# Model-Driven Theme/UML

Andrew Carton, Cormac Driver, Andrew Jackson, and Siobhán Clarke

Distributed Systems Group,
Department of Computer Science and Statistics,
Trinity College Dublin, Ireland
{firstname.lastname}@cs.tcd.ie

**Abstract.** Theme/UML is an existing approach to aspect-oriented modelling that supports the modularisation and composition of concerns, including crosscutting ones, in design. To date, its lack of integration with model-driven engineering (MDE) techniques has limited its benefits across the development lifecycle. Here, we describe our work on facilitating the use of Theme/UML as part of an MDE process. We have developed a transformation tool that adopts model-driven architecture (MDA) standards. It defines a concern composition mechanism, implemented as a model transformation, to support the enhanced modularisation features of Theme/UML. We evaluate our approach by applying it to the development of mobile, context-aware applications-an application area characterised by many non-functional requirements that manifest themselves as crosscutting concerns.

## 1 Introduction

Aspect-oriented software development (AOSD) extends the decomposition and composition mechanisms of existing software development paradigms in order to more effectively modularise interdependent concerns [5]. Theme/UML is part of the broader Theme approach to aspect-oriented analysis and design [1], extending standard UML to explicitly support both the modularisation and composition of concerns in design.

We recently conducted an investigation into the application of Theme/UML to the design of mobile, context-aware applications, which motivated much of the work described in this paper. Mobile, context-aware computing is a computing paradigm in which applications can discover and take advantage of contextual information such as user location, time of day, nearby people/computing devices and user activity [26]. Such applications can run on a range of diverse computing platforms and in multiple deployment environments, from personal digital assistants and mobile phones running Java, to small embedded wearable devices supporting C. In specifying applications of this nature, software developers must consider non-functional mobility and context-awareness concerns that negatively impact software complexity and therefore make the use of Theme/UML appropriate. It emerged from our investigation that although Theme/UML can aid the modularisation of mobility and context-awareness concerns, the prevalence of multiple target environments and the lack of support for automated model-to-code transformations restricted the contribution our designs made towards producing widely deployable solutions. This finding motivated extensions (with supporting tools) to the Theme/UML approach that reduce the effort required to progress from a single system model to multiple deployable applications derived from this model. A model-driven software engineering process was adopted to support the automatic generation of platform-specific models and code from a generic model, thereby addressing platform heterogeneity.

Model-driven engineering (MDE) is an approach to software development that emphasises the use of models as primary engineering artefacts. It addresses platform heterogeneity by abstracting platform-independent models and providing means to automatically transform these models to one or more specific target platforms. The model-driven approach, through architectural separation of concerns, promotes portability, interoperability and reusability [21].

In this paper, we present our work on integrating Theme/UML with an MDE process. We have developed a tool that supports the specification of platform-independent models with Theme/UML and subsequent automatic transformations to platform-specific models and code. This tool is compliant with the model-driven architecture (MDA) standards defined by the Object Management Group (OMG) [10], while retaining the general purpose and intention of the original Theme/UML semantics. We have defined an MDA process with a composition phase implemented as a model transformation, allowing developers to avail of the enhanced modularisation features in Theme/UML. Aspect-oriented platform-independent models, specified in Theme/UML, are automatically transformed to object-oriented platform-specific models and code, giving the developer powerful decomposition and composition capabilities at design time without tying them to an aspect-oriented platform. To demonstrate our approach, we implemented transformations to two mobile environments, J2ME and .NET CF. We conducted a case study-based evaluation by applying the tool to the implementation of a mobile, context-aware application with a number of non-functional requirements that manifest themselves as crosscutting concerns.

The remainder of this paper is as follows. Section 2 describes the model-driven Theme/UML tool from an implementation perspective, while Sect. 3 discusses the application development process it facilitates. Section 4 presents a case study of our approach as applied to the development of a mobile, context-aware application with crosscutting requirements. Section 5 discusses related work while Sect. 6 provides a summary of this paper and a brief overview of our continuation of this work.

## 2 Model-Driven Theme/UML: Implementation

In this section, we present the implementation of the model-driven Theme/UML tool. We first outline our initial design decisions and then describe the implementation phase. The section concludes with a discussion of the challenges and difficulties encountered.

### 2.1 Initial Design Decisions

Our initial design decisions concerned how best to integrate and implement Theme/UML with current MDA guidelines, technologies and tools. Theme/UML is defined as a meta-object facility (MOF)-based extension of the UML 1.3 beta R7 metamodel [11]. This version of the UML originated before the OMG updated their standards to conform to the MDA vision [22], currently at version 2.1.1. As such, this definition was not compatible with the current standards and conventions, and consequently hindered our objective to offer Theme/UML as an MDA solution. In order to achieve this objective, we investigated three strategies.

The first strategy involves extending the UML 2.1 metamodel. This is a heavyweight solution that requires augmentation of the appropriate metaclasses and metarelationships [17] to support the Theme/UML extensions. However, porting Theme/UML to UML 2.1 proved prohibitively challenging, primarily because of the significant dissimilarity between the two versions of the UML metamodels. Furthermore, invasive metamodel changes to the UML preclude the use of standard UML tool support.

Next, we investigated the use of a marking[1] UML profile to support the expression of a composition specification, while using UML Package Merge to realise Theme/UML's composition semantics. As a UML Profile is a lightweight extension mechanism supported both at the modelling level and by the UML compliance levels[2], any compliant UML graphical tool would be adequate. The UML Package Merge is part of the UML metamodel that allows one package to be merged with another, accommodating the interoperability of tools by allowing a higher level of compliance to be merged with a lower level one. In Theme/UML, a `theme` is defined as an extension of a `package`; therefore UML Package Merge could potentially have been used to define Theme/UML's composition semantics by redefining the UML Package Merge at the metamodel level [13]. However, heavyweight metamodel extensions had been ruled out as impractical due to lack of tool support. Investigating Package Merge as a foundation for defining Theme/UML's composition semantics proved to be unsuccessful at the modelling level also, as it lacks the ability to support additional types [30]. Further evidence suggested that the Package Merge is not suitable for meta-model builders and the definition of transformations [29].

The third strategy, similar to the second, involved the definition of a marking UML Profile. The process involves marking a model to indicate the composition specification and then mapping this specification to an instance of a new composition metamodel. A composition metamodel defined in MOF can be used to indicate the structure and behaviour of Theme/UML's composition semantics. We decided that this strategy was more favourable than the others for two

[1] Marking is a technique that allows a set of elements in a UML model to be identified for transformation in a non-invasive way [19].

[2] UML is stratified into a number of horizontal layers of increasing capabilities called compliance levels. These are points at which a tool can claim compliance to the standard.

reasons. The first advantage is gained from the distinct separation of the graphical extensions in the UML and the definition of the composition semantics. The composition semantics can evolve independently from the graphical extensions by extending the composition metamodel. Likewise, if more expression is needed in the marking, only the marking profile and the mapping to the composition metamodel need to change. The second advantage relates to the difficulties of using UML Package Merge in defining the composition semantics, in which their structure and behaviour are expressed entirely in textual form in the UML standard. The use of a composition metamodel, in our opinion, better captures and illustrates these semantics in a more formal manner.

Apart from deciding how best to integrate Theme/UML with the MDA process, we had to decide on which, if any, third party tools to use. Given that we were working with a standard modelling language, we adopted a standard UML editor called MagicDraw[3]. This tool exports models in Eclipse Modelling Framework XMI format, a format commonly supported by MDA tools. For code generation, we adopted the openArchitectureWare (oAW)[4] model-driven generator framework, which aids the production of source code from XMI. The decision to adopt only standard tools and formats means that developers are free to use one of the many UML editors or source code generators that support XMI.

## 2.2 Implementation

The design process is separated into three distinct phases that relate to the activity of the designer during that phase-the modelling phase, the composition phase and the transformation phase. Figure 1 illustrates the mapping specifications and definitions that enclose each phase with a description of their implementing technologies parenthesised beneath each.

**Modelling Phase.** Designers use Theme/UML (see Appendix A for more details) during the modelling phase to modularise application concerns. Two requirements had to be met in order to accomplish the implementation of our MDA strategy at this phase.

1. Theme/UML's composition semantics must be defined in the form of a marking profile.
2. A graphical UML tool is required that supports both the definition of a UML profile and the standard UML features that Theme/UML requires (i.e. Class and Interaction Diagrams).

The first requirement motivated the definition of a Theme/UML Marking Profile, illustrated in Fig. 2, that extends UML 2.1 and supports the designer in creating a composition specification. In this case, the marks guide the designer in creating a composition specification by decorating the UML elements

