

UPGRADE is the European Journal for the Informatics Professional, published bimonthly at <http://www.upgrade-cepis.org/>

Publisher

UPGRADE is published on behalf of CEPIs (Council of European Professional Informatics Societies, <http://www.cepis.org/>) by NOVÁTICA <http://www.ati.es/novatica/>, journal of the Spanish CEPIs society ATI (Asociación de Técnicos de Informática <http://www.ati.es/>).

UPGRADE is also published in Spanish (full issue printed, some articles online) by NOVÁTICA, and in Italian (abstracts and some articles online) by the Italian CEPIs society ALSI <http://www.alsi.it/> and the Italian IT portal Tecnoteca <http://www.tecnoteca.it/>.

UPGRADE was created in October 2000 by CEPIs and was first published by NOVÁTICA and INFORMATIK/INFORMATIQUE, bimonthly journal of SVI/FSI (Swiss Federation of Professional Informatics Societies, <http://www.svifsi.ch/>).

Editorial Team

Chief Editor: Rafael Fernández Calvo, Spain, rfoalvo@ati.es
Associate Editors:

- François Louis Nicolet, Switzerland, nicolet@acm.org
- Roberto Carniel, Italy, rcarniel@dgt.uniud.it

Editorial Board

Prof. Wolfried Stucky, CEPIs Past President
Prof. Nello Scarabottolo, CEPIs Vice President
Fernando Piera Gómez and
Rafael Fernández Calvo, ATI (Spain)
François Louis Nicolet, SI (Switzerland)
Roberto Carniel, ALSI – Tecnoteca (Italy)

English Editors: Mike Andersson, Richard Butchart, David Cash, Arthur Cook, Tracey Darch, Laura Davies, Nick Dunn, Rodney Fennemore, Hilary Green, Roger Harris, Michael Hird, Jim Holder, Alasdair MacLeod, Pat Moody, Adam David Moss, Phil Parkin, Brian Robson.

Cover page designed by Antonio Crespo Foix, © ATI 2003

Layout: Pascale Schürmann

E-mail addresses for editorial correspondence:
rfoalvo@ati.es, nicolet@acm.org or
rcarniel@dgt.uniud.it

E-mail address for advertising correspondence:
novatica@ati.es

Upgrade Newsletter available at

<http://www.upgrade-cepis.org/pages/editinfo.html#newsletter>

Copyright

© NOVÁTICA 2004. All rights reserved. Abstracting is permitted with credit to the source. For copying, reprint, or republication permission, write to the editors.

The opinions expressed by the authors are their exclusive responsibility.

ISSN 1684-5285

2 From the Editors' Desk

The UPGRADE European Network: *N przywitanie* / Welcome!

The members of the Editorial Team of UPGRADE describe the aims and scope of the network of journals of CEPIs member societies, whose contents will enrich ours and offer a broader European view of ICT to our readership.

UML and Model Engineering

Guest Editors: Jesús García-Molina, Ana Moreira, and Gustavo Rossi

Joint issue with NOVÁTICA*

3 Presentation

UML: The Standard Object Modelling Language – *Jesús García-Molina, Ana Moreira, and Gustavo Rossi*

The guest editors introduce the monograph, that includes a series of papers that reflect the state of the art of UML (Unified Modeling Language). These papers illustrate different aspects of UML, ranging from use cases to UML formalization, meta-modelling, profile definition, model quality, model engineering and MDA (Model Driven Architecture).

6 An Introduction to UML Profiles – *Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno*

This paper describes a set of steps to create a profile and argue the importance of profiles in MDA.

14 Aspect-Oriented Design with Theme/UML – *Siobhán Clarke*

The author describes her approach “Theme” to extending the UML in order to support the modularisation of a designer’s concerns, including crosscutting ones.

21 In Search of a Basic Principle for Model Driven Engineering – *Jean Bézivin*

This article offers an interesting look at the essential features of this new software development paradigm.

25 The Object Constraint Language for UML 2.0 – Overview and Assessment – *Heinrich Hussmann and Steffen Zschaler*

This paper, authored by members of the OCL 2.0 team, gives an overview of the new aspects of the second version of this language and also provides a critical discussion of a few selected aspects of it.

29 Developing Security-Critical Applications with UMLsec. A Short Walk-Through – *Jan Jürjens*

The problems of creating high-quality critical systems is analysed in this paper, that shows how using UML modelling can help solve them and presents a tool to support the proposed approach.

