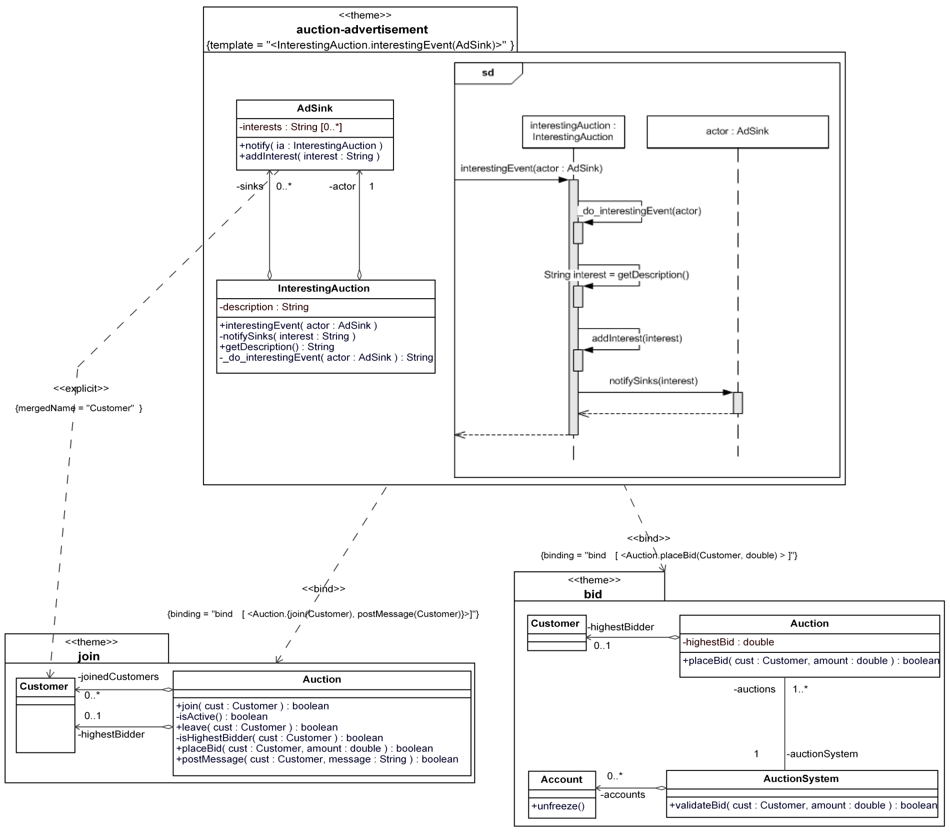


<<theme>>
adapt-ui
{template = "<ActivityMonitor.activateLight()>"}
sd
activityMonitor : ActivityMonitor
uiController : UIController
activateLight()
enableLight()
_do_activateLight()
ActivityMonitor
+activateLight()
-_do_activateLight()
-activityMonitors
0..*
1
-controller
UIController
-lightEnabled : boolean
+display()
+enableLight()
1
-user
User
<<bind>>
{binding = "bind [<{AuctionSystem.enrol(User), AuctionSystem.logon(User)}>]"}
<<explicit>>
{mergedName = "Customer" }
<<bind>>
{binding = "bind [<Auction.{join(Customer), leave(Customer), placeBid(Customer, double), postMessage(Customer)} >]"}
<<theme>>
enrol
User
-name : String
-creditCardNum : String
-address : String
-age : int
AuctionSystem
+enrol(user : User) : boolean
+logon(user : User) : boolean
+logoff(user : User)
-accountExists(user : User) : boolean
-accounts
0..*
0..*
-activeSessions
Account
-name : String
-creditCardNum : String
-address : String
-age : int
+store()
1..*
-bank
Bank
+validate(ccNumber : String, name : String) : boolean
<<theme>>
join
Customer
-joinedCustomers
0..*
0..1
-highestBidder
Auction
+join(cust : Customer) : boolean
-isActive() : boolean
+leave(cust : Customer) : boolean
-isHighestBidder(cust : Customer) : boolean
+placeBid(cust : Customer, amount : double) : boolean
+postMessage(cust : Customer, message : String) : boolean

Figure 7.11: Adapt UI Theme Binding

Figure 7.11 illustrates the instantiation of the Adapt UI theme, indicating the activities which should be associated with activating the UI Light, making it context-aware. The joinpoint that is represented as a template in the Adapt UI theme, `ActivityMonitor.activityMonitor()`, acts as a placeholder for the operations identified through the bindings. This aspect theme takes effect in the `enrol` theme, on the operations `AuctionSystem.enrol()` and `AuctionSystem.logon()`. It also takes effect over the operations `Auction.leave()`, `Auction.join()`, `Auction.placeBid()` and `Auction.postMessage()`. These points indicate when the light should be activated by the `UIController` As a specialisation of a `merge`, explicit composition relationships can also be applied. This is illustrated by the explicit between the `User` and `Customer` classes and is being resolved to `Customer`.

7.4.4 Auction Advertisement Theme Binding



<<theme>>
auction-advertisement
{template = "<InterestingAuction.interestingEvent(AdSink)>" }
sd
AdSink
-interests : String [0..*]
+notify(ia : InterestingAuction)
+addInterest(interest : String)
-sinks 0..*
-actor 1
InterestingAuction
-description : String
+interestingEvent(actor : AdSink)
-notifySinks(interest : String)
+getDescription() : String
-_do_interestingEvent(actor : AdSink) : String
interestingAuction : InterestingAuction
actor : AdSink
interestingEvent(actor : AdSink)
_do_interestingEvent(actor)
String interest = getDescription()
addInterest(interest)
notifySinks(interest)
<<explicit>>
{mergedName = "Customer" }
<<bind>>
{binding = "bind [<Auction.{join(Customer), postMessage(Customer)}>]"}
<<bind>>
{binding = "bind [<Auction.placeBid(Customer, double) >]"}
<<theme>>
join
Customer
-joinedCustomers
0..*
0..1
-highestBidder
Auction
+join(cust : Customer) : boolean
-isActive() : boolean
+leave(cust : Customer) : boolean
-isHighestBidder(cust : Customer) : boolean
+placeBid(cust : Customer, amount : double) : boolean
+postMessage(cust : Customer, message : String) : boolean
<<theme>>
bid
Customer
-highestBidder
0..1
Auction
-highestBid : double
+placeBid(cust : Customer, amount : double) : boolean
-auctions 1..*
1 -auctionSystem
Account
+unfreeze()
0..*
-accounts
AuctionSystem
+validateBid(cust : Customer, amount : double) : boolean

Figure 7.12: Auction Advertisement Theme Binding

Figure 7.12 illustrates the instantiation of the Auction Advertisement theme. This context-aware concern collects auction activity for each user and notifies the user if an activity occurs on an auction that they may be interested in, based on past activities. In the `join` theme, the operations `Auction.join()` and `Auction.postMessage()` and in the `bid` theme, `Auction.placeBid()`, represent both the points at which activities can be collected by the context-aware theme for a user and the points which are of interest to the user to be notified about. There is an explicit composition relationship between AdSink and Customer, indicating that the two are identical and are merged as the same.

7.4.5 Location Theme Binding

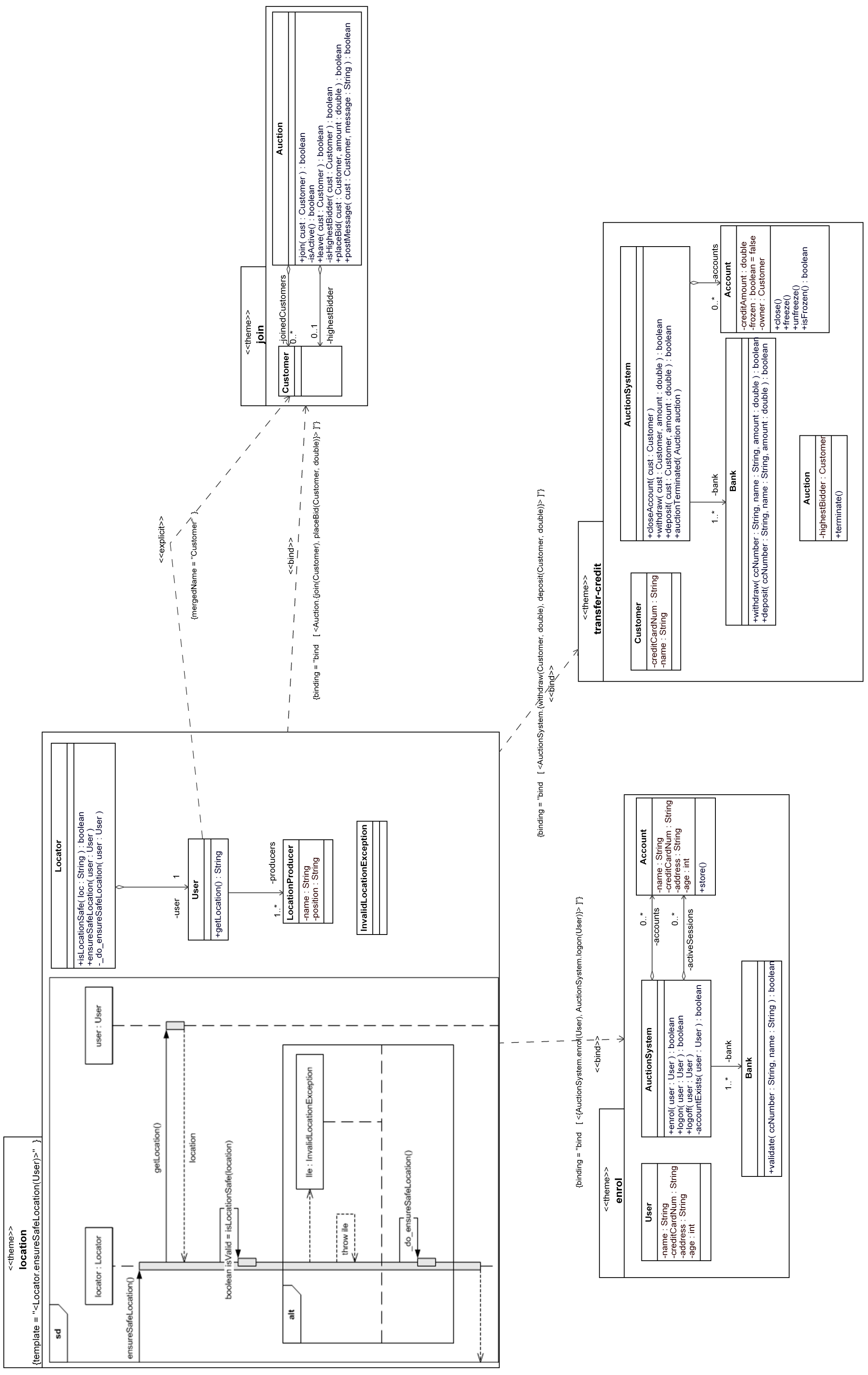
Figure 7.13 demonstrates the bindings of the location theme to the base themes enrol, transfer-credit and join. The location theme ensures that an operation that is triggered is safe before that operation can proceed. This is important if a user is in an unsafe or untrusted location, which is undesirable if making an important transaction. The points in the base themes that identify when this operation is checked is as follows. In the enrol theme the pointcuts AuctionSystem.enrol() and AuctionSystem.logon(), forbidding anyone in a dodgy location the ability to log in to the system until they are safe. In the transfer-credit theme the pointcuts are AuctionSystem.withdraw() and AuctionSystem.deposit(), which obviously encapsulate important money transaction behaviour. In the join theme the pointcuts are Auction.join() and Auction.placeBid() need to be secured to ensure that a user can't join and read the message board or place bids on an auction, until he is in a safe location.