[3] MagicDraw 12.5, `http://www.magicdraw.com`

[4] openArchitectureWare, `http://www.openarchitectureware.org`

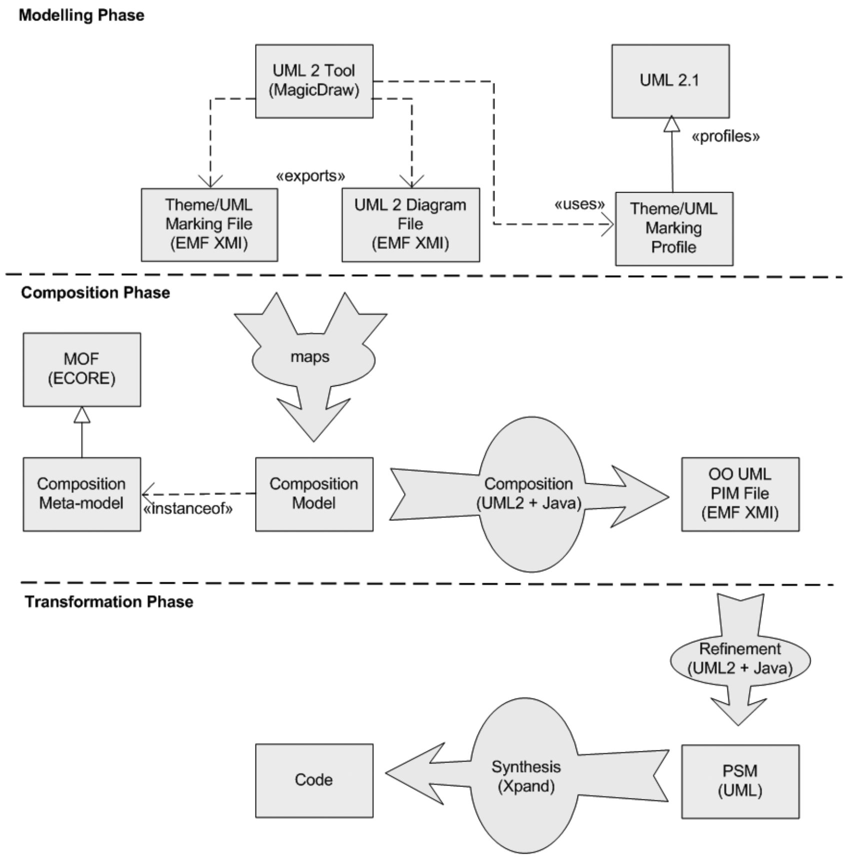


**Fig. 1.** Model-Driven Theme/UML Mappings and Definitions

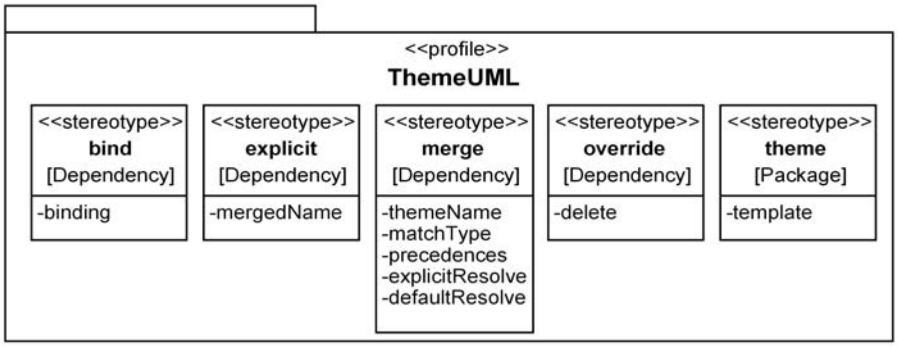

<<profile>>
ThemeUML
<<stereotype>> bind [Dependency]
-binding
<<stereotype>> explicit [Dependency]
-mergedName
<<stereotype>> merge [Dependency]
-themeName
-matchType
-precedences
-explicitResolve
-defaultResolve
<<stereotype>> override [Dependency]
-delete
<<stereotype>> theme [Package]
-template

**Fig. 2.** Theme/UML Marking Profile

with stereotypes and tagged values from the Theme/UML Marking Profile. We use this lightweight extension mechanism to support extension of Theme/UML without requiring invasive changes at the UML metamodel level. There are five stereotypes indicated in the Theme/UML Marking Profile. A `theme` stereotype allows a UML Package to be marked to indicate that it may be used in a composition relationship. If the theme is to be designed as an aspect, then the tagged definition `template` indicates the string that represents the template parameters that trigger crosscutting behaviour. A `merge` stereotype is placed on a `Dependency` to indicate the themes involved in a merge composition relationship. The tagged definitions of this stereotype (`themeName, matchType, precedences, explicitResolve` and `defaultResolve`) can be applied on the stereotype to indicate the properties of the merge. The `override` stereotype can be placed on a `Dependency` and indicates an override composition relationship, while the tagged definition `delete` represents the elements to be deleted. A `bind` stereotype is applied to a `Dependency` and is constrained as a binary dependency between an aspect and base theme. The tagged definition `binding` represents the elements that instantiate the templates of the aspect theme. Finally, an `explicit` stereotype allows explicit matching of concepts in a composition relationship and the tagged definition `mergedName` indicates the composed value.

Magicdraw was chosen to meet our second requirement for three reasons. First, it supports UML 2.1 modelling and therefore supports the implementation of a UML Profile definition. Second, it exports to the Eclipse Modelling Framework[5] (EMF) XML Metadata Interchange (XMI), which is compatible with transformations at the later stages of the MDA process. Third, it supports both class and sequence UML diagrams, which is a necessity for Theme/UML.

After completing a design in Theme/UML, the tool exports two files-the Theme/UML Marking Profile File and the UML 2 Diagram File. Both files are serialised with the EMF XMI.

**Composition Phase.** The composition phase allows the designer to automatically compose the model according to the composition specification that was created during the modelling phase. This phase is implemented using two transformations. The first transformation takes the two files from the output of the Modelling Phase and maps them to create a composition model that is an instance of the composition metamodel. The second transformation takes this composition model and executes it to produce an EMF XMI file that holds the object-oriented PIM. This horizontal transformation, as illustrated in the middle of Fig. 1, is termed a composition.

***Mapping.*** The first transformation is defined as a mapping from the Theme/UML Marking Profile (c.f. Fig. 2) to the Composition Metamodel (c.f. Fig. 3), as illustrated in Fig. 1. The mapping specification uses the UML elements decorated with marks to transform them into a composition model. This is achieved in two steps. In the first step, an associated element in the `ComposableElement` hierarchy (c.f. Fig. 3) is created that corresponds to the

[5] `http://www.eclipse.org/modeling/emf/`

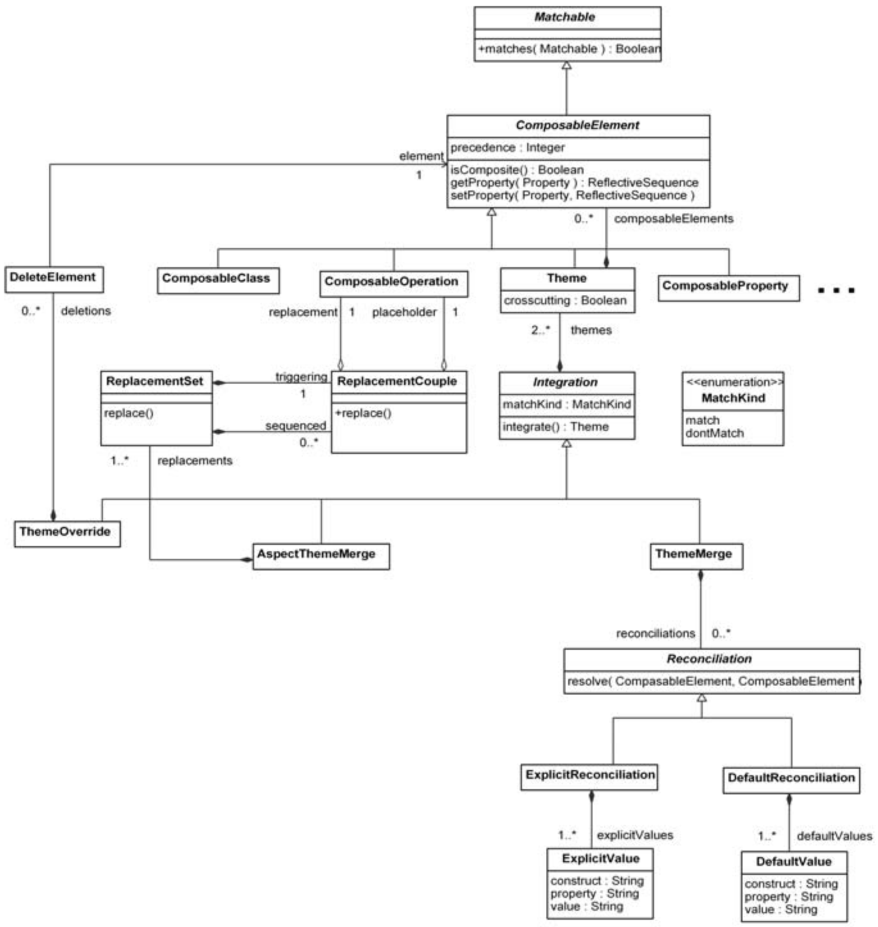


**Fig. 3.** Theme/UML Composition Metamodel

UML element being mapped. For example, a UML `Package` with a `theme` stereotype applied in the UML Design Model specifies the creation of a `Theme` element in the composition model. In the second step, a detailed composition specification is created in the composition model that maps each composition relationship and its properties in the UML Design Model to their equivalent in the composition metamodel. For example, a UML `Dependency` with a `bind` stereotype in the UML Design Model specifies the creation of an `AspectThemeMerge` in the composition model, with a `binding` tagged value on that stereotype, resulting in the creation of its respective `ReplacementSet` and `ReplacementCouple`s as containing properties for that integration type.

As a result of using strings as tagged values, the mapping implementation heavily relies on parsing techniques and the use of the Object Constraint Language (OCL) as a means to extract and query elements in the UML model, respectively.

In particular, OCL proved especially useful in supporting Theme/UML's pointcut selection mechanism.

***Composition.*** Figure 3 illustrates the metamodel used to describe the structure and behaviour of Theme/UML's composition semantics[6]. Each element that can be involved in a composition is defined by a `ComposableElement`. A `ComposableElement` implements a `Matchable` element that abstracts the notion of a matching criterion. This matching criterion is specific to each element and is implemented in a manner appropriate to the element being matched. For example, a UML Operation is matched to the name of the operation, the types of the parameters and the type of the return value. An `Integration` is an abstract metaclass that describes the way in which themes are to be integrated. The three integration strategies that Theme/UML defines are `ThemeMerge`, `ThemeOverride` and `AspectThemeMerge`. Each have their additional metaclasses and metarelationships that define how the integration is supported and behaves.

A `ThemeMerge` integration describes how base themes are to be composed. This necessitates a definition of how overlapping specifications are resolved through the `Reconciliation` hierarchy. An `ExplicitReconciliation` allows a designer to indicate an explicit preference in the composed theme if elements in a merge match, using one or more `ExplicitValues`. An `ExplicitValue` indicates the specification of a single matching element, referencing the `construct` property of the element and the `value` of that element upon composition. Likewise, a `DefaultReconciliation` allows a designer to specify the default value for elements of a particular type if a conflict arises between elements of that type in the composition. The reconciliation can have one or more `DefaultValues`. A `DefaultValue` indicates the specification of a single matching element of a particular type and the `value` of that type upon reconciliation. The final reconciliation strategy defined by Theme/UML is precedence. A precedence reconciliation specifies precedence on a composable element when a match occurs in a merge. A precedence strategy is integrated into an attribute of a `ComposableElement` rather than having its own metaclass.