36 ON the Nature of Use Case-Actor Relationships – *Gonzalo Génova-Fuster and Juan Llorens-Morillo*

In this paper some issues are addressed that regard the relationships in which use cases and actors may take part, presently defined in UML as associations.

43 Metrics for UML Models – *Marcela Genero, Mario Piattini-Velthuis, José-Antonio Cruz-Lemus, and Luis Reynoso*

This paper offer a vision of the state of the art of metrics for measuring quality of some basic UML diagrams (such as class, state and use case diagrams) and OCL expressions.

49 Using Refactoring and Unification Rules to Assist Framework Evolution – *Mariela I. Cortés, Marcus Fontoura, and Carlos J.P. de Lucena*

In their paper the authors use UML-F, a UML designed for describing frameworks, to present two techniques aimed at facilitating framework maintenance and evolution.

Mosaic

UPGRADE European Network

From “Pro Dialog” (Poland):

56 Parallel Programming Support System for Transputers – Educational Software – *Mikolaj Szczepanski and Rafal Walkowiak*

The paper presents a method for integrating applications data, aimed at data aggregation and transfer in software applications when integration of those applications has to be fast and should be done with minimum source code modifications.

News Sheet

61 ENISA: The European Network and Information Security Agency created

61 News from EUCIP and ECDL

Next issue (June 2004):

“Digital Signature”

(The full schedule of UPGRADE is available at our website)

* This monograph will be also published in Spanish (full issue printed; summary, abstracts and some articles online) by NOVÁTICA, journal of the Spanish CEPIs society ATI (Asociación de Técnicos de Informática) at <http://www.ati.es/novatica/>, and in Italian (online edition only, containing summary abstracts and some articles) by the Italian CEPIs society ALSI and the Italian IT portal Tecnoteca at <http://www.tecnoteca.it/>.

Aspect-Oriented Design with Theme/UML

Siobhán Clarke

Aspect-oriented software development represents a set of emerging technologies that seek to extend the modularisation capabilities of existing software development paradigms. For example, those concerns that ‘crosscut’ a system (i.e., have an impact across multiple modules of the system) can be designed and programmed separately, to be later composed into a running system. In the object-oriented paradigm, there are many examples of crosscutting concerns that cannot be modularised – classic ones are transaction handling, distribution support and audit trails. There has been significant success in aspect-oriented programming languages that are based on extending object-oriented programming languages. This paper describes an approach to extending the UML to support the modularisation of a designer’s concerns, including crosscutting ones.

Keywords: Aspects, Aspect-Oriented Design, Design Composition, Metamodels, UML.

1 Introduction

Modularisation using conventional object-oriented design models is by class, interface and method. This kind of modularisation matches well with object-oriented code, providing a measure of traceability between object-oriented designs and code. However, it does not align well with the structure of requirements specifications, which are generally described by feature and capability. There is a negative impact to this structural mismatch – support for individual requirements is scattered across the design, and support for multiple requirements is tangled in individual design units. This reduces comprehensibility and traceability, making designs difficult to develop, re-use and extend [2].

A new model is required that supports the designer in identifying the different requirements or features or concerns in a specification, and supports the modularisation of each into separate design models. This paper describes Theme/UML, which is an extension to the standard UML (Unified Modeling Language) to support this kind of modularisation. The design of the system is modularised into ‘themes’. A theme is any feature, or concern, or requirement of interest that has to be catered for in the system. Themes may relate to each other, in the same way as requirements or features are related to other parts of the system. This is likely to cause overlaps in the design models, which has important implications for a design language. We must assume that if we modularise in this manner, then we must be able to cater for relationships and overlaps between the model elements in the separated theme design models. Theme/UML recognises two kinds of overlap. The first is *concept sharing*, where different themes have design elements that represent the same core concepts in the domain. For example, “Add Customer Details” and “Take Customer Booking” requirements are both likely to have a notion of “customer”. Each theme will contain specifications for those

same concepts designed from the perspective of the theme. The second category of overlap is the classic aspect-oriented *cross-cutting*, where behaviour in one theme will be triggered in tandem with behaviour in other themes. Themes, therefore, represent a broader view of modularisation than the ‘aspects’ from many aspect-oriented programming approaches such as in [7], and are more closely related to multidimensional concerns from [11].