7.5 Object-Oriented Composition

After the composition specification for the case-study is applied the next step is to move onto phase two (c.f. Figure 5.4) and compose both the base and aspect themes into a single object-oriented merged design. The tool takes the models with the composition relationships and merges them according to their semantics. This section will give a number of examples illustrating how the tool works and the results of the composition. The examples include the composition of the base, the composition of the adapt-ui aspect theme with the enrol theme, the composition of the auction-advertisement aspect theme with the bid theme and the composition of the location aspect theme with the join theme. To demonstrate how aspect interference is dealt with, the composition of adapt-ui and auction-advertisement themes with the join base themes will be illustrated as both aspect themes share two pointcuts with this base theme.

7.5.1 Result of Base Merge

Figure 7.14 shows the result of the composition of the base merge that has been applied to the enrol, join, browse, bid, manage-auction and transfer-credit themes in Figure 7.10. As illustrated, the artefacts have been merged based on name and type indicated by the match[name] criteria. From observation, the classes that were shared among multiple themes have been unified e.g. the operations in the resultant merge of the same class have all the operations belonging to those of the separate classes combined. Also, there is no User class as it has been merged with its new name Customer, according to the explicit composition relationship.



<<theme>>
location
{template = "<Locator.ensureSafeLocation(User)>" }
sd
locator : Locator
user : User
ensureSafeLocation()
getLocation()
location
boolean isValid = isLocationSafe(location)
alt
Ile : InvalidLocationException
throw ile
_do_ensureSafeLocation()
Locator
+isLocationSafe(loc : String) : boolean
+ensureSafeLocation(user : User)
-_do_ensureSafeLocation(user : User)
-user
1
User
+getLocation() : String
-producers
1..*
LocationProducer
-name : String
-position : String
InvalidLocationException
<<explicit>>
{mergedName = "Customer" }
<<bind>>
{binding = "bind [<Auction.{join(Customer), placeBid(Customer, double)}>]"}
<<theme>>
join
Customer
-joinedCustomers
0..*
0..1
-highestBidder
Auction
+join(cust : Customer) : boolean
-isActive() : boolean
+leave(cust : Customer) : boolean
-isHighestBidder(cust : Customer) : boolean
+placeBid(cust : Customer, amount : double) : boolean
+postMessage(cust : Customer, message : String) : boolean
<<bind>>
{binding = "bind [<AuctionSystem.{withdraw(Customer, double), deposit(Customer, double)}>]"}
<<theme>>
transfer-credit
Customer
-creditCardNum : String
-name : String
AuctionSystem
+closeAccount(cust : Customer)
+withdraw(cust : Customer, amount : double) : boolean
+deposit(cust : Customer, amount : double) : boolean
+auctionTerminated(Auction auction)
-bank
1..*
Bank
+withdraw(ccNumber : String, name : String, amount : double) : boolean
+deposit(ccNumber : String, name : String, amount : double) : boolean
-accounts
0..*
Account
-creditAmount : double
-frozen : boolean = false
-owner : Customer
+close()
+freeze()
+unfreeze()
+isFrozen() : boolean
Auction
-highestBidder : Customer
+terminate()
<<bind>>
{binding = "bind [<{AuctionSystem.enrol(User), AuctionSystem.logon(User)}>]"}
<<theme>>
enrol
User
-name : String
-creditCardNum : String
-address : String
-age : int
AuctionSystem
+enrol(user : User) : boolean
+logon(user : User) : boolean
+logoff(user : User)
-accountExists(user : User) : boolean
-accounts
0..*
-activeSessions
0..*
Account
-name : String
-creditCardNum : String
-address : String
-age : int
+store()
-bank
1..*
Bank
+validate(ccNumber : String, name : String) : boolean

Figure 7.13: Location Theme Binding

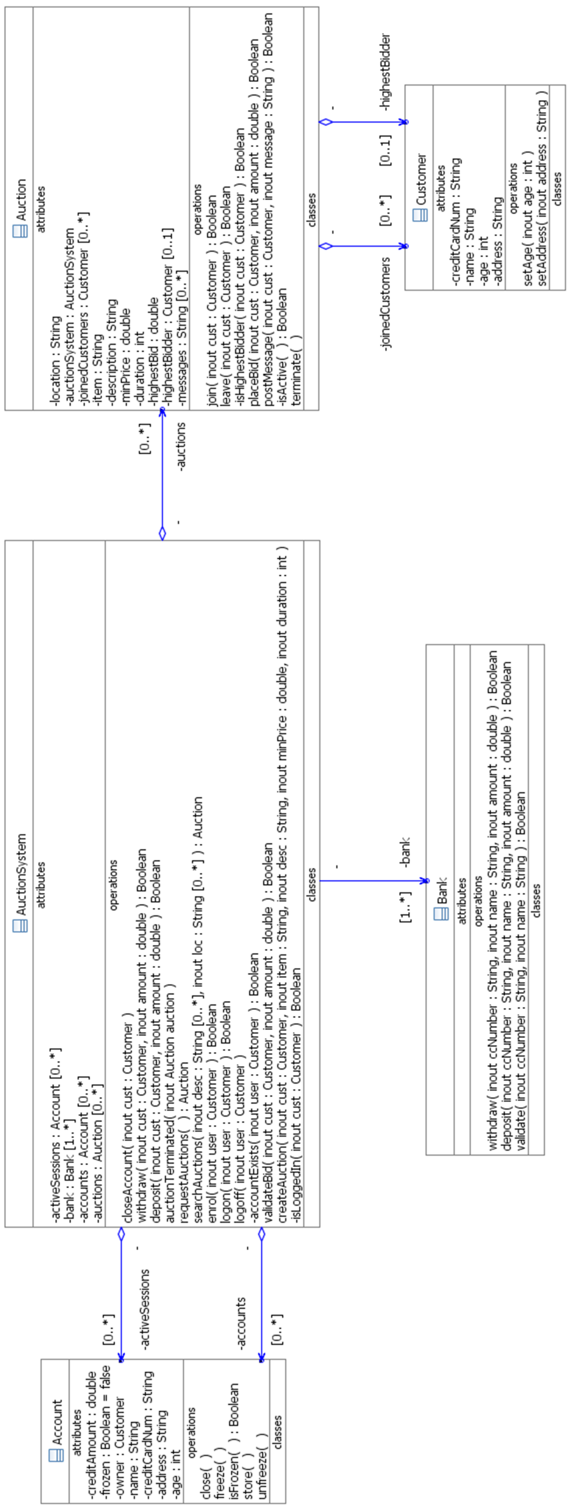

Auction
attributes
-location : String
-auctionSystem : AuctionSystem
-joinedCustomers : Customer [0..*]
-item : String
-description : String
-minPrice : double
-duration : int
-highestBid : double
-highestBidder : Customer [0..1]
-messages : String [0..*]
operations
join(inout cust : Customer) : Boolean
leave(inout cust : Customer) : Boolean
-isHighestBidder(inout cust : Customer) : Boolean
placeBid(inout cust : Customer, inout amount : double) : Boolean
postMessage(inout cust : Customer, inout message : String) : Boolean
-isActive( ) : Boolean
terminate( )
classes
Customer
attributes
-creditCardNum : String
-name : String
-age : int
-address : String
operations
setAge(inout age : int)
setAddress(inout address : String)
classes
-joinedCustomers
[0..*]
-
[0..1]
-highestBidder
-
[0..*]
-auctions
-
AuctionSystem
attributes
-activeSessions : Account [0..*]
-bank : Bank [1..*]
-accounts : Account [0..*]
-auctions : Auction [0..*]
operations
closeAccount(inout cust : Customer)
withdraw(inout cust : Customer, inout amount : double) : Boolean
deposit(inout cust : Customer, inout amount : double) : Boolean
auctionTerminated(inout Auction auction)
requestAuctions( ) : Auction
searchAuctions(inout desc : String [0..*], inout loc : String [0..*]) : Auction
enrol(inout user : Customer) : Boolean
logon(inout user : Customer) : Boolean
logoff(inout user : Customer)
-accountExists(inout user : Customer) : Boolean
validateBid(inout cust : Customer, inout amount : double) : Boolean
createAuction(inout cust : Customer, inout item : String, inout desc : String, inout minPrice : double, inout duration : int)
-isLoggedIn(inout cust : Customer) : Boolean
classes
-
-bank
[1..*]
Bank
attributes
operations
withdraw(inout ccNumber : String, inout name : String, inout amount : double) : Boolean
deposit(inout ccNumber : String, inout name : String, inout amount : double) : Boolean
validate(inout ccNumber : String, inout name : String) : Boolean
classes
Account
attributes
-creditAmount : double
-frozen : Boolean = false
-owner : Customer
-name : String
-creditCardNum : String
-address : String
-age : int
operations
close( )
freeze( )
isFrozen( ) : Boolean
store( )
unfreeze( )
classes
[0..*]
-activeSessions
-
-accounts
[0..*]
-

Figure 7.14: Merged Base

7.5.2 Aspect Merge of Adapt UI and Enroll Themes

Figure 7.15 and 7.16 illustrate the result of the composition of the aspect context-aware `adapt-ui` theme with the base theme `enrol` specified in Figure 7.11. From the merge, the `ActivityMonitor` gets merged with the `AuctionSystem` through the binding. The `User` class also gets merged as `Customer` through the base merge (the semantics of the tool always merges the base first). The `logon` and `enrol` operations get renamed to `_do_logon` and `_do_enrol` respectively, along with encapsulating any behaviour they encapsulate. The new `logon` and `enrol` operations now contain the crosscutting operations that have been merged as illustrated by the sequence diagrams in Figure 7.11.