The second integration strategy defined by Theme/UML, `ThemeOverride`, describes how one theme's specification is overridden by that of another theme. This metaclass can contain a set of `DeleteElements` which indicate the elements that get deleted upon the override.

The third integration strategy, `AspectThemeMerge`, specifies how an aspect theme is composed with base themes. Each `AspectThemeMerge` has a number of `ReplacementSets` equivalent to the number of sequence diagrams in each aspect theme that it represents. Each `ReplacementSet` must have one triggering `ReplacementCouple` and can have many sequenced `ReplacementCouples`. A `ReplacementCouple` references both a placeholder `ComposableOperation` and its replacement `ComposableOperation`.

The composition metamodel was realised in Ecore and implemented using EMF libraries. Ecore is the EMF's meta metamodel and is synonymous with

[6] Due to space limitations, Fig. 3 only illustrates a subset of the composable elements.

MOF, with some slight variations. The EMF implements both the UML 2 standard and the OCL standard with Ecore in Java and provides a supporting library called UML2. The EMF also defines its own XMI schema that allows libraries to read and write any EMF-based model.

While the composition metamodel defines the structure and behaviour of Theme/UML's composition semantics, a mapping specification defines how these semantics are executed. In our approach, we implemented a mapping specification that targets an object-oriented PIM. In this case, all the integration strategies are executed. However, if a transformation to an AO PIM is desired, the metamodel is extensible enough to support the definition of a mapping specification that only executes some of the integration strategies (e.g. targeting an asymmetric AOP platform would require only the overlapping specifications to be resolved).

**Transformation Phase.** The output from the composition phase is an object-oriented PIM that can be transformed into a platform-specific model. Rather than go straight from a PIM to code, we made the decision to go to an intermediate PSM. The reason for this is that the proposed approach is elaboration-oriented, meaning the PIM is not computationally complete and does not contain the full executable specification [18]. The PSM is open for re-factoring and elaboration of low-level details by the designer. There are two transformations implemented in this phase, refinement and synthesis, which support the developer in moving from a PIM-based design to a PSM-based design and finally to code respectively.

After choosing a target platform, a model-to-model transformation refines the object-oriented PIM into a PSM suitable to model the concepts for the chosen platform. This refinement requires a number of platform-specific extensions. For each PSM, a UML profile is created that extends the standard UML datatypes with those that are specific to the language and platform. The profile can also include the namespaces and datatypes needed to further elaborate the PSM. The transformation was implemented using Java and the UML2 library.

The second transformation, illustrated as synthesis, allows a PSM to be transformed into code. This transformation is implemented using a template-based code-generation technology called XPand-part of the oAW framework. In general, there are two main approaches for model-to-text (M2T) transformation, visitor-based approaches and template-based approaches [3]. Template-based tools such as XPand use a text-based declarative language as a means for selection of model nodes and iterative expansion. We decided to use Xpand to transform the UML class diagrams to code. For the generation of behavioural code with sequence diagrams, we used a visitor-based approach implemented in Java. Sequence diagrams are written in the UML in-order, and so a visitor-based approach is more desirable than a template-based approach as the visitor can step through the full trace in order and generate code on the fly. As XPand supports Java extensions, the two approaches could be integrated, producing both compilable structural and behavioural code from the class diagrams and sequence diagrams respectively. The code generation capabilities could be extended by implementing support for standard UML behavioural diagrams.

## 2.3 Discussion

This section discusses the difficulties and challenges we encountered while implementing our approach to the integration of Theme/UML with current MDA standards, guidelines and technologies.

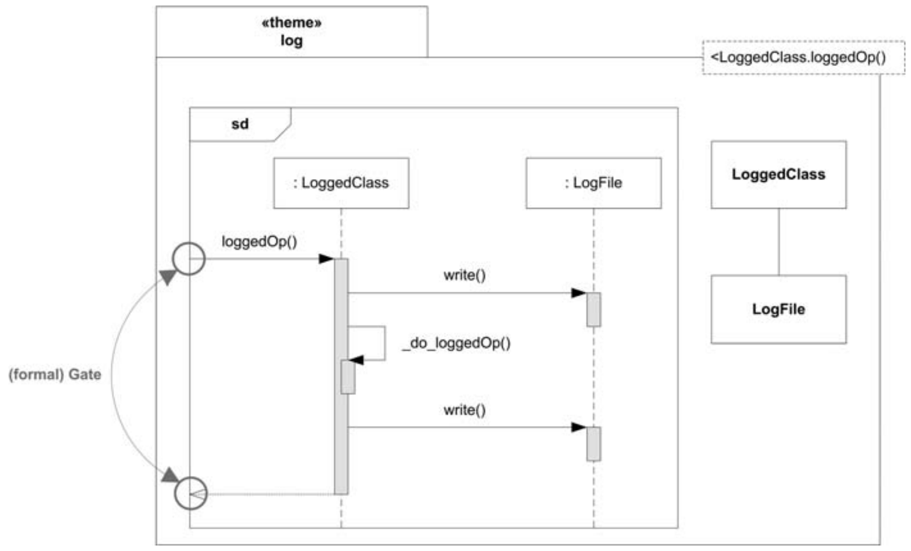


**Fig. 4.** UML 2.1 sequence diagram

**Modelling Triggering and Returning Messages.** In Theme/UML, UML sequence diagrams are used to indicate how and when the crosscutting occurs in relation to the abstract templates of an aspect theme. The UML metamodel in which Theme/UML was defined had no support for indicating a message in the case where the sender or receiver was unknown. Consequently, this resulted in these messages being drawn without a sender or receiver, violating a number of constraints of the metamodel. This would be especially problematic in the creation of a mapping specification, where it is assumed all UML models are compliant to the constraints and are well-formed. However, the UML 2.1 metamodel has improved the definition of sequence diagrams. A `Gate` is a connection point for relating a `Message` outside an `InteractionFragment` with a `Message` inside the `InteractionFragment`. With Gate support, the sender and receiver of the initial triggering message can now be unspecified while conforming to the constraints of the metamodel. Figure 4 shows the updated Theme/UML semantics and notation for indicating triggering and returning operations, explicitly indicating where the gates are.

**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 existing UML metaclasses, profile extensions for n-ary relationships were required. `Association` is restricted as a relationship between certain types; therefore, `Dependency` is the next best option, allowing n-ary relationships between `NamedElements`. It emerged that MagicDraw only supported one-to-one relationships with a `Dependency`, and as such deviates from the standard. To work around this, the desired relationships were emulated by drawing an additional `Dependency` on the `Dependency` that was drawn between two model elements. This workaround could be successfully implemented since a `Dependency` itself is a `NamedElement`. However, the solution necessitated extra parsing logic to determine all the elements participating in a composition relationship.

**Modelling Sequence Diagrams.** When we began designing our tool, we surveyed a number of UML 2 modelling tools, including Topcased[7], Poseidon[8] and Rational Software Architect[9]. We decided to use Magicdraw as the community edition was free; it offered export to EMF XMI and had support for class and sequence diagrams. However, it emerged that the EMF XMI export implemented by MagicDraw was faulty for sequence diagrams. We based an alternate approach on the UML2 editor provided by the UML2 library of the EMF. This workaround involves using this tree-based graphical tool to create the sequence diagrams by hand. The graphical tool offers the designer a little more abstraction than working with the raw XMI directly (which requires detailed knowledge of the specification). Although this workaround is undesirable from a designer perspective, it was the only option available as no other free tool surveyed was capable of viewing or writing sequence diagrams to EMF XMI correctly. Once a tool that supports sequence diagrams becomes available, it can be used instead.

**Code Generation for Sequence Diagrams.** A visitor-based approach was adopted to generate code from sequence diagrams. However, we discovered that the sequence diagrams in the UML 2.1 specification are currently unsuitable for the purpose of code generation. The OMG Revision Task Force for UML[10] currently lists a number of pending revisions. One such revision describes that the arguments of a `Message` can only be `ValueSpecifications`, and the creation, referencing and assignment of variables in the underlying model remains ambiguous. To get around this restriction, a `LiteralString` is used to pass arguments in textual form. However, this solution is undesirable because it precludes complete validation of the model. We are currently awaiting publication of the next UML 2 standard to evaluate the fixes for these issues in order to provide better support for code generation from sequence diagrams.

[7] `http://www.topcased.org`
[8] `http://www.gentleware.com`
[9] `http://www-306.ibm.com/software/awdtools/architect/swarchitect`
[10] `http://www.omg.org/issues`

**Selection of Transformation Tools.** Prior to the design of our tool, we investigated a number of Model-to-Model (M2M) transformation languages such as ATL[11], Kermeta[12] and oAW Xtend[13]. The UML 2 is a large and complex metamodel, and writing valid transformations has been proven to be both challenging and intricate [8]. At that time, we found it easier to use the EMF and UML2 libraries in Java. One of the difficulties we observed with tools like ATL was that it was difficult to transform from a source UML model to a destination UML model when changes to only a small number of meta-model items were required. A tool such as ATL requires rules to copy every single element in the UML metamodel (which is very large) to a new model. Using the libraries, copying a full model requires only a few lines of code and is therefore more feasible. With the rapidly improving state of model-driven tools, however, modern M2M tool support can potentially achieve what we desired during our development phase. For example, ATL now supports superimposition, which allows new rules to be superimposed onto another set of rules, e.g. a full UML2 copy transformation. Redoing our transformations in this manner may be an interesting piece of future work as we believe that working with model-transformation tools is a good way of reducing the complexities of designing mapping specifications and increases extensibility and usability for both the developer and the user.

## 3 Model-Driven Theme/UML: Process

Tool support that integrates both aspect-orientation and MDA is inadequate without a complementary systematic process that clearly defines its use. Previous research on aspect-oriented design (AOD) has amalgamated work on best practises to produce a unified and refined AOD process [14]. Likewise, the MDA Guide [22] provides a flexible and extensive treatise on model-driven processes. Using both individual processes as a basis, we have devised an integrated process.