Theme/UML’s extensions to the UML centre around supporting the designer in specifying how overlapping themes relate to each other, and in the composition capabilities that are defined. In all aspect-oriented approaches, there must be a way to designate how the aspects relate to the rest of the system. To provide this capability, Theme/UML has defined a new kind of relationship, called a composition relationship that allows the

Siobhán Clarke is a lecturer in the Department of Computer Science at Trinity College, Dublin, Ireland. She holds a BSc from Dublin City University (1986), and subsequently worked for IBM as a software engineer for eleven years. Her PhD thesis (2001) was based on extending the decomposition and composition capabilities of object-oriented design languages, in particular, the UML. Her research interests are aspect-oriented software development, and the application of AOSD to design and programming models for mobile, context-aware computing. She is a member of the Distributed Systems Group at Trinity College, and lectures at undergraduate and postgraduate level on software engineering and related topics. Siobhán is on the Editorial Board of IEEE Internet Computing, and Editor of the Spotlight Column in the same journal. Siobhán has served on the program committee for the International Conferences on Aspect-Oriented Software Development (AOSD) from 2002 to 2004, and is on the organizing committee for AOSD 2004. She has co-organized and/or been on the program committee for multiple workshops at conferences such as OOPSLA, ECOOP, ICSE, AOSD and UML in the area of aspect-oriented software development.
<Siobhan.Clarke@cs.tcd.ie>

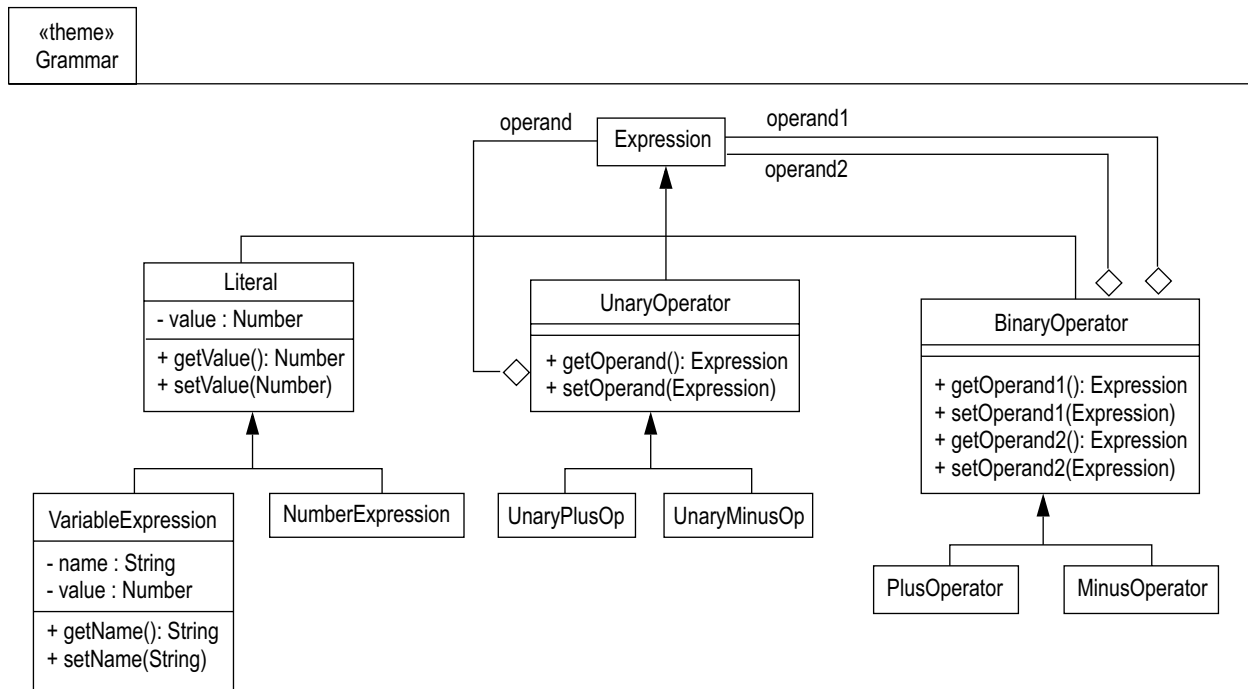


Figure 1-1: Grammar Theme.

designer identify those parts of the theme designs that relate to each other and therefore should be composed. For themes that crosscut others, this means identifying when and where in those other themes the additional behaviour should occur. For other kinds of overlaps, this means identifying elements in the theme designs that correspond to each other and saying how they should be integrated. Theme/UML also provides semantics for model composition based on the composition relationship specification. This supports a level of verification that allows the designer to consider the overall system design, including the composition specification, to ensure that it makes sense.