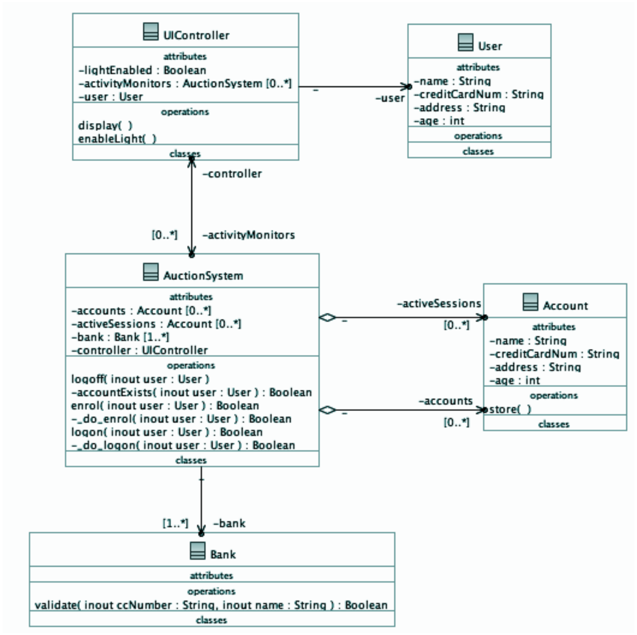
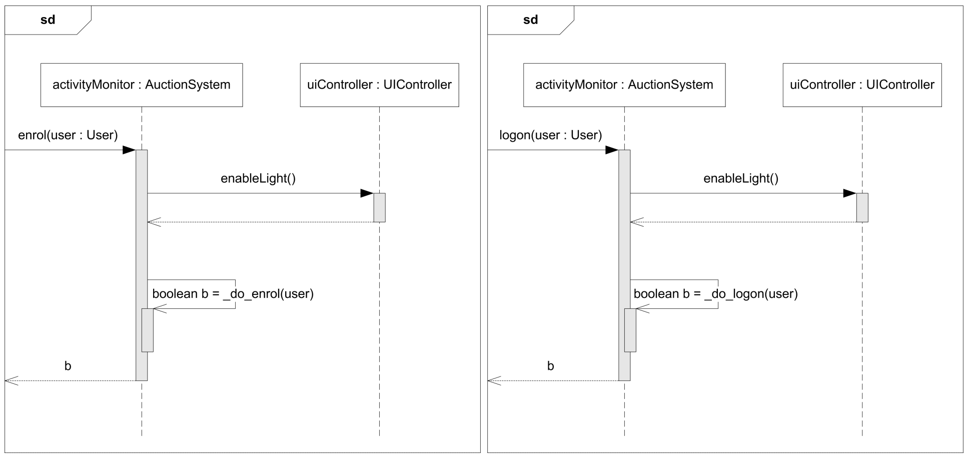


Figure 7.15: Result of Adapt UI and Enroll Theme Merge

7.5.3 Aspect Merge of Auction Advertisement and Bid Themes

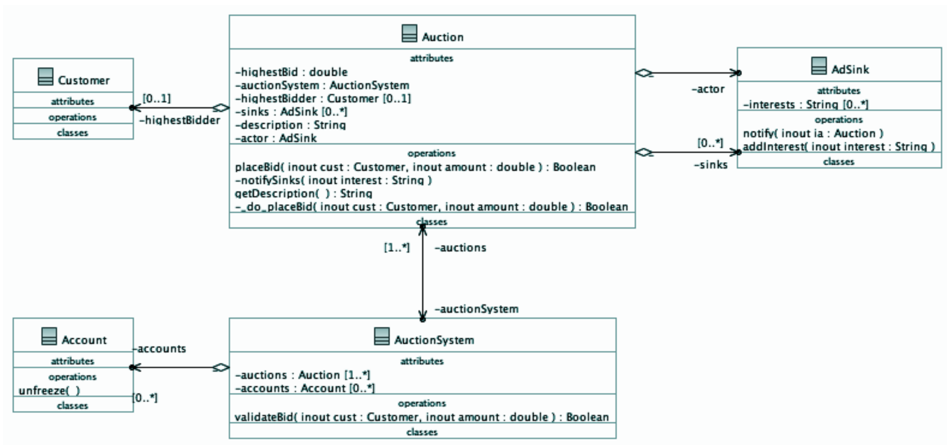
Figure 7.17 and 7.18 illustrate the result of the composition of the aspect context-aware `auction-advertisement` theme with the base theme `bid` specified in Figure 7.12 theme. There is one pointcut specified by the binding from the aspect theme to the base theme `Auction.placeBid()`. As a result of the merge, the `InterestingAuction` class is merged with the `Auction` class through the binding and the `AdSink` is merged with the `Customer` class through the explicit composition relationship on the `join` theme. As a side note, the explicit composition relationship is only applied to only one theme per reconciliation. It is quite possible to change that to an n-ary relationship so it covers both themes and it is probably a good idea to do so for clarity. However, this does not change the overall effect of the merge, because as mentioned in the design chapter, the explicit compositions



sd
activityMonitor : AuctionSystem
uiController : UIController
enrol(user : User)
enableLight()
boolean b = _do_enrol(user)
b
sd
activityMonitor : AuctionSystem
uiController : UIController
logon(user : User)
enableLight()
boolean b = _do_logon(user)
b

Figure 7.16: Result of Adapt UI and Enroll Theme Merge Sequence Diagram

get resolved first, then the base gets merged. The behaviour of the new `placeBid` operation which crosscuts the old is indicated by the sequence diagram that got merged.



Auction
attributes
-highestBid : double
-auctionSystem : AuctionSystem
-highestBidder : Customer [0..1]
-sinks : AdSink [0..*]
-description : String
-actor : AdSink
operations
placeBid(inout cust : Customer, inout amount : double) : Boolean
-notifySinks(inout interest : String)
getDescription( ) : String
-_do_placeBid(inout cust : Customer, inout amount : double) : Boolean
classes
Customer
attributes
operations
classes
[0..1]
-highestBidder
AdSink
attributes
-interests : String [0..*]
operations
notify(inout ia : Auction)
addInterest(inout interest : String)
classes
-actor
[0..*]
-sinks
[1..*]
-auctions
-auctionSystem
Account
attributes
operations
unfreeze( )
classes
-accounts
[0..*]
AuctionSystem
attributes
-auctions : Auction [1..*]
-accounts : Account [0..*]
operations
validateBid(inout cust : Customer, inout amount : double) : Boolean
classes

Figure 7.17: Result of Auction Advertisement and Bid Theme Merge

7.5.4 Aspect Merge of Location and Join Themes

Figure 7.19 and 7.20 illustrate the result of the composition specification applied to the aspect theme `location` with the base theme `join` illustrated in Figure 7.13. As a result of the aspect merge, the `User` class in the aspect theme gets merged with the `Customer` class through the explicit composition relationship. Likewise, the `Locator` class gets merged with `Auction` class through the template binding. Two pointcuts are identified and are replaced with their crosscutting behaviour as illustrated in 7.20. Due to space constraints, only the `join` operation behaviour is illustrated. The `placeBid` behaviour has a similar format.