### 3.1 Process Phases

The requirements of the application should be analysed with a view to identifying concerns before design begins. Theme/Doc, a concern identification approach, supports aspect-oriented requirements analysis and provides explicit mappings from its output to Theme/UML [1]. Theme/Doc can be realised in the MDA process by taking the role of a computation-independent model, where a transformation realises the mappings to a PIM. Other aspect-oriented requirements analysis approaches can be used, provided a mapping exists to Theme/UML, such as that outlined by Sánchez et al. [25]. It is not pertinent to the outlined approach whether this mapping is realised as a manual transformation (indicated by completely elaborating the PIM) or by a semi-automatic transformation (where some artefacts are generated). Future work will investigate tool

[11] `http://www.eclipse.org/m2m/atl`
[12] `http://www.kermeta.org`
[13] `http://www.openarchitectureware.org`

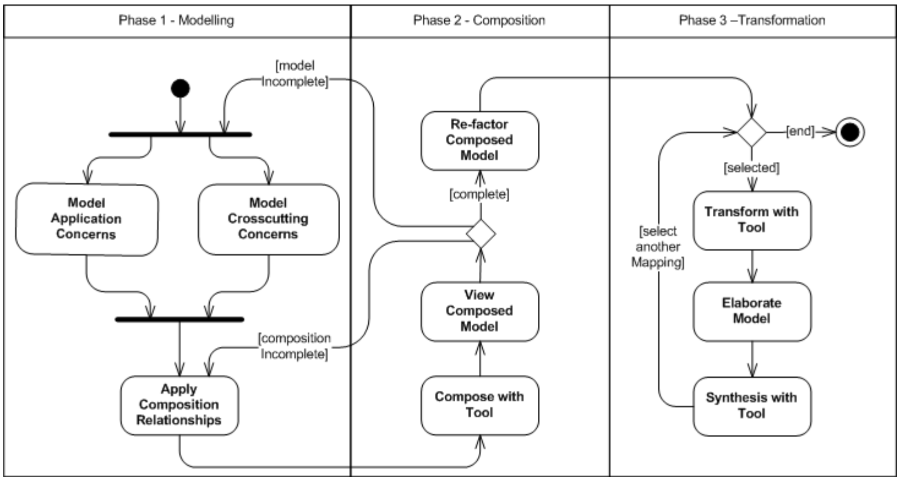


**Fig. 5.** The Model-Driven Theme/UML process

support for these mappings. If automation is provided, the designer would begin with a set of pre-generated UML artefacts that could be further elaborated. This process is illustrated in Fig. 5 as an activity diagram, with the three phases represented by swimlanes.

**Modelling Phase.** The modelling phase illustrates two activities-modelling base application concerns and modelling crosscutting concerns. As Theme/UML supports a symmetric decomposition, and its concern spaces are considered declaratively complete, both of these activities can be done concurrently and independently of each other. This is illustrated by the fork in Fig. 5, and allows themes to be designed in isolation–either by an individual or a team of designers. Each theme is modelled inside a UML Package and should not reference any element outside the package. This ensures that the concern is declaratively complete. The UML Package has 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 has a sequence diagram for each sequence of templates. This sequence diagram illustrates the interaction of the templates with the behaviour of the theme itself.

When themes have been modelled, the designer applies the composition relationships, specifying how themes are to be composed. At the coarsest level of granularity, the individual themes themselves are marked for composition. Support is also available to indicate finer compositions that deviate from the composition specification of the composite container. Base themes use a `merge` stereotype applied to a `Dependency` from the Theme/UML marking profile. The `themeName` tagged definition 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. The `defaultResolve` and `explicitResolve` stereotypes are available as reconciliation options if a conflict arises. An `explicit` stereotype, applied to a `Dependency`, 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 designer can then proceed to the composition phase as indicated in Fig. 5.

**Composition Phase.** Given a UML model with Theme/UML marks applied, the designer can use the tool to compose themes. As illustrated in Fig. 5, the designer can view the composed model and can then choose to take one of three actions. The designer may go back to the modelling phase in the case that the composition relationships need to be reapplied or adjusted due to the composed model being incorrect or incomplete. The second possibility involves going back to the start of the modelling phase to edit the model. Finally, the designer can decide that the composed model is complete.

The next step in the process is refactoring the composed model. We decided to make the composed model open for refactoring for two reasons. The first reason is the possibility of cycles in generalisations. This problem may occur as a result of merging different class hierarchies. The problem has been addressed theoretically through the use of subject-oriented flattening [28,23]. Tool support and process integration for this solution remain future work. Currently, if the problem arises in the composed model, the designer can correct it manually.

The second reason for making the composed model open for refactoring is the need to resolve ambiguities that may arise in the composed model. Conceivably, while designing themes, matching associations may get modelled at different points in each class hierarchy. After composition, these will get duplicated and consequently result in redundant associations. Theme/UML does not naturally cater for these conceptual ambiguities in the semantics of its integration strategies.

**Transformation Phase.** To begin the transformation process, the designer chooses the target platform. The tool takes the PIM, and using the mapping for the target environment, produces a PSM representing the domain-specific extensions of the PIM for that environment. In our approach, the object-oriented PIM that is produced from composition is refined to either a J2ME or .NET CF PSM. A PSM is a direct representation of the underlying platform, modelling precise library support and features of the specific environment. From a pragmatic point of view, it is usually not suitable to model the full specification in the PSM. For example, one could imagine that programming a complex algorithm would be much more effective through the use of code, rather than tediously modelling it with a UML activity diagram [12]. If the full structural and behavioural specification is not modelled in the PSM, it can be specified subsequently in the source code. After elaborating the design of the PSM, the designer can transform from model to code. This kind of transformation is known as synthesis or code generation [20].

## 4 Case Study

In this section, we present an overview of a case study that we conducted in order to assess the applicability of model-driven Theme/UML to an application development scenario. The case study demonstrates how our approach facilitates both the separation of concerns in a mobile, context-aware auction system and the subsequent automatic composition of these concerns to produce platform-specific models and source code. The auction system offers typical functionality such as placing and browsing bids, managing accounts and purchasing goods. It also offers context-awareness features such as notification of auctions that may be of interest to the user, and mobility features such as ensuring that the user is in a valid location before a transaction can proceed and adapting the user interface (UI) to changes in the environment.

Analysis of the requirements specification for the auction system with Theme/Doc identified six base themes and three aspect themes. The base themes cater for the following behaviour:

– Enrolling with the system.
– Browsing auctions.
– Joining auctions.
– Bidding on auctions.
– Transferring credit.
– Administration of auctions.

The aspect themes support the following crosscutting behaviour:

– Adapting the UI (specifically the backlight) based on system events.
– Determining and querying user location.
– Recommending auctions based on user profile and auction history.

Starting at the modelling phase, the analysis provided by Theme/Doc allowed us to create and elaborate a detailed design of each theme. In the interest of brevity, we do not include design of all themes, although we include the `enroll` (cf. Fig. 6) and `join` (cf. Fig. 7) base themes and the `adapt-ui` aspect theme (cf. Figs. 9 and 10) as examples of themes designed for the auction system application. We will refer to these themes throughout the remainder of the case study overview.

After completing the design, we applied the composition relationships to the themes and their elements to create a specification that would indicate the integration of all the themes. Figure 8 illustrates a merge between the two base themes, `enroll` and `join`. The merged theme is given a name, `auctionSystem`, through the use of the `themeName` tag definition. Examination of the base themes reveals that we generally used the same vocabulary to model the same concepts, and so a `match[name]` matching criterion is attached to match elements with

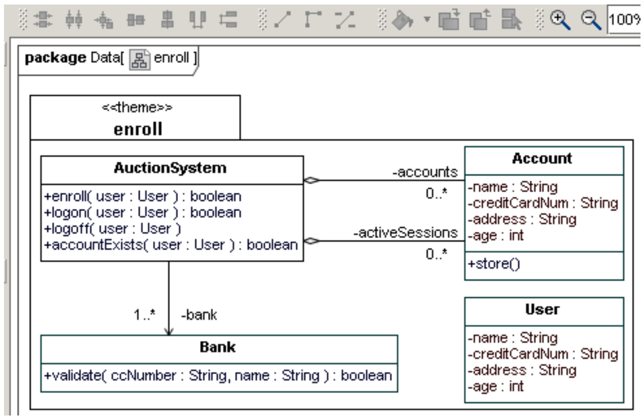


**Fig. 6.** MagicDraw screenshot of the `enroll` theme

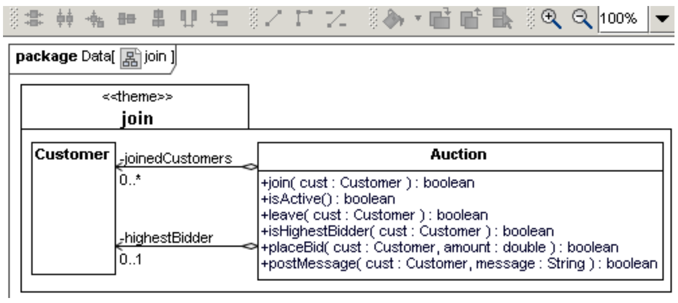

package Data[ join ]
<<theme>>
join
Customer
-joinedCustomers
0..*
-highestBidder
0..1
Auction
+join( cust : Customer ) : boolean
+isActive() : boolean
+leave( cust : Customer ) : boolean
+isHighestBidder( cust : Customer ) : boolean
+placeBid( cust : Customer, amount : double ) : boolean
+postMessage(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

**Fig. 7.** MagicDraw screenshot of the `join` theme

the same name and type. During this process, the concept of `User` in the `enroll` theme was found to be the same as that of `Customer` in the `join` theme. An explicit composition relationship was applied to resolve this conflict. This relationship specifies that the two classes are the same and that they should be merged under the unified `Customer` class.

Aspect themes can be integrated through the `bind` composition relationship. A `bind` is defined as a specialisation of a merge integration and supports merging of the structure and behaviour of an aspect theme with a base theme. Figures 9 and 10 illustrate the `adapt-ui` theme, along with its composition specification to the base themes `enroll` and `join`. As illustrated in Fig. 10[14], the sequence diagrams in aspect themes specify how (advice) and when (joinpoint) in relation to the abstract templates the crosscutting behaviour takes place. The

[14] The sequence diagram is not currently shown as part of the aspect theme due to the error with MagicDraw's sequence diagram export behaviour (see Sect. 2.3). We show a manually constructed sequence diagram as well as part of the UML2 tree editor's view of the behaviour under discussion.