Theme/UML is part of an approach to aspect-oriented analysis and design called Theme. Theme has two parts: Theme/Doc is a set of heuristics and tools for visualisation and analysis of software requirements documentation for the purposes of finding the themes to be designed. Theme/UML is the second part, which is a way to design aspects based on the UML [1].

2 Theme/UML through an Example

The primary motivation for Theme/UML is provide a means to allow designers to reason about the concerns they may have, separately. Designers may work with separate design models for each of those concerns. Of course, since an executing system will have all those concerns working together, the designer will, at some point, need to be able to specify the relationships between the separate models and verify those decisions. Theme/UML, therefore, prescribes three high-level steps to add to the design process. First, separately design the themes. Second, specify the relationships between the theme designs. Finally, compose the themes for verification purposes.

In this section, we work through a small example of these three steps based on an abridged version of an expression evaluation system. Expressions have a small grammar that supports variables, literals, plus/minus unary operations and plus/minus binary operators. Supported operations over the expression are evaluation and checking of syntax. Operations over the expression are logged. For the purposes of this example, we will assume that there have been four themes identified. They are the grammar theme, the evaluation theme, the checking theme and the logging theme.

2.1 Theme Design

For the most part, standard UML and standard object-oriented design principles may be used to design the individual themes. Except for small extensions to the UML required for designing crosscutting themes, there is no additional notation or semantics in Theme/UML for individual theme design. In addition, any UML structural or behavioural model that is required for the specification of the theme may be included in the theme design. However, we consider that the use of Theme/UML is likely to be a lot simpler than standard UML, as it has the advantage of freeing the designer from having to worry about how an individual theme affects or is related to other themes.

2.1.1 Themes that Share Domain Concepts

In our expression evaluation system, three of the themes overlap from the perspective of sharing domain concepts. For example, the grammar must support a binary plus operator, and the evaluation theme has special behaviour for the plus operation over two operands. However, from a behavioural (and