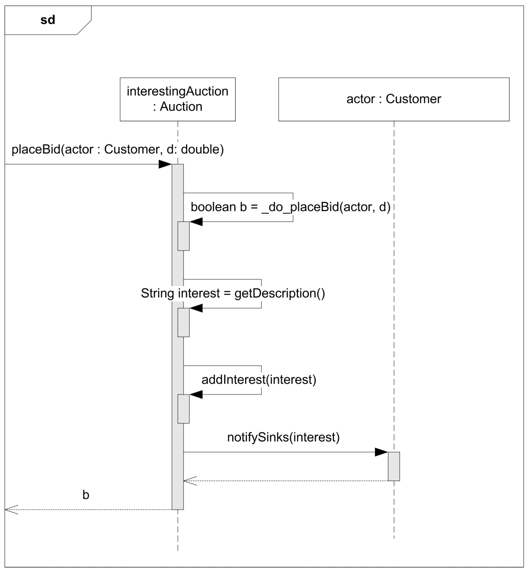

sd
interestingAuction : Auction
actor : Customer
placeBid(actor : Customer, d: double)
boolean b = _do_placeBid(actor, d)
String interest = getDescription()
addInterest(interest)
notifySinks(interest)
b

Figure 7.18: Result of Auction Advertisement and Bid Theme Merge Sequence Diagram

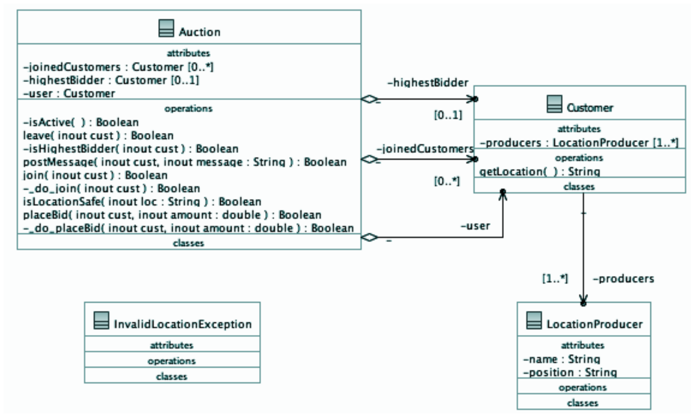

Auction
attributes
-joinedCustomers : Customer [0..*]
-highestBidder : Customer [0..1]
-user : Customer
operations
-isActive( ) : Boolean
leave(inout cust) : Boolean
-isHighestBidder(inout cust) : Boolean
postMessage(inout cust, inout message : String) : Boolean
join(inout cust) : Boolean
-_do_join(inout cust) : Boolean
isLocationSafe(inout loc : String) : Boolean
placeBid(inout cust, inout amount : double) : Boolean
-_do_placeBid(inout cust, inout amount : double) : Boolean
classes
-highestBidder
[0..1]
-joinedCustomers
[0..*]
-user
Customer
attributes
-producers : LocationProducer [1..*]
operations
getLocation( ) : String
classes
[1..*]
-producers
InvalidLocationException
attributes
operations
classes
LocationProducer
attributes
-name : String
-position : String
operations
classes

Figure 7.19: Result of Location and Join Theme Merge

7.5.5 Aspect Interference of Adapt-UI and Auction-Advertisement Aspect Themes with Join Base Theme

Figure 7.11 and 7.12 specify the aspect compositions over the `join` base theme by the `adapt-ui` and `auction-advertisement` aspect themes respectively. From these diagrams, it is observed that their pointcut shadows intersect i.e. both the `Auction.join()` and `Auction.postMessage()` operations are crosscut by both themes. In such a circumstance, the tool composes the aspects with the base non-deterministically i.e. there is no way to specify which aspect is to get merged first. This is a research task for future work, to determine the best way to handle aspect interference in Theme/UML. Figure 7.21 and 7.22 demonstrate the merged result of both these aspect themes on the `join` theme. The

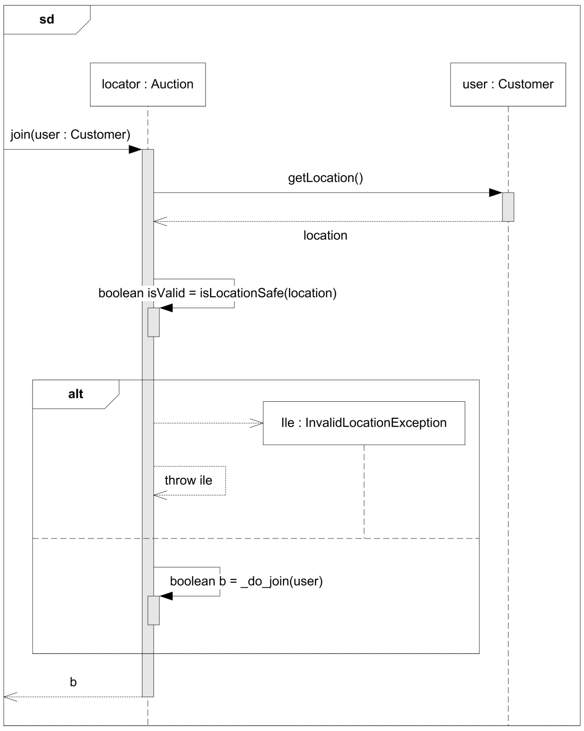

sd
locator : Auction
user : Customer
join(user : Customer)
getLocation()
location
boolean isValid = isLocationSafe(location)
alt
Ile : InvalidLocationException
throw ile
boolean b = _do_join(user)
b

Figure 7.20: Result of Location and Join Theme Merge Sequence Diagram

sequence diagrams only illustrate the interference for one crosscutting operation `join`. As indicated in 7.22, the first theme that gets merged in this example is the `auction-advertisement` theme. The `join()` method gets merged and renamed as usual, with the crosscutting behaviour of that theme now encapsulated in that method. The pointcut gets updated for the `adapt-ui` theme, in which it is now pointing at the `_do_join()` method. The `adapt-ui` aspect theme now gets composed into this. The `_do_join()` method gets renamed to `_do__do_join()` and the `_do_join()` method now encapsulates the behaviour for the `adapt-ui` aspect theme. The _do__do_join() encapsulates the original `join` behaviour before both aspect merges.

7.6 Mapping of Design to PSMs

For brevity, the case-study assumes (and exaggerates) a straight waterfall process, so the design and composition specifications are correct the first time around. The next step according to the process (c.f. Figure 5.4 is to re-factor the composition if any corrections need to be made as explained in the design chapter. The case-study has no faults so the next step moves the process into phase three. This step begins by selecting the platform for transformation. In this approach, the two platforms, .NET CF and J2ME will be the target platforms. Successively, the designer transforms the design into these two target PSMs and then further elaborates them, adding in more concrete detail for the platform.

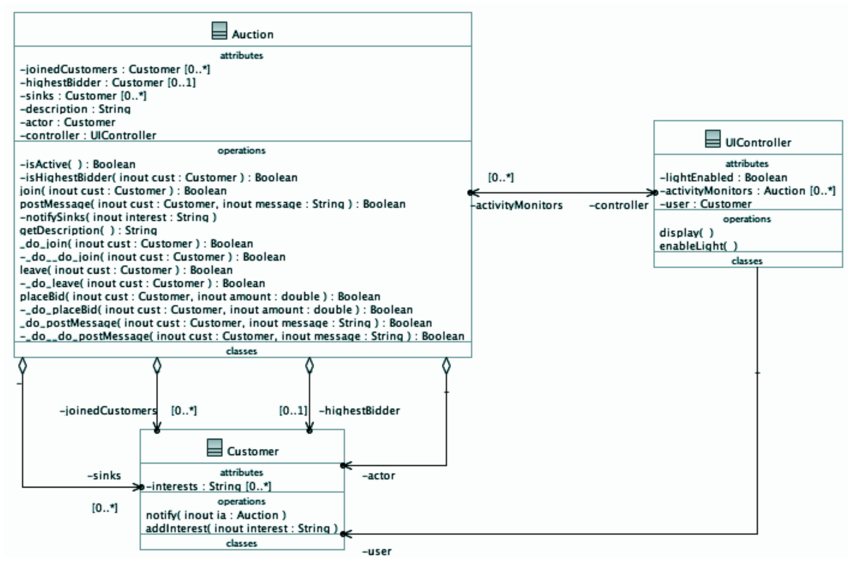

Auction
attributes
-joinedCustomers : Customer [0..*]
-highestBidder : Customer [0..1]
-sinks : Customer [0..*]
-description : String
-actor : Customer
-controller : UIController
operations
-isActive( ) : Boolean
-isHighestBidder(inout cust : Customer) : Boolean
join(inout cust : Customer) : Boolean
postMessage(inout cust : Customer, inout message : String) : Boolean
-notifySinks(inout interest : String)
getDescription( ) : String
_do_join(inout cust : Customer) : Boolean
-_do__do_join(inout cust : Customer) : Boolean
leave(inout cust : Customer) : Boolean
-_do_leave(inout cust : Customer) : Boolean
placeBid(inout cust : Customer, inout amount : double) : Boolean
-_do_placeBid(inout cust : Customer, inout amount : double) : Boolean
_do_postMessage(inout cust : Customer, inout message : String) : Boolean
-_do__do_postMessage(inout cust : Customer, inout message : String) : Boolean
classes
UIController
attributes
-lightEnabled : Boolean
-activityMonitors : Auction [0..*]
-user : Customer
operations
display( )
enableLight( )
classes
[0..*]
-activityMonitors
-controller
-joinedCustomers
[0..*]
[0..1]
-highestBidder
Customer
attributes
-interests : String [0..*]
operations
notify(inout ia : Auction)
addInterest(inout interest : String)
classes
-sinks
[0..*]
-actor
-user

Figure 7.21: Result of Aspect Interference

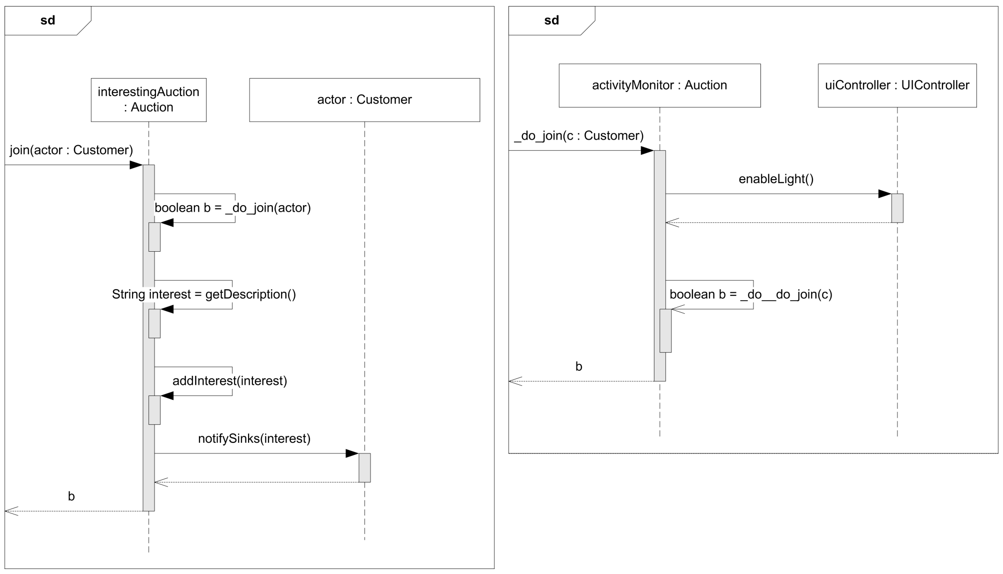

sd
interestingAuction : Auction
actor : Customer
join(actor : Customer)
boolean b = _do_join(actor)
String interest = getDescription()
addInterest(interest)
notifySinks(interest)
b
sd
activityMonitor : Auction
uiController : UIController
_do_join(c : Customer)
enableLight()
boolean b = _do__do_join(c)
b

Figure 7.22: Result of Aspect Interference Sequence Diagram

The unelaborated .NET CF PSM and J2ME PSM are displayed in Appendix I and II respectively. Once finished, the designer again, successively transforms the PSM into code.