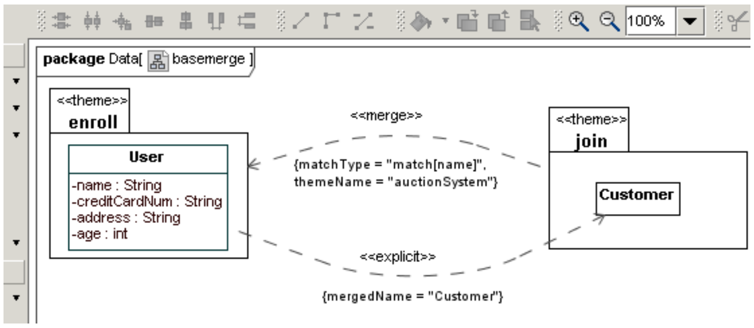


**Fig. 8.** MagicDraw screenshot of the base merge composition specification

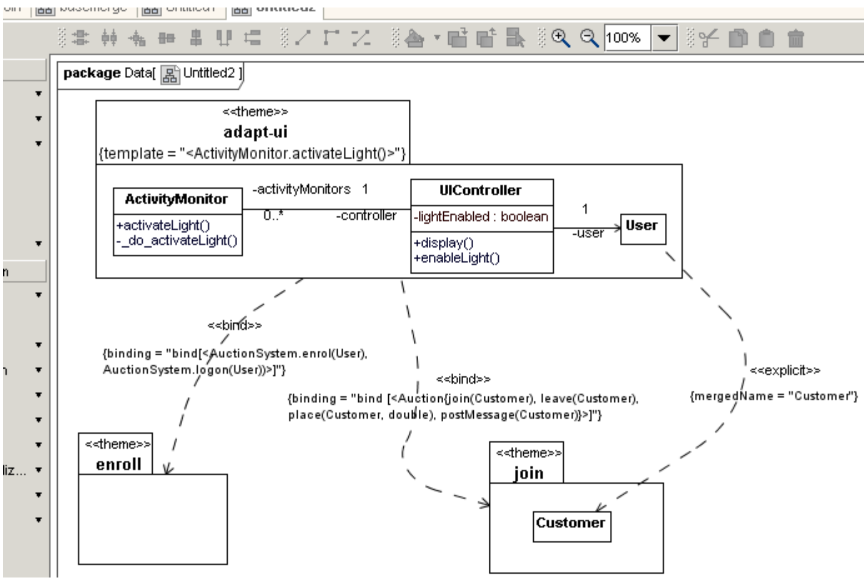
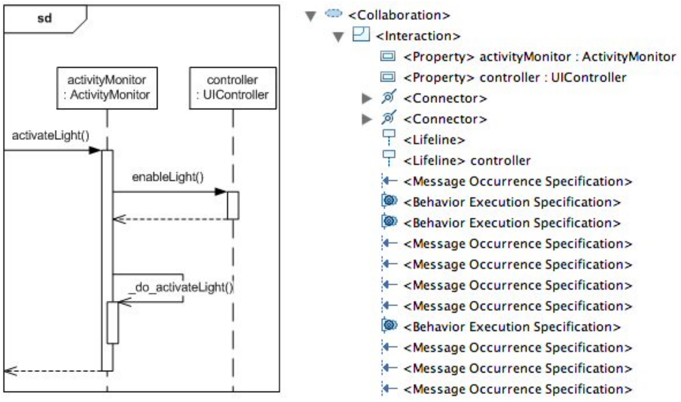

100%
package Data[ Untitled2 ]
<<theme>>
adapt-ui
{template = "<ActivityMonitor.activateLight()>"}
ActivityMonitor
+activateLight()
-_do_activateLight()
-activityMonitors 1
0..*
-controller
UIController
-lightEnabled : boolean
+display()
+enableLight()
1
-user
User
<<bind>>
{binding = "bind[<AuctionSystem.enrol(User),
AuctionSystem.logon(User)>]"}
<<bind>>
{binding = "bind [<Auction{join(Customer), leave(Customer),
place(Customer, double), postMessage(Customer)}>]"}
<<explicit>>
{mergedName = "Customer"}
<<theme>>
enroll
<<theme>>
join
Customer

**Fig. 9.** MagicDraw screenshot of `adapt-ui` and its composition specification

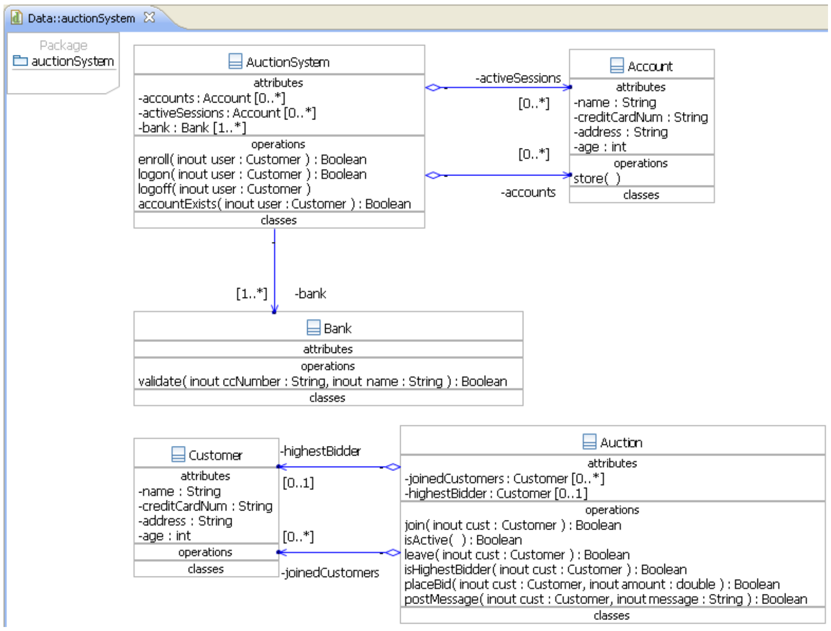
`activateLight()` joinpoint in the `adapt-ui` theme acts as a placeholder to the operations identified in the `bind` statement. It is these operations that actually trigger activation of the UI backlight following the base-aspect merge.

At the composition phase, we used the tool to take the themes and related composition relationships and merged them. The result of this composition specification, applied in Fig. 8, is depicted in Fig. 11. For ease of illustration, we only show the result of the bases being merged. Figures 12 and 13 show the result of the full composition produced by the tool, i.e. the composition specification applied in Fig. 8 and Fig. 9. The classes that were shared among multiple themes



<Collaboration>
<Interaction>
<Property> activityMonitor : ActivityMonitor
<Property> controller : UIController
<Connector>
<Connector>
<Lifeline>
<Lifeline> controller
<Message Occurrence Specification>
<Behavior Execution Specification>
<Behavior Execution Specification>
<Message Occurrence Specification>
<Message Occurrence Specification>
<Message Occurrence Specification>
<Message Occurrence Specification>
<Behavior Execution Specification>
<Message Occurrence Specification>
<Message Occurrence Specification>
<Message Occurrence Specification>

**Fig. 10.** Two views of the `adapt-ui` crosscutting behaviour


Data::auctionSystem

Package
auctionSystem

AuctionSystem
attributes
-accounts : Account [0..*]
-activeSessions : Account [0..*]
-bank : Bank [1..*]
operations
enroll( inout user : Customer ) : Boolean
logon( inout user : Customer ) : Boolean
logoff( inout user : Customer )
accountExists( inout user : Customer ) : Boolean
classes

Account
attributes
-name : String
-creditCardNum : String
-address : String
-age : int
operations
store( )
classes

Bank
attributes
operations
validate( inout ccNumber : String, inout name : String ) : Boolean
classes

Customer
attributes
-name : String
-creditCardNum : String
-address : String
-age : int
operations
classes

Auction
attributes
-joinedCustomers : Customer [0..*]
-highestBidder : Customer [0..1]
operations
join( inout cust : Customer ) : Boolean
isActive( ) : 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
classes