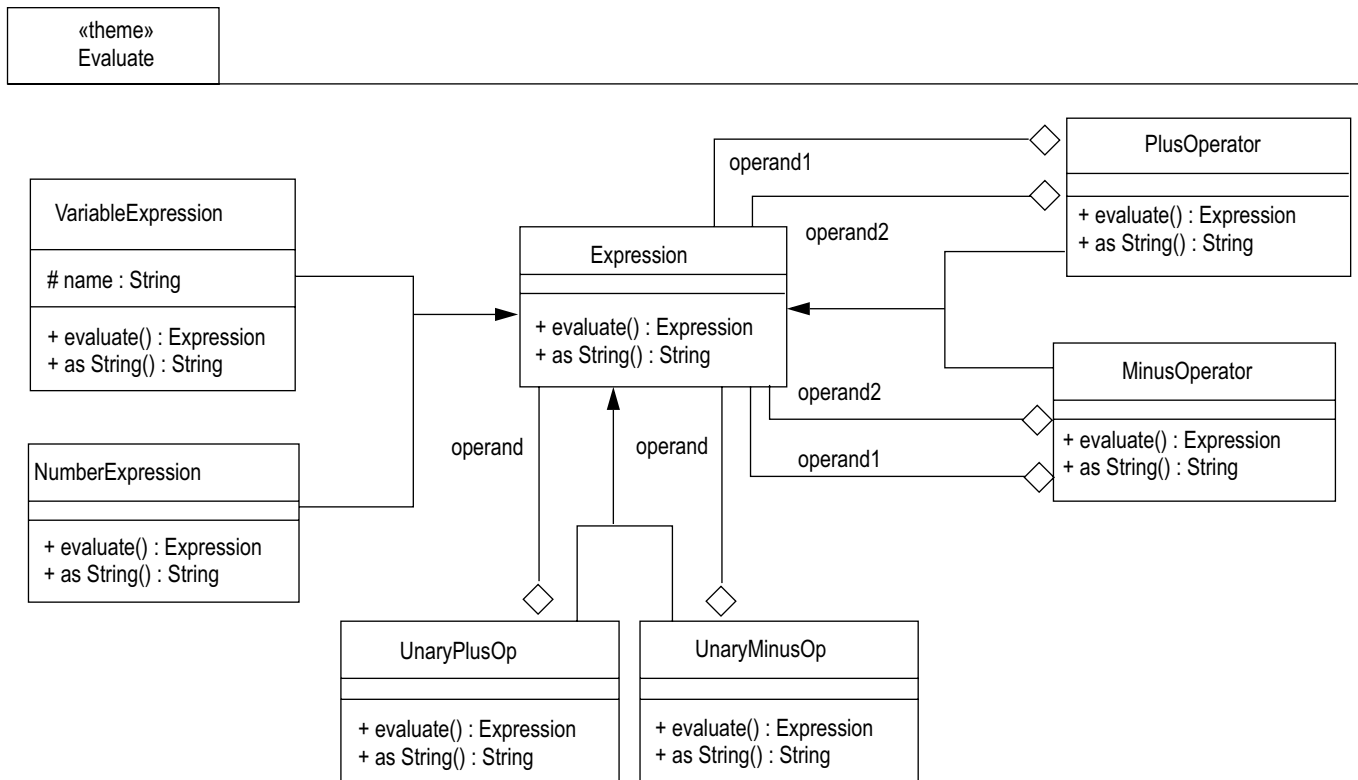


Figure 1-2: Evaluate Theme.

possibly also a structural) perspective, the design elements that are needed to design the grammar and the evaluation or checking operations are different. See Figure 1-1 and Figure 1-2 for an example of the structural design of the Grammar and Evaluation themes, respectively. The Grammar theme uses the standard Composite pattern [5] to model the container/component nature of expressions, and has basic accessor operations defined.

The Evaluation theme has a slightly different hierarchy defined, with all behaviour defined relevant only to evaluating expressions. For space reasons, we omit the behavioural specifications of these themes.

2.1.2 Themes that Crosscut

The logging theme represents one of the classic aspect-oriented software development crosscutting examples. Since all operations must be logged, all operations in all themes must be augmented with the logging behaviour. However, we want to be able to design the logging behaviour separately from any operations to be logged – in other words, to reason about those operations without referring to them explicitly.

This is where we see our first diversion from the standard UML. Theme/UML extends the UML’s notion of templates with *template parameters* and *pattern classes* to support independent reasoning about design elements to be crosscut. Similar to templates in the UML, a template parameter is a parameterised model element that can be referred to in the theme design, but that will be replaced by real model elements in a

composed design. There is no special notation for a pattern class – it refers to any class that has a template element. The representation of all template parameters for all

pattern classes is combined in a single box and placed on the package box for the overall theme. There may be multiple template parameters in different pattern classes defined for a single crosscutting theme.

The log theme writes text to a log file before and after the execution of every operation being logged. This logging behaviour has no particular interest in the semantics of the operation being logged – just that it is executing. An operation, therefore, is a template parameter to be replaced by real operations in the composed design. Any class that has an operation to be logged (and so, has operations that will replace the template parameter) is therefore considered to be a pattern class. Figure 1–3 illustrates the design.

The template parameter, called `_loggedOp()` is in the pattern class called `LoggedClass`. From the interaction diagram in the theme design, we see the behaviour specified for logging operations. Notice that there are other operations defined for a logged class such as `beforeInvoke()` and `afterInvoke()`. These are examples of elements particular to logging that are not template parameters, and so will not be replaced in the composed design. These elements are merged unchanged into any class that replaces the `LoggedClass` pattern in a composed design. Logging also introduces a structural property that will be added unchanged to any class that replaces the pattern – the relationship to the `LogFile` single-