7.7 Evaluation

Chapter 1 presented three challenges and difficulties with designing mobile, context-aware applications as the motivation for this work. Firstly, as discussed, current software engineering guidelines and principles are inadequate for handling context. Secondly, designing with context necessitates better abstractions to think and reason about them. Thirdly, the distributed nature of many interacting devices coupled with the wide range of heterogeneous devices that context-aware computing introduces make designs very complex.

From these motivating challenges, the following subsections will discuss to what end these goals have been achieved with this new approach, using the experience of this case-study as an example.

7.7.1 Guidelines for handling context

As discussed in Chapter 1, concerns relating to mobile, context-aware adaptation are crosscutting. Traditional paradigms such as object-orientation offer insufficient guidelines for handling these types of concerns, resulting in them manifesting in the forms of scattering and tangling. These adverse effects resulting from poor concern localisation have been proven to reduce maintainability, comprehensibility and manageability in mobile, context-aware applications [39]. From analysing this case-study, it is apparent that AOSD has a favourable impact on designing mobile, context-aware concerns, effectively reducing the scattering of these concerns with other concerns and making them less tangled with others. From the requirements specification, it can be observed that the context-aware concerns in the case-study have inherent relationships with other requirements. In object-orientation, implementing these concerns would results in their logic being scattered throughout multiple classes and tangled with other concerns. By using Theme/UML, each requirement or set of similar requirements can be designed in one theme. The themes that are crosscutting can be designed in aspect themes and then the crosscutting can be applied at composition time. This reduces the effects of scattering throughout the application as the concern is localised to the aspect theme. Likewise, the effect of tangling is removed from the design, as each requirement is aligned with a single theme and does not affect the others when designing. However, the difficulty in this approach comes at composition time, when the designer needs to be able reason about how the themes are to be composed together. Although the clear cut separation of context-aware concerns is an advantage, the composition adds extra complexities, as illustrated by the case-study. The context-aware concerns were easy to design in separation but at composition it was necessary to understand the themes fully to be able to apply the pointcuts. The mobile, context-aware concerns in the case-study appeared to be non-orthogonal. Some of the aspects themes interfered with each other at composition time by sharing pointcuts, which reduced comprehensibility of the overall application. Tool support for analysing and applying precedence when aspect interference occurs would be worthwhile in alleviating this problem.

7.7.2 Abstractions for modelling context

An abstraction in computer science allows a concept to be represented or modelled by hiding the lower-level details so the concept can be better reasoned about. As illustrated in the case-study, Theme/UML was used as a domain-specific PIM where the context-aware concerns could be modelled in a way that abstracted the other concerns that it interacted with. This provided a powerful abstraction over previous approaches by allowing concerns based on context adaptation to be designed without having to worry about the details of its interaction with the rest of the application. Here the combination of AOSD and MDA can be seen. The advantages of using an aspect-oriented modelling language, Theme/UML, as a model-driven PIM gave the designer better support and abstractions for thinking about context in a platform independent way. The composition process merged the context-aware concerns into an object-oriented representation allowing the designer to transform this PIM into a more refined PSM. The designer could then elaborate the details of the platform at that layer of abstraction and then transform the PSM to code.

7.7.3 Design support for many interacting heterogeneous mobile devices

Model-driven architecture is naturally suited for dealing with commonalities between designs for many platforms. The aim of MDA is to design once and use mappings that encapsulate domain-specific knowledge to transform the design into one that is suitable for many different platforms. From the case-study, a mobile, context-aware application was designed in a platform independent way and then that design was used to transform to two target platforms, .NET CF and J2ME. This demonstrated that a single design can be specified in a platform-independent way and be suitable for many different mobile devices. While the mappings in this approach were quite trivial, encapsulating only data types, it is possible to design more powerful model compilers that encapsulate more detailed domain-specific knowledge about the target platform, and marks could be used on the PIM to guide better transformations. Powerful model compilers that lean towards the more translation-oriented MDA view are very expensive and time consuming to develop but offer great power where the target platform will be in use for some time. However, the elaborational MDA view favours better vertical separation of concerns with less development in the design of model compilers. Overall, the case-study demonstrated that this approach is a worthwhile endeavor for designs spanning multiple mobile devices.

7.8 Summary

The chapter presented a mobile, context-aware case-study to evaluate the approach outlined in this dissertation. The case-study was used to evaluate against the three main challenges presented in Chapter 1 that motivated this work. The evaluation findings suggested that this approach was favourable for designing mobile, context-aware applications offering better guidelines for handling support, better abstractions and supporting the design on multiple different mobile devices.

Chapter 8

Conclusion and Future Work

8.1 Introduction

This chapter discusses the objectives of this dissertation and how they were met. It reflects on the work that was done and suggests some suitable future work.

8.2 Conclusion

The objective of this dissertation was to outline a new and better approach for the design of mobile, context-aware applications. The dissertation started by proposing a number of challenges that motivated this work. These challenges outlined the need for better guidelines and abstractions to think and handle context in design as well as support for designing in a platform-independent way to accommodate multiple heterogeneous mobile devices. To help manage these challenges and alleviate some of the complexities with designing for mobile, context-aware devices, this dissertation outlined a process that combined an implementation of an aspect-oriented modelling language called Theme/UML with MDA technologies. A state-of-the-art review was done on aspect-oriented model-driven weaving tools. From the insights gained from these, the design and implementation chapters defined a process and implementation that would be suitable for combining AOSD and MDA for the domain of mobile, context-aware computing. The contributions for this dissertation was to develop tool support for Theme/UML in the context of MDA, design and implement an object-oriented composition algorithm to map the Theme/UML constructs to an OO PIM and then write two mappings that would transform this OO PIM into two platforms for mobile devices, J2ME and .NET CF. A mobile, context-aware case-study was designed to test this approach. The case study presented the requirements specification of a distributed auction-system with support for mobile devices. It demonstrated how a domain-specific, mobile context-aware concern that emerges from a requirement specification can be modelled separately from the application concerns and then illustrated how it can be composed with them. The evaluation findings suggested that this approach was favourable for designing mobile, context-aware applications offering better guidelines for handling support, better abstractions and supporting the design on multiple different mobile devices.

8.3 Future Work

This section will outline some possible future work.

8.3.1 Model-Driven Theme/UML

This approach implemented Theme/UML as a model-driven solution. However, only the necessary Theme/UML semantics were implemented for the approach. One of the restrictions is that aspect themes can only specify one template sequence. Theme/UML supports the specification of multiple template sequences per aspect theme. While one template sequence is enough for most circumstances, specifying multiple template sequences are very handy and are occasionally needed. The composition metamodel would need to be extended so the `ReferenceSet` would reference the associated `ComposableInteraction`.

Reflecting on the current implementation, one point of interest is Theme/UML's dependency on text-based forms to specify elements both in the merge reconciliations and aspect bindings. This text-based parsing and searching is very tedious and error prone for the designer as well as for the implementor. This is illustrated by the grammars provided to parse these expressions in the implementation chapter. One possible re-implementation would be to remove these forms and to utilise a pure profile approach, where each element is marked instead of marking the container of the theme. A style like that suggested in [53] would suitable and more convenient. However the profile would have to align itself with definition of Theme/UML's software composition patterns and match the way the reconciliation and bindings are currently done.

Sequence diagrams in the UML 2.1 are very poorly defined and are unsuitable as behavioural diagrams for code generation. They are more used for elaboration purposes and not for specifying executable behaviour. Until the sequence diagrams have been fixed by the OMG, the possibility for another UML behavioural diagrams in Theme/UML is necessary for specifying crosscutting and code generation. Activity Diagrams or State Diagrams seem like a suitable alternative. These diagrams are more mature, with better tool support and better defined semantics for code generation. A future work would include dropping the support for sequence diagrams in favour of one of these diagrams.

8.3.2 Aspect Interference

Aspect Interference occurs when more than one aspect share common pointcuts. At these points, it is necessary to declare which aspect gets precedence. Theme/UML does not provide any support for specifying precedence on pointcuts or the order of aspect merges. Instead, it advises the aspects get incrementally merged at each iteration and the pointcut get readjusted before the next aspect merge. This approach is not very suitable for a designer wishing to design a full composition specification with many aspect themes. The current work merges the aspects non-deterministically because there is no way to declare precedence. An interesting future work would be to research the best and most suitable way to place precedence on a pointcut which is crosscut by multiple themes.

8.3.3 Metamodel of devices

Another interesting point work for mobile, context-aware computing involves creating a metamodel that defines the capabilities and requirements of a specific mobile device. Each mobile device can be defined with this metamodel, for example specifying its platform requirements, operating system, processor speed, memory and battery power. It would be interesting to observe how a mapping can be applied giving a distributed design and a definition of the devices that the design is being run on. The transformation could use these specific details to generate heuristics and the most optimised setup for the design. For example, given an aspect that encapsulates a very computationally heavy concern, the mapping could use the information of the devices to suggest that this aspect gets woven into a device that is most suitable, perhaps one with a powerful processor.