**Fig. 11.** Screenshot of the merged base

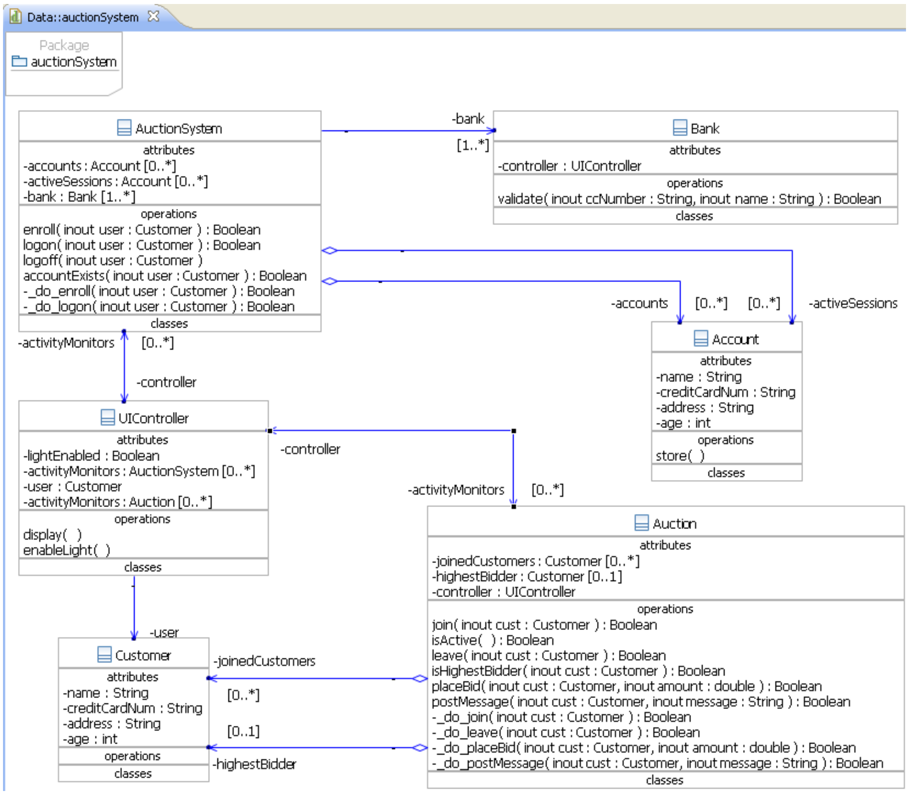

Data::auctionSystem
Package
auctionSystem
AuctionSystem
attributes
-accounts : Account [0..*]
-activeSessions : Account [0..*]
-bank : Bank [1..*]
operations
enroll( inout user : Customer ) : Boolean
logon( inout user : Customer ) : Boolean
logoff( inout user : Customer )
accountExists( inout user : Customer ) : Boolean
-_do_enroll( inout user : Customer ) : Boolean
-_do_logon( inout user : Customer ) : Boolean
classes
-bank
[1..*]
Bank
attributes
-controller : UIController
operations
validate( inout ccNumber : String, inout name : String ) : Boolean
classes
-accounts
[0..*]
[0..*]
-activeSessions
Account
attributes
-name : String
-creditCardNum : String
-address : String
-age : int
operations
store( )
classes
-activityMonitors
[0..*]
-controller
UIController
attributes
-lightEnabled : Boolean
-activityMonitors : AuctionSystem [0..*]
-user : Customer
-activityMonitors : Auction [0..*]
operations
display( )
enableLight( )
classes
-controller
-activityMonitors
[0..*]
Auction
attributes
-joinedCustomers : Customer [0..*]
-highestBidder : Customer [0..1]
-controller : UIController
operations
join( inout cust : Customer ) : Boolean
isActive( ) : 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
-_do_join( inout cust : Customer ) : Boolean
-_do_leave( inout cust : Customer ) : Boolean
-_do_placeBid( inout cust : Customer, inout amount : double ) : Boolean
-_do_postMessage( inout cust : Customer, inout message : String ) : Boolean
classes
-user
Customer
attributes
-name : String
-creditCardNum : String
-address : String
-age : int
operations
classes
-joinedCustomers
[0..*]
[0..1]
-highestBidder

**Fig. 12.** Screenshot of merging `adapt-ui` with the `enroll` and `join` themes

have been unified, e.g. the resultant merge of the same class has all the operations belonging to separate versions of that class before the merge. Also, there is no `User` class as it has been merged with its new name, `Customer`.

The aspect theme was also composed with the base themes. For example, the `ActivityMonitor` behaviour in the `adapt-ui` theme gets merged with the `AuctionSystem` through the binding to the `enroll` theme. The `logon` and `enroll` operations are renamed to `_do_logon` and `_do_enrol`, respectively. The new logon and enroll operations now contain the crosscutting behaviour that they have been merged with. The case is similar for the `join` theme.

With the object-oriented composition of themes completed and no refactoring necessary, it was possible to produce a PSM. In the transformation phase, we choose both available target platforms, J2ME and .NET CF. The tool was used to transform the object-oriented design produced in the previous phase into the two target PSMs, adding in more concrete detail for each specific platform as appropriate. Figure 14 illustrates the J2ME PSM produced during the

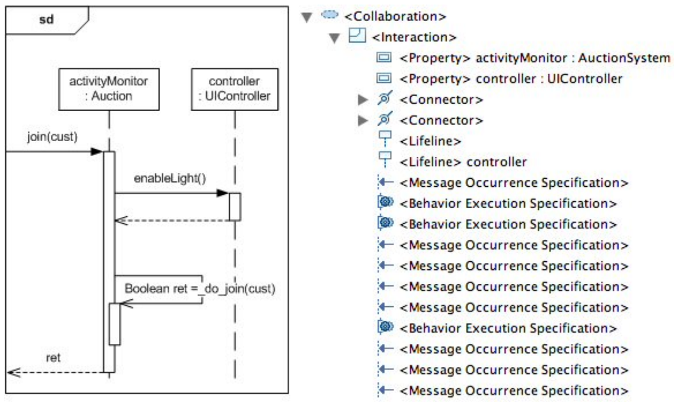

sd
activityMonitor : Auction
controller : UIController
join(cust)
enableLight()
Boolean ret = _do_join(cust)
ret
<Collaboration>
<Interaction>
<Property> activityMonitor : AuctionSystem
<Property> controller : UIController
<Connector>
<Connector>
<Lifeline>
<Lifeline> controller
<Message Occurrence Specification>
<Behavior Execution Specification>
<Behavior Execution Specification>
<Message Occurrence Specification>
<Message Occurrence Specification>
<Message Occurrence Specification>
<Message Occurrence Specification>
<Behavior Execution Specification>
<Message Occurrence Specification>
<Message Occurrence Specification>
<Message Occurrence Specification>

**Fig. 13.** Two views of the merged `adapt-ui` crosscutting behaviour

transformation process, depicting the modified datatypes for J2ME platform and the automatically generated accessors and mutators.

At this point, either the J2ME PSM or the .NET CF PSM could be inspected. As a PSM is refined from a computationally incomplete PIM (i.e. the approach is elaboration-oriented), it was necessary to further elaborate the model both structurally and behaviourally using platform-specific library extensions. Either PSM can be elaborated partially or to completion at the model level, with the remaining elaboration achieved through code. After elaboration, the PSM was ready for synthesis, i.e. transformation to source code. The J2ME and .NET CF source code that was automatically generated for the `join` method (which includes crosscutting `adapt-ui` behaviour) is illustrated in Fig. 15.

## 4.1 Discussion

We observed from this case study that Model-Driven Theme/UML has a positive impact on system modularity when applied to the development of an application with crosscutting mobility and context-awareness concerns. Theme/UML facilitated the separation of concerns at design time that would have otherwise resulted in scattering and tangling in core system behaviour. Through the specification of composition relationships between modularised concerns, it was possible to produce a design with which the tool could operate. Given a collection of modules and a description of their relationships, the tool automatically generated platform-specific models for J2ME and .NET CF platforms. The tool then used these PSMs to generate source code for the respective target platforms, saving time and reducing the risk of error introduction. The tool supports a solution-focused development approach that allows developers to concentrate on the design of the initial model and avail the benefits of automatic PSM and code generation.

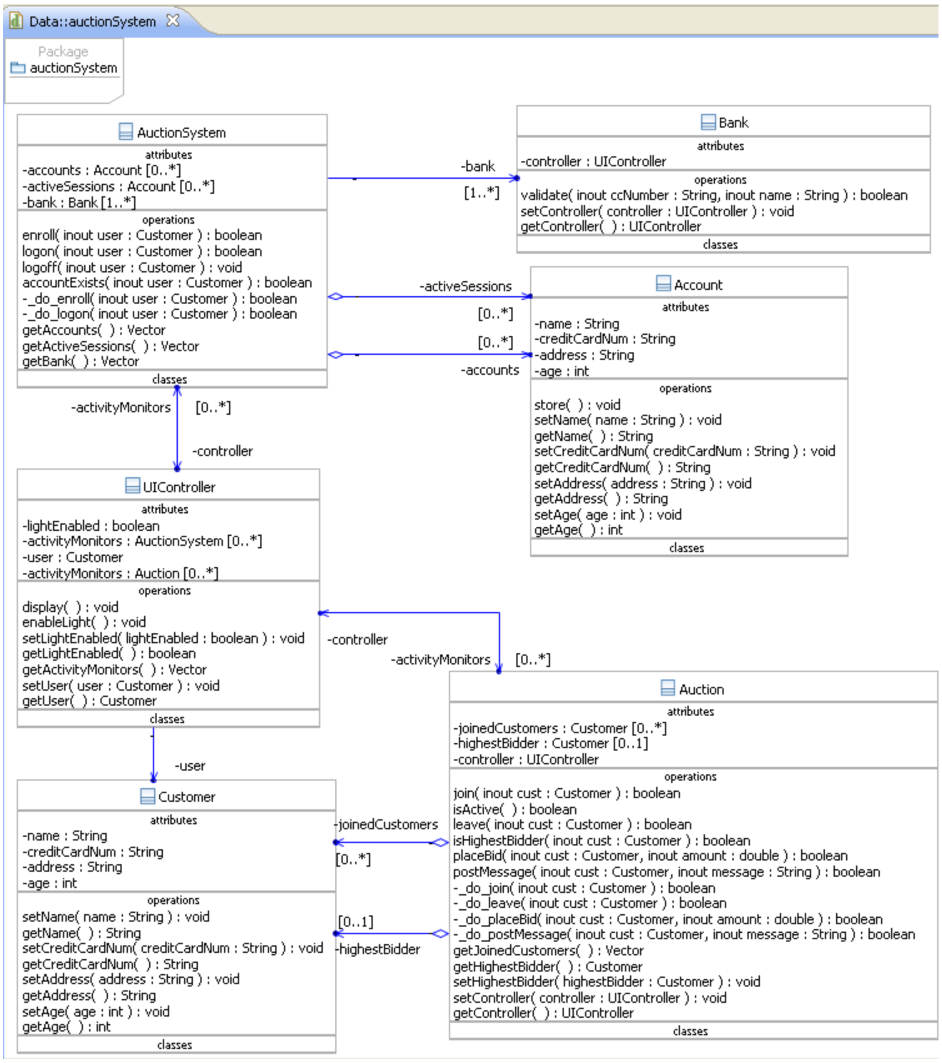
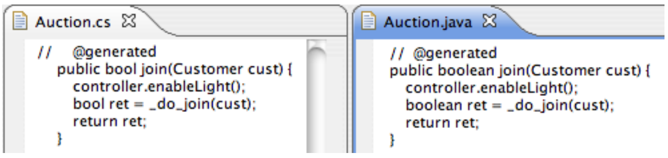