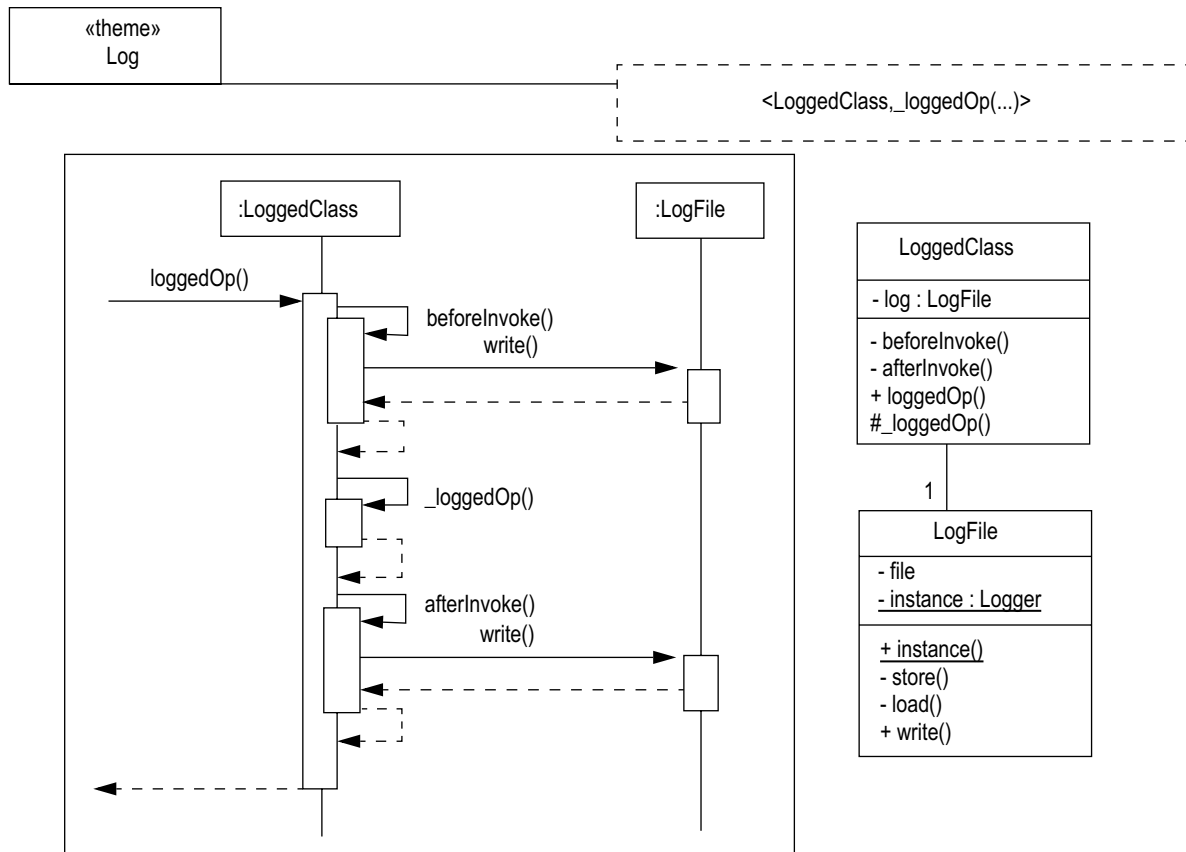


Figure 1-3: Logging Theme.

ton. Notice also how the designer represents the replacing operation with an “_” pre-pended to the template parameter name. `_loggedOp()` is used in the template box, with a generated operation realised by the interaction diagram named as `loggedOp()`. By doing this, when a replacing operation is to be logged, the required logging behaviour to be triggered before and after its execution is clearly defined within the interaction.

2.2 Specifying Relationships between Themes

Theme/UML has defined a new kind of relationship, called a *composition relationship*, that allows a designer identify the overlaps in the different themes to be composed, and specify how those overlaps should be integrated in a composed model.

2.2.1 Relating Themes that Share Domain Concepts

The composition relationship between themes whose overlapping is as a result of sharing domain concepts allows the designer designate which design elements within the themes should be considered to be the same core concept, how to resolve conflicts between them, and also how to integrate them. Multiple relationships may be included between two (or more) themes.

In the simplest case where shared concepts have been given the same name, all a designer needs to do is draw a composition relationship between the relevant themes with a `In the simplest`

case where shared concepts have been given the same name, all a designer needs to do is draw a composition relationship between the relevant themes with a `match[name]` tag. The elements will be merged in a composed design, with matching elements appearing once. Where the matching element is a container like a class, the elements in all matching containers will appear in the one composed container. The simplest case is appropriate for the expression example. Figure 1-4 illustrates how to specify a composition of the grammar, check (not illustrated in previous section) and evaluate themes.