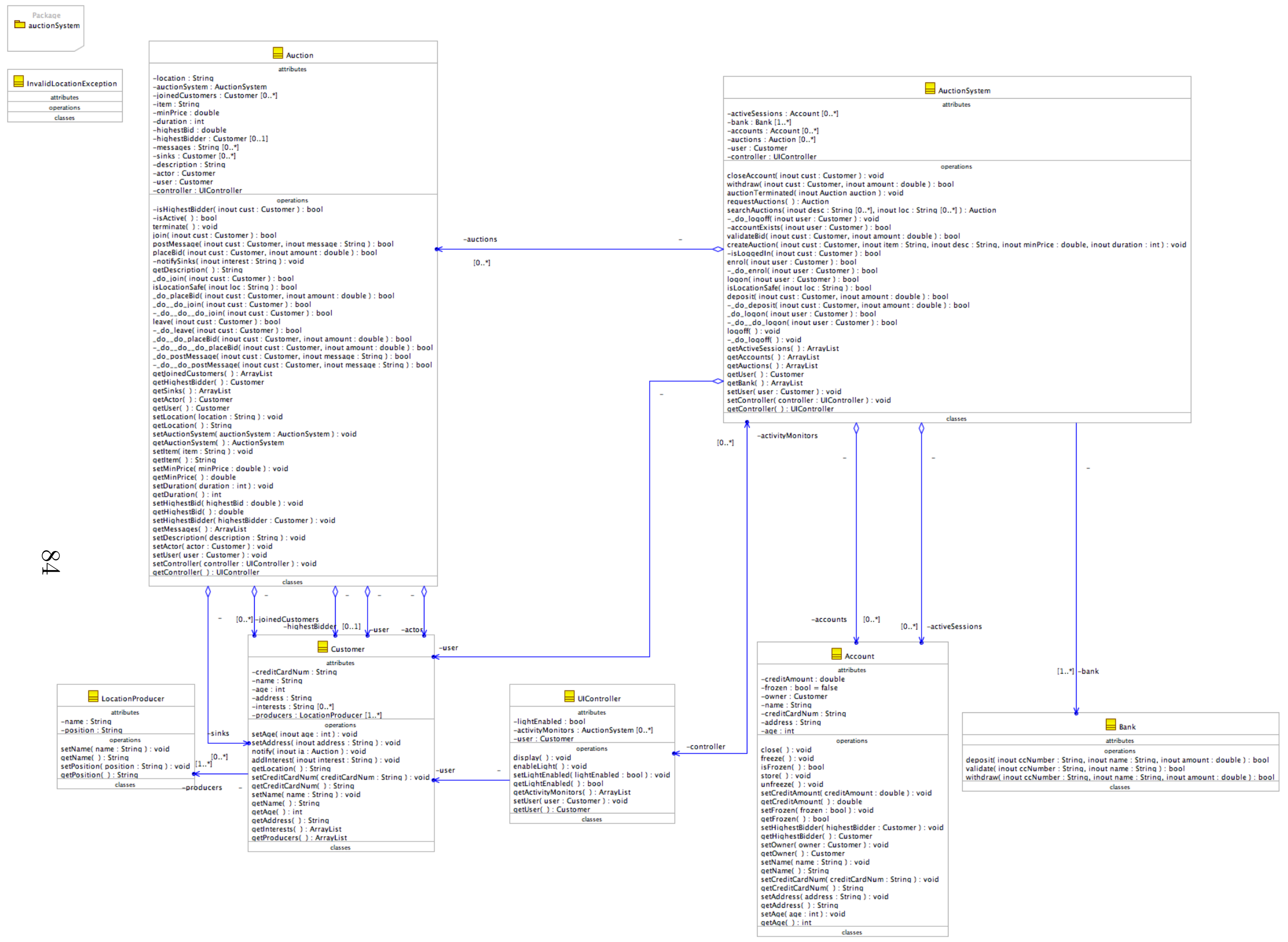
8.3.4 Repository of Mappings

The outlined approach implemented mappings to .NET CF and J2ME. There is a possibility for more mappings that would encompass future work. The more mappings to different devices the more capable the design is to be disseminated to more devices. The range of mappings is not limited to object-oriented mobile target platforms. Indeed this approach could be applied successfully to any domain. Mappings to aspect-oriented languages are also a point of future work.

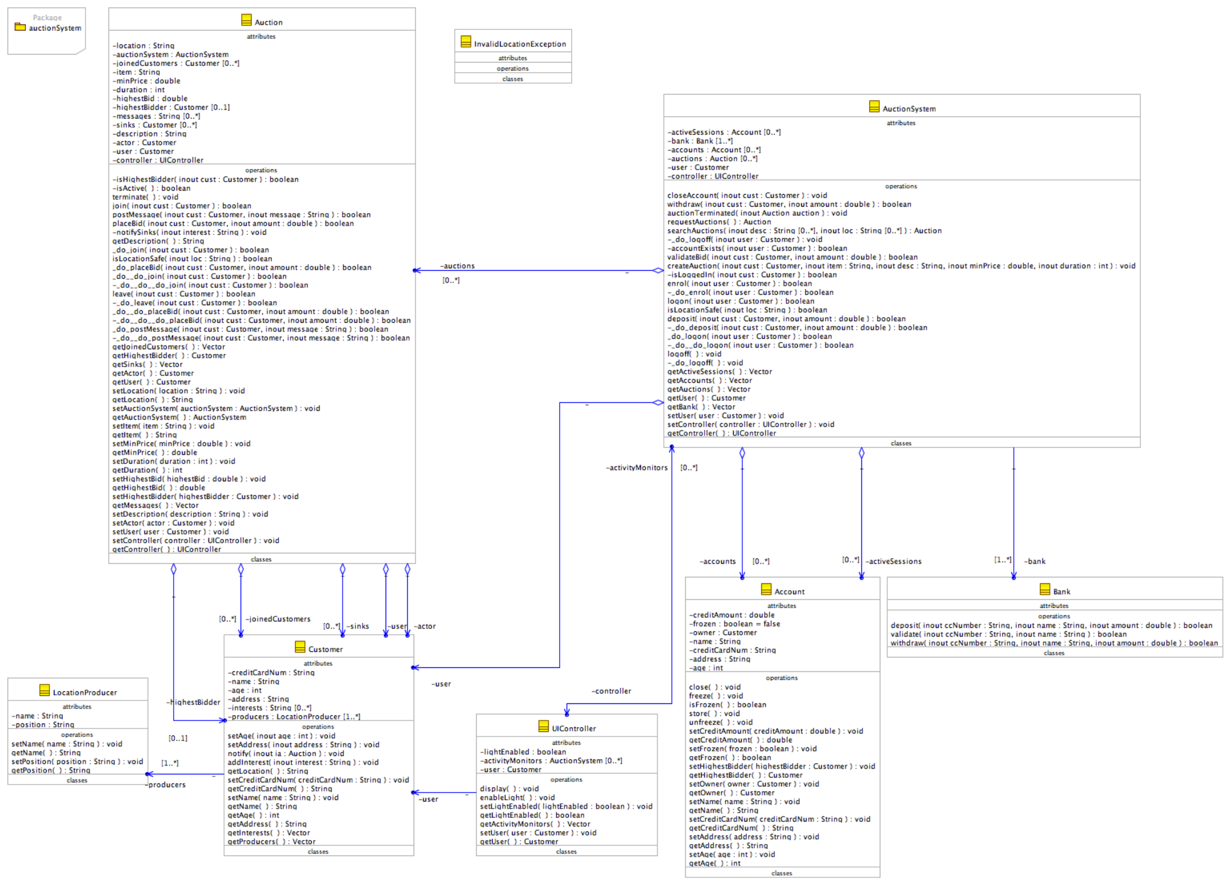
8.4 Summary

This chapter concluded this dissertation with some insights into the work and some suitable suggestions for future work.