Data::auctionSystem
Package
auctionSystem
AuctionSystem
attributes
-accounts : Account [0..*]
-activeSessions : Account [0..*]
-bank : Bank [1..*]
operations
enroll( inout user : Customer ) : boolean
logon( inout user : Customer ) : boolean
logoff( inout user : Customer ) : void
accountExists( inout user : Customer ) : boolean
-_do_enroll( inout user : Customer ) : boolean
-_do_logon( inout user : Customer ) : boolean
getAccounts( ) : Vector
getActiveSessions( ) : Vector
getBank( ) : Vector
classes
-bank
[1..*]
Bank
attributes
-controller : UIController
operations
validate( inout ccNumber : String, inout name : String ) : boolean
setController( controller : UIController ) : void
getController( ) : UIController
classes
-activeSessions
[0..*]
[0..*]
-accounts
Account
attributes
-name : String
-creditCardNum : String
-address : String
-age : int
operations
store( ) : void
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
-activityMonitors
[0..*]
-controller
UIController
attributes
-lightEnabled : boolean
-activityMonitors : AuctionSystem [0..*]
-user : Customer
-activityMonitors : Auction [0..*]
operations
display( ) : void
enableLight( ) : void
setLightEnabled( lightEnabled : boolean ) : void
getLightEnabled( ) : boolean
getActivityMonitors( ) : Vector
setUser( user : Customer ) : void
getUser( ) : Customer
classes
-controller
-activityMonitors
[0..*]
-user
Customer
attributes
-name : String
-creditCardNum : String
-address : String
-age : int
operations
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
-joinedCustomers
[0..*]
[0..1]
-highestBidder
Auction
attributes
-joinedCustomers : Customer [0..*]
-highestBidder : Customer [0..1]
-controller : UIController
operations
join( inout cust : Customer ) : boolean
isActive( ) : 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
-_do_join( inout cust : Customer ) : boolean
-_do_leave( inout cust : Customer ) : boolean
-_do_placeBid( inout cust : Customer, inout amount : double ) : boolean
-_do_postMessage( inout cust : Customer, inout message : String ) : boolean
getJoinedCustomers( ) : Vector
getHighestBidder( ) : Customer
setHighestBidder( highestBidder : Customer ) : void
setController( controller : UIController ) : void
getController( ) : UIController
classes

**Fig. 14.** Screenshot of the platform-specific J2ME model



```
//   @generated
  public bool join(Customer cust) {
    controller.enableLight();
    bool ret = _do_join(cust);
    return ret;
  }
```

```
// @generated
public boolean join(Customer cust) {
  controller.enableLight();
  boolean ret = _do_join(cust);
  return ret;
}
```

**Fig. 15.** Source code generated from the .NET CF (left) and J2ME (right) PSMs

## 5 Related work

Composition Directives [24,27] is an approach implemented in Kermeta called Kompose [6,7] and supports the composition of both aspect and base UML models. This work takes a hybrid symmetry approach to merging, i.e. the composition procedure does not distinguish between an aspect and a base model, and was designed to deal with the inadequacies of a simple name-based matching strategy. For example, when merging two operations with the same name but different argument lists or return values, a simple name-based matching strategy would produce a merged result using just the names as matches. The Composition Directives approach supports different model elements having unique, sensible matching strategies, according to their syntactic properties. To accomplish this, a composition metamodel was devised. The idea of a composition metamodel in our work was originally inspired by this approach, but we subsequently focused on supporting the original definition of the Theme/UML semantics. The similarities include an abstraction of the matching criterion, as well as an enumeration of the composition elements. In terms of differences, contrasting composition algorithms are employed. Theme/UML defines an abstract integration type and therefore the composition algorithm is iterative. Alternatively, Composition Directives defines a single merge implemented as a recursive composition algorithm.

The Atlas Model Weaver (AMW)[15] is a tool that facilitates the creation of links between models [4]. 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 that conforms to a weaving metamodel. AMW can be used to support aspect weaving[16], although it is not centred specifically around the notion of aspect-orientation. While our approach specifies a metamodel that defines how models get composed, AMW defines a metamodel for weaving links between models. It allows models to be visualised in a tree-like manner and supports the association of links between two metamodels or models using the weaving metamodel. It also defines the notion of a weaving session in which the weaving metamodel, the models and their metamodels are loaded and links are defined and woven. Contrary to this approach, our approach uses a UML profile to define the weaving/composition relationships at modelling time. The AMW weaving process does not distinguish between primary and aspect models, making it purely symmetric.

The Motorola WEAVR [2] is a commercial add-in to the Telelogic TAU tool[17] and is designed for use in telecoms systems engineering. WEAVR 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 our approach, which is elaboration-based, WEAVR is a translation-based approach that uses state machines and an action language to fully specify the application

[15] `http://www.eclipse.org/gmt/amw`

[16] `http://www.eclipse.org/gmt/amw/usecases/AOM`

[17] `http://www.telelogic.com`

logic at the model level. Similar to our approach, it uses a UML Profile to specify aspect-oriented extensions. For example, to illustrate an aspect, a class is extended with the `aspect` stereotype, allowing tagged definitions in the form of attributes, operations, signal definitions and ports, which are treated like inter-type declarations. Furthermore, it allows precedence of connectors to be applied to the same pointcut, aiding the management of aspect interference. This feature is not catered for in Theme/UML.

XWeave is a model weaver that supports composition of different architectural viewpoints. The weaver facilitates software product-line engineering, allowing for variable parts of architectural models to be woven according to a specific product configuration [9]. Xweave adopts a form of asymmetric aspect-orientation, unlike Theme/UML, which defines both symmetric and asymmetric forms. 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 elements have the same name and type. This is similar to the matching criterion defined in our composition metamodel. Pointcut expression weaving is based on the oAW expression language, which is itself similar to OCL. This approach is more powerful than the wildcard-based string selection mechanism used by Theme/UML. One drawback of the XWeave approach is the limited support for advice. Base model elements cannot be removed, changed or overriden by aspect models and hence they only support additive weaving. Theme/UML supports these features through the semantics of its integration strategies.

Modelling Aspects Using a Transformation Approach (MATA) [15] is a UML aspect-oriented modelling tool. Unlike our approach, which is based on model composition, MATA uses graph transformations to specify and compose aspects. Using the UML metamodel as a type graph, any UML model can therefore be represented as an instance of this type graph and a transformation based on graph theory applied on it. The tool currently supports class, sequence and state diagrams. The aspect model consists of a set of graph rules that can be applied as a graph transformation to the base model using a pattern. MATA is built on top of IBM's Rational Software Modeler and uses the graph rule execution tool AGG as a back-end for graph transformations.

Klein et al. [16] suggest an approach for weaving multiple behavioural aspects using sequence diagrams. In their approach, a base scenario describes the behaviour of the system using a sequence diagram, and a behavioural aspect describes a concern that crosscuts this base scenario. They propose various types of pointcut, allowing joinpoints to be matched even when extra messages occur in between and also demonstrate how these can be statically woven. This approach formally defines a more concise custom metamodel and addresses the semantic difficulty of explicitly composing one sequence diagram with another. Although this approach differs from Theme/UML in that it supports asymmetric separation, it is considered a complimentary approach that could be integrated to enhance Theme/UML's support for behavioural modelling.

## 6 Summary and Future Work

In this paper we have presented our efforts to integrate AOSD techniques with the MDE process. We have described new tool support for model-driven Theme/UML from both an implementation and a methodological perspective, and illustrated the capabilities of the tool by means of a case study.

We are currently investigating revisions and extensions to the tool to support both the modularisation of distributed, real-time embedded (DRE) concerns at the model level and transformations to embedded platforms. In addition to this work, we are developing an aspect-oriented MDE tool suite. The tool suite combines the work described in this paper with similar work that was conducted in tandem. This related work provides similar capabilities in terms of modularisation of concerns at the model-level, but differs from the approach described here in terms of the types of transformations supported.

## Acknowledgments

We would like to acknowledge the support of AOSD-Europe and of Lero: The Irish Software Engineering Research Centre, funded by Science Foundation Ireland. Thanks also to Jorge Fox for his comments on earlier drafts of this paper.

## References

1. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design. The Theme Approach. Object Technology Series. Addison-Wesley, Boston (2005)
2. Cottenier, T., van den Berg, A., Elrad, T.: The Motorola WEAVR: Model Weaving in a Large Industrial Context (2007)
3. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture (October 2003)
4. Didonet Del Fabro, M., Bézivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: a generic model weaver. In: Journées sur l'Ingénierie Dirigée par les Modèles (IDM 2005), pp. 105–114 (2005)
5. Filman, R.E., Elrad, T., Clarke, S., Akşit, M. (eds.): Aspect-Oriented Software Development. Addison-Wesley, Boston (2005)
6. Fleurey, F., Baudry, B., France, R., Ghosh, S.: A generic approach for automatic model composition. In: Aspect Oriented Modeling (AOM) Workshop, Nashville, USA (October 2007)
7. France, R., Fleurey, F., Reddy, R., Baudry, B., Ghosh, S.: Providing support for model composition in metamodels. In: EDOC 2007: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference, Washington, DC, USA, p. 253. IEEE Computer Society, Los Alamitos (2007)
8. France, R.B., Ghosh, S., Dinh-Trong, T., Solberg, A.: Model-Driven Development Using UML 2.0: Promises and Pitfalls. Computer 39(2), 59 (2006)
9. Groher, I., Voelter, M.: XWeave: models and aspects in concert. In: AOM 2007: Proceedings of the 10th international workshop on Aspect-Oriented Modeling, pp. 35–40. ACM Press, New York (2007)