Of course, the simplest case will probably not be applicable in every situation. Theme/UML also has a means to match elements with different names, exclude elements that may

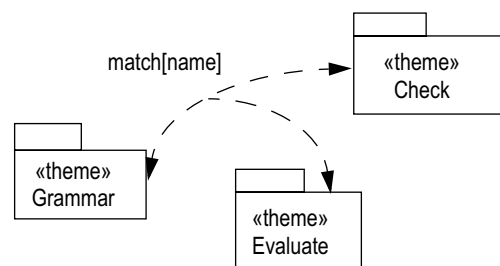


Figure 1-4: Composition Relationship – Shared Concepts

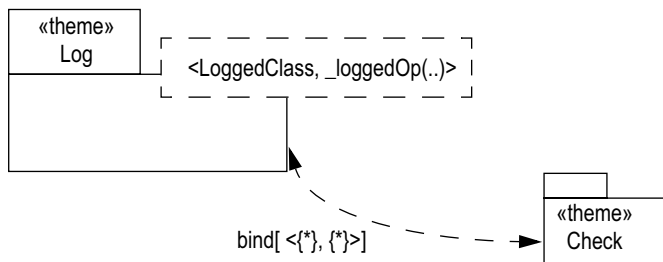


Figure 1-5: Composition Relationship – Crosscutting.

coincidentally have the same name but are not a representation of the same concept, and specify how to reconcile any conflicts in matching elements. We will also describe an integration policy based on overriding elements in one theme with elements from another theme. This is useful for extension themes that contain design changes to possibly obsolete elements in the existing design.

2.2.2 Relating Themes that Crosscut

As described for the log design, a theme that contains a design for crosscutting behaviour specifies templates as placeholders for real design elements. Relating themes where one of them has crosscutting behaviour entails listing the real design elements that will replace such templates. The Theme/UML composition relationship has a `bind[]` attachment with a syntax for listing all elements that will replace each of the pattern classes and template parameters. Figure 1-5 illustrates a straightforward example of this for relating the log theme with, as an example, the evaluate theme.

The `<{*},{*}>` parameters of the `bind` annotation to the composition relationship indicate that all classes in the base theme(s) being composed (in this case, just the evaluate theme), and all operations within those classes, should (separately) replace the pattern class and template operation, respectively. In other words, every operation in every class should be logged. The syntax of the `bind` attachment also caters for more complex and more refined specifications of the real elements that should replace the template parameters in a composition.

2.3 Composed Themes

At this point in the design process, the internals of individual themes are well understood, and composition relationships have been defined where overlaps between themes are also understood – both in terms of how they each may have elements that describe the same domain concept, and also relating to crosscutting behaviour. Theme/UML defines semantics for composed themes that allow the designer verify that the overlaps specified using composition relationships are actually what were intended.

The process of actually composing the designs is a verification activity to see if the composition relationships defined make sense. The composed design will, of course, display the non-modular characteristics that this whole approach is designed to avoid. When considering the composed design, the

designer should therefore verify that the behaviour is as expected, and not evaluate the design against software engineering qualities such as modularity or inheritance refinements. We recommend that the development team choose an aspect-oriented implementation language to implement the separated themes, and use the composition specification in the Theme design to guide how the separate code artefacts are composed.

3 Extensions to the UML Metamodel

The main extension to the UML metamodel in Theme/UML is the composition relationship, which is defined between composable elements. For the purpose of composition, we consider composable elements to be either composite or primitive. Primitives are defined as elements whose full specifications are composed with other primitives (e.g., attribute, operation). Some elements contain other elements, however, and therefore cannot be considered to be primitive. For example, a class contains attributes and operations, and those attributes and operations must be examined individually for overlapping and integration. Such elements are composite elements. Composites are defined as elements whose components are not considered part of the full specification of the composed and therefore are considered separately for composition. Composites may contain other composites (e.g., a package contains classes).

A composition relationship is defined between composable elements. It is a new kind of relationship for the UML, and is subclassed from the UML's Relationship metaclass (see Figure 2). Composition entails synthesising two or more input themes into an output theme. Identifying inputs to a composition must first involve identifying the input theme, and specifying a composition relationship between those themes – this composition relationship is considered to be the contextual composition relationship that defines the context for any additional composition relationships specified between elements inside the input themes.