Appendix I .NET CF PSM


Package
auctionSystem
InvalidLocationException
attributes
operations
classes
Auction
attributes
-location : String
-auctionSystem : AuctionSystem
-joinedCustomers : Customer [0..*]
-item : String
-minPrice : double
-duration : int
-highestBid : double
-highestBidder : Customer [0..1]
-messages : String [0..*]
-sinks : Customer [0..*]
-description : String
-actor : Customer
-user : Customer
-controller : UIController
operations
-isHighestBidder(inout cust : Customer) : bool
-isActive( ) : bool
terminate( ) : void
join(inout cust : Customer) : bool
postMessage(inout cust : Customer, inout message : String) : bool
placeBid(inout cust : Customer, inout amount : double) : bool
-notifySinks(inout interest : String) : void
getDescription( ) : String
_do_join(inout cust : Customer) : bool
isLocationSafe(inout loc : String) : bool
_do_placeBid(inout cust : Customer, inout amount : double) : bool
_do__do_join(inout cust : Customer) : bool
-_do__do__do_join(inout cust : Customer) : bool
leave(inout cust : Customer) : bool
-_do_leave(inout cust : Customer) : bool
_do__do_placeBid(inout cust : Customer, inout amount : double) : bool
-_do__do__do_placeBid(inout cust : Customer, inout amount : double) : bool
_do_postMessage(inout cust : Customer, inout message : String) : bool
-_do__do_postMessage(inout cust : Customer, inout message : String) : bool
getJoinedCustomers( ) : ArrayList
getHighestBidder( ) : Customer
getSinks( ) : ArrayList
getActor( ) : Customer
getUser( ) : Customer
setLocation(location : String) : void
getLocation( ) : String
setAuctionSystem(auctionSystem : AuctionSystem) : void
getAuctionSystem( ) : AuctionSystem
setItem(item : String) : void
getItem( ) : String
setMinPrice(minPrice : double) : void
getMinPrice( ) : double
setDuration(duration : int) : void
getDuration( ) : int
setHighestBid(highestBid : double) : void
getHighestBid( ) : double
setHighestBidder(highestBidder : Customer) : void
getMessages( ) : ArrayList
setDescription(description : String) : void
setActor(actor : Customer) : void
setUser(user : Customer) : void
setController(controller : UIController) : void
getController( ) : UIController
classes
AuctionSystem
attributes
-activeSessions : Account [0..*]
-bank : Bank [1..*]
-accounts : Account [0..*]
-auctions : Auction [0..*]
-user : Customer
-controller : UIController
operations
closeAccount(inout cust : Customer) : void
withdraw(inout cust : Customer, inout amount : double) : bool
auctionTerminated(inout Auction auction) : void
requestAuctions( ) : Auction
searchAuctions(inout desc : String [0..*], inout loc : String [0..*]) : Auction
-_do_logoff(inout user : Customer) : void
-accountExists(inout user : Customer) : bool
validateBid(inout cust : Customer, inout amount : double) : bool
createAuction(inout cust : Customer, inout item : String, inout desc : String, inout minPrice : double, inout duration : int) : void
-isLoggedIn(inout cust : Customer) : bool
enrol(inout user : Customer) : bool
-_do_enrol(inout user : Customer) : bool
logon(inout user : Customer) : bool
isLocationSafe(inout loc : String) : bool
deposit(inout cust : Customer, inout amount : double) : bool
-_do_deposit(inout cust : Customer, inout amount : double) : bool
_do_logon(inout user : Customer) : bool
-_do__do_logon(inout user : Customer) : bool
logoff( ) : void
-_do_logoff( ) : void
getActiveSessions( ) : ArrayList
getAccounts( ) : ArrayList
getAuctions( ) : ArrayList
getUser( ) : Customer
getBank( ) : ArrayList
setUser(user : Customer) : void
setController(controller : UIController) : void
getController( ) : UIController
classes
-auctions
[0..*]
-activityMonitors
[0..*]
-joinedCustomers
-highestBidder
[0..1]
-user
-actor
-user
Customer
attributes
-creditCardNum : String
-name : String
-age : int
-address : String
-interests : String [0..*]
-producers : LocationProducer [1..*]
operations
setAge(inout age : int) : void
setAddress(inout address : String) : void
notify(inout ia : Auction) : void
addInterest(inout interest : String) : void
getLocation( ) : String
setCreditCardNum(creditCardNum : String) : void
getCreditCardNum( ) : String
setName(name : String) : void
getName( ) : String
getAge( ) : int
getAddress( ) : String
getInterests( ) : ArrayList
getProducers( ) : ArrayList
classes
-sinks
[0..*]
[1..*]
-producers
LocationProducer
attributes
-name : String
-position : String
operations
setName(name : String) : void
getName( ) : String
setPosition(position : String) : void
getPosition( ) : String
classes
-user
UIController
attributes
-lightEnabled : bool
-activityMonitors : AuctionSystem [0..*]
-user : Customer
operations
display( ) : void
enableLight( ) : void
setLightEnabled(lightEnabled : bool) : void
getLightEnabled( ) : bool
getActivityMonitors( ) : ArrayList
setUser(user : Customer) : void
getUser( ) : Customer
classes
-controller
-accounts
[0..*]
[0..*]
-activeSessions
Account
attributes
-creditAmount : double
-frozen : bool = false
-owner : Customer
-name : String
-creditCardNum : String
-address : String
-age : int
operations
close( ) : void
freeze( ) : void
isFrozen( ) : bool
store( ) : void
unfreeze( ) : void
setCreditAmount(creditAmount : double) : void
getCreditAmount( ) : double
setFrozen(frozen : bool) : void
getFrozen( ) : bool
setHighestBidder(highestBidder : Customer) : void
getHighestBidder( ) : Customer
setOwner(owner : Customer) : void
getOwner( ) : Customer
setName(name : String) : void
getName( ) : String
setCreditCardNum(creditCardNum : String) : void
getCreditCardNum( ) : String
setAddress(address : String) : void
getAddress( ) : String
setAge(age : int) : void
getAge( ) : int
classes
[1..*]
-bank
Bank
attributes
operations
deposit(inout ccNumber : String, inout name : String, inout amount : double) : bool
validate(inout ccNumber : String, inout name : String) : bool
withdraw(inout ccNumber : String, inout name : String, inout amount : double) : bool
classes

Appendix II J2ME PSM


Package
auctionSystem
Auction
attributes
-location : String
-auctionSystem : AuctionSystem
-joinedCustomers : Customer [0..*]
-item : String
-minPrice : double
-duration : int
-highestBid : double
-highestBidder : Customer [0..1]
-messages : String [0..*]
-sinks : Customer [0..*]
-description : String
-actor : Customer
-user : Customer
-controller : UIController
operations
-isHighestBidder(inout cust : Customer) : boolean
-isActive( ) : boolean
terminate( ) : void
join(inout cust : Customer) : boolean
postMessage(inout cust : Customer, inout message : String) : boolean
placeBid(inout cust : Customer, inout amount : double) : boolean
-notifySinks(inout interest : String) : void
getDescription( ) : String
_do_join(inout cust : Customer) : boolean
isLocationSafe(inout loc : String) : boolean
_do_placeBid(inout cust : Customer, inout amount : double) : boolean
_do__do_join(inout cust : Customer) : boolean
-_do__do__do_join(inout cust : Customer) : boolean
leave(inout cust : Customer) : boolean
-_do_leave(inout cust : Customer) : boolean
_do__do_placeBid(inout cust : Customer, inout amount : double) : boolean
-_do__do__do_placeBid(inout cust : Customer, inout amount : double) : boolean
_do_postMessage(inout cust : Customer, inout message : String) : boolean
-_do__do_postMessage(inout cust : Customer, inout message : String) : boolean
getJoinedCustomers( ) : Vector
getHighestBidder( ) : Customer
getSinks( ) : Vector
getActor( ) : Customer
getUser( ) : Customer
setLocation(location : String) : void
getLocation( ) : String
setAuctionSystem(auctionSystem : AuctionSystem) : void
getAuctionSystem( ) : AuctionSystem
setItem(item : String) : void
getItem( ) : String
setMinPrice(minPrice : double) : void
getMinPrice( ) : double
setDuration(duration : int) : void
getDuration( ) : int
setHighestBid(highestBid : double) : void
getHighestBid( ) : double
setHighestBidder(highestBidder : Customer) : void
getMessages( ) : Vector
setDescription(description : String) : void
setActor(actor : Customer) : void
setUser(user : Customer) : void
setController(controller : UIController) : void
getController( ) : UIController
classes
InvalidLocationException
attributes
operations
classes
AuctionSystem
attributes
-activeSessions : Account [0..*]
-bank : Bank [1..*]
-accounts : Account [0..*]
-auctions : Auction [0..*]
-user : Customer
-controller : UIController
operations
closeAccount(inout cust : Customer) : void
withdraw(inout cust : Customer, inout amount : double) : boolean
auctionTerminated(inout Auction auction) : void
requestAuctions( ) : Auction
searchAuctions(inout desc : String [0..*], inout loc : String [0..*]) : Auction
-_do_logoff(inout user : Customer) : void
-accountExists(inout user : Customer) : boolean
validateBid(inout cust : Customer, inout amount : double) : boolean
createAuction(inout cust : Customer, inout item : String, inout desc : String, inout minPrice : double, inout duration : int) : void
-isLoggedIn(inout cust : Customer) : boolean
enrol(inout user : Customer) : boolean
-_do_enrol(inout user : Customer) : boolean
logon(inout user : Customer) : boolean
isLocationSafe(inout loc : String) : boolean
deposit(inout cust : Customer, inout amount : double) : boolean
-_do_deposit(inout cust : Customer, inout amount : double) : boolean
_do_logon(inout user : Customer) : boolean
-_do__do_logon(inout user : Customer) : boolean
logoff( ) : void
-_do_logoff( ) : void
getActiveSessions( ) : Vector
getAccounts( ) : Vector
getAuctions( ) : Vector
getUser( ) : Customer
getBank( ) : Vector
setUser(user : Customer) : void
setController(controller : UIController) : void
getController( ) : UIController
classes
-auctions
[0..*]
-activityMonitors
[0..*]
-accounts
[0..*]
[0..*]
-activeSessions
[1..*]
-bank
[0..*]
-joinedCustomers
[0..*]
-sinks
-user
-actor
-highestBidder
[0..1]
[1..*]
-producers
-user
-controller
-user
Customer
attributes
-creditCardNum : String
-name : String
-age : int
-address : String
-interests : String [0..*]
-producers : LocationProducer [1..*]
operations
setAge(inout age : int) : void
setAddress(inout address : String) : void
notify(inout ia : Auction) : void
addInterest(inout interest : String) : void
getLocation( ) : String
setCreditCardNum(creditCardNum : String) : void
getCreditCardNum( ) : String
setName(name : String) : void
getName( ) : String
getAge( ) : int
getAddress( ) : String
getInterests( ) : Vector
getProducers( ) : Vector
classes
LocationProducer
attributes
-name : String
-position : String
operations
setName(name : String) : void
getName( ) : String
setPosition(position : String) : void
getPosition( ) : String
classes
UIController
attributes
-lightEnabled : boolean
-activityMonitors : AuctionSystem [0..*]
-user : Customer
operations
display( ) : void
enableLight( ) : void
setLightEnabled(lightEnabled : boolean) : void
getLightEnabled( ) : boolean
getActivityMonitors( ) : Vector
setUser(user : Customer) : void
getUser( ) : Customer
classes
Account
attributes
-creditAmount : double
-frozen : boolean = false
-owner : Customer
-name : String
-creditCardNum : String
-address : String
-age : int
operations
close( ) : void
freeze( ) : void
isFrozen( ) : boolean
store( ) : void
unfreeze( ) : void
setCreditAmount(creditAmount : double) : void
getCreditAmount( ) : double
setFrozen(frozen : boolean) : void
getFrozen( ) : boolean
setHighestBidder(highestBidder : Customer) : void
getHighestBidder( ) : Customer
setOwner(owner : Customer) : void
getOwner( ) : Customer
setName(name : String) : void
getName( ) : String
setCreditCardNum(creditCardNum : String) : void
getCreditCardNum( ) : String
setAddress(address : String) : void
getAddress( ) : String
setAge(age : int) : void
getAge( ) : int
classes
Bank
attributes
operations
deposit(inout ccNumber : String, inout name : String, inout amount : double) : boolean
validate(inout ccNumber : String, inout name : String) : boolean
withdraw(inout ccNumber : String, inout name : String, inout amount : double) : boolean
classes

Bibliography

[1] Batista, T., Chavez, C., Garcia, A., Kulesza, U., Sant'Anna, C., and Lucena, C. Aspectual Connectors: Supporting the Seamless Integration of Aspects and ADLs. In *ACM SIGSoft XX Brazilian Symposium on Software Engineering (SBES'06)* (Florianopolis, Brazil, 2006).