10. Object Management Group. Model-Driven Architecture, http://www.omg.org/mda (accessed October 22, 2007)
11. Object Management Group. OMG UML Specification Version 1.3., ftp://ftp.omg.org/pub/docs/ad/99-06-03.pdf (accessed October 25, 2007)
12. Hailpern, B., Tarr, P.: Model-driven development: the good, the bad, and the ugly. IBM Systems Journal 45(3), 451–461 (2006)
13. Jackson, A., Barais, O., Jézéquel, J.-M., Clarke, S.: Toward A Generic And Extensible Merge. In: Models and Aspects workshop, at ECOOP 2006, Nantes, France (2006)
14. Jackson, A., Clarke, S.: Towards a Generic Aspect Oriented Design Process. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 110–119. Springer, Heidelberg (2006)
15. Jayaraman, P.K., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 151–165. Springer, Heidelberg (2007)
16. Klein, J., Fleurey, F., Jézéquel, J.-M.: Weaving multiple aspects in sequence diagrams. In: Rashid, A., Aksit, M. (eds.) Transactions on AOSD III. LNCS, vol. 4620, pp. 167–199. Springer, Heidelberg (2007)
17. Object Management Group. UML 2.0 Infrastructure Specification, http://www.omg.org/docs/ptc/03-09-15.pdf (accessed October 25, 2007)
18. McNeile, A.: MDA: The Vision with the Hole, http://www.metamaxim.com/download/documents/MDAv1.pdf (accessed October 30, 2007)
19. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Longman Publishing Co., Inc., Boston (2002); foreword By-Ivar Jacoboson
20. Mens, T., Czarnecki, K., Van Gorp, P.: Discussion – A Taxonomy of Model Transformations. In: Bezivin, J., Heckel, R. (eds.) Language Engineering for Model-Driven Software Development. Dagstuhl Seminar Proceedings, vol. 04101, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2005)
21. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG) (2003)
22. OMG. MDA Guide Version 1.0.1, http://www.omg.org/docs/omg/03-06-01.pdf (accessed November 2, 2007)
23. Ossher, H., Kaplan, M., Katz, A., Harrison, W., Kruskal, V.: Specifying subject-oriented composition. Theory and Practice of Object Systems 2(3), 179–202 (1996)
24. Reddy, Y.R., Ghosh, S., France, R.B., Straw, G., Bieman, J.M., McEachen, N., Song, E., Georg, G.: Directives for Composing Aspect-Oriented Design Class Models, pp. 75–105 (2006)
25. Sánchez, P., Fuentes, L., Jackson, A., Clarke, S.: Aspects at the Right Time. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development IV. LNCS, vol. 4640, pp. 54–113. Springer, Heidelberg (2007)
26. Schilit, B., Adams, N., Want, R.: Context-Aware Computing Applications. In: Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, US, pp. 85–90. IEEE Computer Society, Los Alamitos (1994)
27. Straw, G., Georg, G., Song, E., Ghosh, S., France, R.B., Bieman, J.M.: Model composition directives. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 84–97. Springer, Heidelberg (2004)

28. Walker, R.J.: Eliminating cycles in composed class hierarchies. Technical Report TR-2000-07, University of British Columbia (2000)
29. Zito, A., Dingel, J.: Modeling UML 2 Package Merge With Alloy. In: 1st Alloy Workshop (Alloy 2006), Portland, OR, USA, pp. 86–95 (2006)
30. Zito, A., Diskin, Z., Dingel, J.: Package Merge in UML 2: Practice vs. Theory? In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 185–199. Springer, Heidelberg (2006)

## Appendix

## A Theme/UML Overview

*The Theme Approach* is an aspect-oriented methodology that encompasses the requirements analysis, design and mapping to implementation phases of the development lifecycle [1]. Theme/Doc provides a systematic means to analyse a text-based requirements specification in order to identify base and crosscutting concerns and the relationships between them. Theme/UML is an aspect-oriented modelling language that supports the design of concerns and maintains the relationships previously identified by Theme/Doc. The Theme Approach also details mapping specifications from Theme/UML to aspect-oriented programming languages such as AspectJ.

Theme/UML is aspect-oriented design language with an accompanying methodology. The Theme/UML design language is a Meta-Object Facility (MOF) extension of the UML 1.3 beta R7, enhancing standard UML with new modularisation and compositional constructs. The accompanying methodology provides guidelines on the use of these new constructs. The constructs include a new type of classifier called a *theme*, a composition relationship and three integration strategies-*merge*, *override* and *bind*.

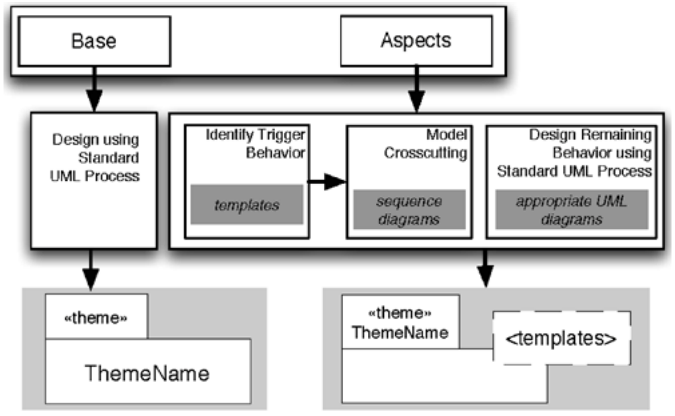


**Fig. 16.** Designing with Theme/UML

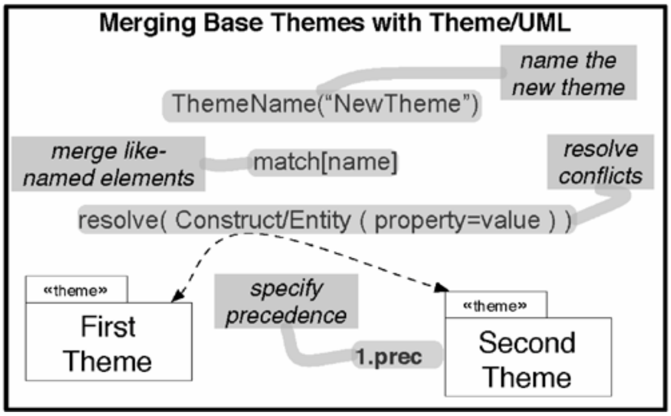


**Fig. 17.** Merge Integration Strategy

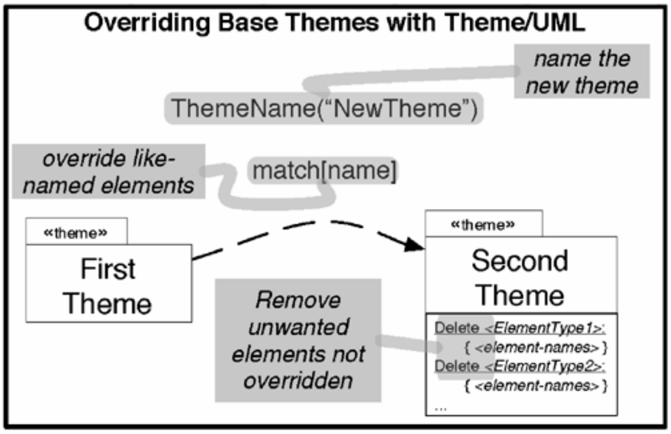


**Fig. 18.** Override Integration Strategy

A theme is a construct based on the existing definition of the standard UML *package* and encapsulates the design specification of a base or aspect concern. As illustrated in Fig. 16, a base theme is designed using the standard UML process and can include any of the standard diagram types. An aspect theme is one that encapsulates a crosscutting concern and is designed relative to the abstract templates, with sequence diagrams specifying when and how the templates interact with the base themes.

As Theme/UML aligns to a symmetric decomposition, themes are considered to be declaratively complete. This means that the design specification of a concern is self-contained and does not reference anything outside the theme in which it is defined. This property allows a more rigorous separation of individual themes from each other. Consequently, this property may result in overlapping

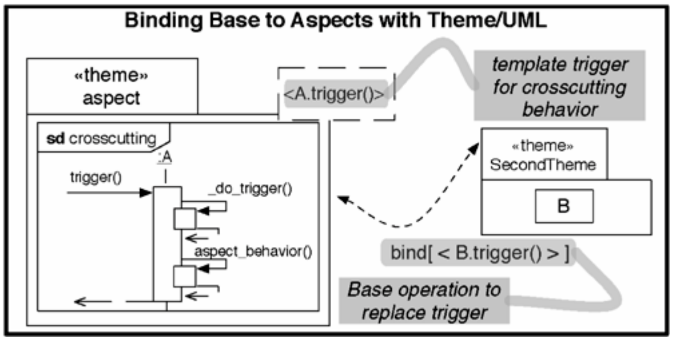


**Fig. 19.** Bind Integration Strategy

concepts being represented in multiple individual theme designs. Consequently, these concepts must be reconciled at composition time.

Theme/UML supports compositional constructs for both overlapping and crosscutting specifications. An overlapping or shared concept can arise because equivalent concepts can be considered in more than one theme. A *merge* integration strategy can exist between two or more themes and allows like-named elements to be matched, thereby resolving conflicts between themes. Figure 17 illustrates a merge between two themes. The *match[name]* property indicates that elements are to be matched and merged based on name and type. *ThemeName("NewTheme")* indicates that the result of the merged themes will produce a new theme called *NewTheme*. To achieve resolution of conflicts, Theme/UML supports three reconciliation strategies. The first strategy, *prec*, indicates the precedence of each theme's design specification in the merge. Figure 17 illustrates that the second theme has a higher precedence than the first theme, and therefore, its design specification will get priority in the merge. The second reconciliation strategy is an *explicit* reconciliation that takes the form *resolve(Entity (property = value))* and allows any property of any specific *Entity* in a theme to be assigned a *value*. The third reconciliation strategy is a *default* reconciliation and has a similar form, with *Construct* replacing the *Entity* instead (c.f. Fig. 17). In this case, any property of a UML construct (e.g. operation visibility kind) can be given a value (e.g. private) and this reconciliation gets executed during the merge. The second kind of composition extension that Theme/UML supports for overlapping specifications is called an *override*. An *override*, as indicated in Fig. 18, is a relationship between two themes where one theme's design specification overrides the other. The semantics of the integration properties are similar to the merge. One difference is that elements can be explicitly indicated to be deleted in a theme prior to the merge.

For crosscutting specifications, an integration strategy called a *bind* facilitates the composition of an aspect theme with a base theme. Figure 19 depicts an aspect theme being bound to a base theme. The aspect theme is designed in relation to the abstract templates. In this example, the triggering template operation is called *A.trigger()*. The sequence diagram illustrates the behaviour of the aspect theme in relation to this triggering behaviour. The operation *_do_trigger()* encapsulates the existing behaviour of the operation in the base theme that is bound to the template method in the aspect 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 specification represents the instantiation of the aspect theme. The operation *B.trigger()* is the operation being bound to, and the triggering template operation is replaced with this method upon the aspect's instantiation.