[2] Bergmans, L., and Aksit, M. Composing crosscutting concerns using composition filters. *Commun. ACM 44*, 10 (2001), 51–57.

[3] Bézivin, J. On the Unification Power of Models. *Software and System Modeling (SoSym) 4*, 2 (2005), 171–188.

[4] Booch, G. *Object-oriented analysis and design with applications (2nd ed.).* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[5] Carton, A., Clarke, S., Senart, A., and Cahill, V. Aspect-Oriented Model-Driven Development for Mobile Context-aware Computing. In *SEPCASE '07: Proceedings of the 1st International Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments* (Washington, DC, USA, 2007), IEEE Computer Society, p. 5.

[6] Chitchyan, R., Pinto, M., Rashid, A., and Fuentes, L. Compass: Composition-Centric Mapping of Aspectual Requirements to Architecture. In *Transactions on Aspect-Oriented Software Development: Special Issue on Early Aspects* (2007).

[7] Chitchyan, R., Rashid, A., Rayson, P., and Waters, R. Semantics-based composition for aspect-oriented requirements engineering. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development* (New York, NY, USA, 2007), ACM Press, pp. 36–48.

[8] Clarke, S. *Composition of Object-Oriented Software Design Models.* PhD thesis, Dublin City University, Jan. 2001.

[9] Clarke, S. Extending standard UML with model composition semantics. *Sci. Comput. Program. 44*, 1 (2002), 71–100.

[10] Clarke, S., and Baniassad, E. *Aspect-Oriented Analysis and Design. The Theme Approach.* Object Technology Series. Addison-Wesley, Boston, USA, 2005.

[11] Clarke, S., and Walker, R. J. Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J. Tech. Rep. TR-2001-05, 08 2001.

[12] Czarnecki, K., and Helsen, S. Classification of Model Transformation Approaches. In *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture* (oct 2003).

[13] Dey, A., Salber, D., and Abowd, G. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications, 2001.

[14] Dey, A. K. Understanding and Using Context. *Personal Ubiquitous Comput. 5*, 1 (2001), 4–7.

[15] Didonet Del Fabro, M., Bzivin, J., Jouault, F., Breton, E., and Gueltas, G. Amw: a generic model weaver. In *Journes sur l'Ingnierie Dirige par les Modles (IDM05)* (2005), pp. 105–114. 2-7261-1284-6.

[16] Dijkstra, E. W. *A Discipline of Programming.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[17] Filman, R. E., Elrad, T., Clarke, S., and Akşit, M., Eds. *Aspect-Oriented Software Development.* Addison-Wesley, Boston, 2005.

[18] Frederick P. Brooks, J. No silver bullet: essence and accidents of software engineering. *Computer 20*, 4 (1987), 10–19.

[19] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[20] Gray, J., Bapty, T., Neema, S., Schmidt, D., Gokhale, A., and Natarajan, B. An Approach for Supporting Aspect-Oriented Domain Modeling, 2003.

[21] Groher, I., and Voelter, M. Xweave: models and aspects in concert. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling* (New York, NY, USA, 2007), ACM Press, pp. 35–40.

[22] Harrison, W. H., and Ossher, H. Subject-Oriented Programming (A Critique of Pure Objects). In *OOPSLA* (1993), pp. 411–428.

[23] Harrison, W. H., Ossher, H. L., and Tarr, P. L. Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. Research Report RC22685 (W0212-147), IBM, December 2002.

[24] J. Zhang, T. Cottenier, A. v. d. B., and Gray, J. Aspect Interference and Composition in the Motorola Aspect-Oriented Modeling Weaver.

[25] Jackson, A., Barais, O., Jzquel, J.-M., and Clarke, S. Toward A Generic And Extensible Merge. In *Models and Aspects workshop, at ECOOP 2006* (Nantes, France, 2006).

[26] Jackson, A., and Clarke, S. Towards a Generic Aspect Oriented Design Process. In *Aspect Oriented Modelling workshop, at MoDELS* (Montego Bay, Jamaica, 2005), M. S. E. 2005, Ed., pp. 110–119.

[27] Jackson, A., and Clarke, S. Initial Version of Aspect-Orienteed Design Approach, AOSD-Europe D38, 2006.

[28] Jackson, A., and Clarke, S. Testing Executable Themes, 2006.

[29] Jackson, A., Klein, J., Baudry, B., and Clarke, S. KerTheme: Testing Aspect Models. In *Model Driven Development and Model Driven Testing workshop at ECMDA* (2006).

[30] Jacobson, I., and Ng, P.-W. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series).* Addison-Wesley Professional, 2004.

[31] Kiczales, G. Aspect-oriented programming. *ACM Comput. Surv. 28*, 4es (1996), 154.

[32] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An Overview of AspectJ. *Lecture Notes in Computer Science 2072* (2001), 327–355.

[33] Kolovos, D. S., Paige, R. F., and Polack, F. Merging Models with the Epsilon Merging Language (eml). In *MoDELS* (2006), O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds., vol. 4199 of *Lecture Notes in Computer Science*, Springer, pp. 215–229.

[34] Lieberherr, K. J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, 1996.

[35] Mellor, S. J., and Balcer, M. *Executable UML: A Foundation for Model-Driven Architectures.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Foreword By-Ivar Jacoboson.

[36] Mens, T., Czarnecki, K., and Gorp, P. V. Discussion – A Taxonomy of Model Transformations. In *Language Engineering for Model-Driven Software Development* (2005), J. Bezivin and R. Heckel, Eds., no. 04101 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. <http://drops.dagstuhl.de/opus/volltexte/2005/11> [date of citation: 2005-01-01].

[37] Miller, J., and Mukerji, J. MDA Guide Version 1.0.1. Tech. rep., Object Management Group (OMG), 2003.

[38] Moreira, A., Rashid, A., and Araujo, J. Multi-Dimensional Separation of Concerns in Requirements Engineering. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 285–296.

[39] Munnelly, J., Fritsch, S., and Clarke, S. An Aspect-Oriented Approach to the Modularisation of Context. In *PERCOM '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 114–124.

[40] nad JrgenDingel, A. Z. Modeling UML 2 Package Merge With Alloy".

[41] Nuseibeh, B., Kramer, J., and Finkelstein, A. A framework for expressing the relationships between multiple views in requirements specification. *Software Engineering 20*, 10 (1994), 760–773.

[42] Ossher, H., Kaplan, M., Katz, A., Harrison, W., and Kruskal, V. Specifying subject-oriented composition. *Theor. Pract. Object Syst. 2*, 3 (1996), 179–202.

[43] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15*, 12 (1972), 1053–1058.

[44] Pinto, M., Gamez, N., and Fuentes, L. Towards the Architectural Definition of the Health Watcher System with AO-ADL. In *EARLYASPECTS '07: Proceedings of the Early Aspects at ICSE* (Washington, DC, USA, 2007), IEEE Computer Society, p. 5.

[45] Pottinger, R., and Bernstein, P. Merging models based on given correspondences, 2003.

[46] Rashid, A. Early Aspects: A Model for Aspect-Oriented Requirements Engineering, 2002.

[47] Reddy, Y. R., Ghosh, S., France, R. B., Straw, G., Bieman, J. M., McEachen, N., Song, E., and Georg, G. Directives for Composing Aspect-Oriented Design Class Models. 75–105.

[48] Sánchez, P., Fuentes, L., Jackson, A., and Clarke, S. Aspects at the Right Time. In *Transactions on Aspect-Oriented Software Development: Special Issue on Early Aspects* (2007).

[49] Schilit, B., Adams, N., and Want, R. Context-Aware Computing Applications. In *IEEE Workshop on Mobile Computing Systems and Applications* (Santa Cruz, CA, US, 1994).

[50] Senart, A., Cunningham, R., Bouroche, M., O'Connor, N., Reynolds, V., and Cahill, V. MoCoA: Customisable Middleware for Context-Aware Mobile Applications. In *OTM Conferences (2)* (2006), R. Meersman and Z. Tari, Eds., vol. 4276 of *Lecture Notes in Computer Science*, Springer, pp. 1722–1738.

[51] Shlaer, S., and Mellor, S. J. *Object lifecycles: modeling the world in states.* Yourdon Press, Upper Saddle River, NJ, USA, 1992.

[52] Straw, G., Georg, G., Song, E., Ghosh, S., France, R. B., and Bieman, J. M. Model Composition Directives. In *UML* (2004), pp. 84–97.

[53] T. Cottenier, A. v. d. B., and Elrad, T. The Motorola WEAVR: Model Weaving in a Large Industrial Context.

[54] Tarr, P., Ossher, H., Harrison, W., and Stanley M. Sutton, J. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (Los Alamitos, CA, USA, 1999), IEEE Computer Society Press, pp. 107–119.

[55] Walker, R. J. Eliminating cycles in composed class hierarchies. Tech. Rep. TR-2000-07, 2000.

[56] Weiser, M. The computer for the 21st century. *Scientific American 265*, 3 (September 1991), 66–75.

[57] Zito, A., Diskin, Z., and Dingel, J. Package Merge in UML 2: Practice vs. Theory? In *MoDELS* (2006), O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds., vol. 4199 of *Lecture Notes in Computer Science*, Springer, pp. 185–199.