A primitive composition relationship is a composition relationship between primitives, and a composite composition relationship is a composition relationship between composites. Composable elements explicitly related by a composition relationship are said to overlap, and are integrated based on the semantics of the particular integration strategy attached to the composition relationship. A more general approach to identifying corresponding elements is possible with the Match metaclass, which supports the specification of match criteria for components of composite elements related by a composition relationship. Integration as specified in Figure 2 is an abstract metaclass, where it is intended that it be specialised to cater for the particular integration specification required. The default semantics for integration strategies is that a composition process results in the composition of elements to new model elements, as defined by the 'composed' relationship from the Integration metaclass. See [2] for extensions to the metamodel for two kinds of integration strategies that have been defined within Theme/UML – merge and override.

Theme/UML also extends the UML's notion of templates to support crosscutting themes. Theme/UML template parameters

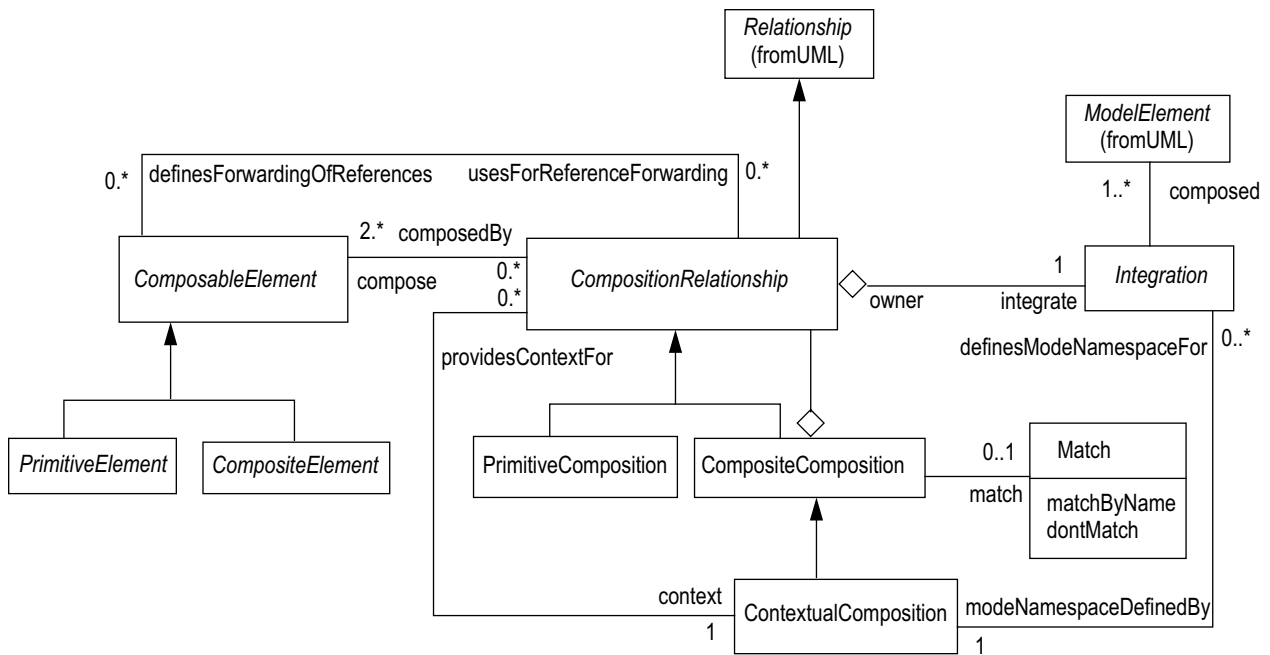


Figure 2: Composition Relationship.

differ from the UML templates model in two primary ways. First, templates within a crosscutting theme are centred on pattern classes (placeholder classes to be replaced by ‘real’ class elements) within a theme first, which may have additional template parameters defined. Second, binding actual classes and model elements from (an)other theme(s) is achieved with an extension to a composition relationship with merge integration defined. This composition relationship’s arguments define which classes replace the pattern classes, and which elements within the replacing classes replace a pattern class’s template parameters. See [ref socp] for more details.

4 Related Work

The UML itself contains a small number of mechanisms that could be used to separate different elements that support different requirements. For example, attributes and operations may be organised within classes using stereotypes to group them for particular needs. In addition, multiple models of the same kind (e.g. multiple object models or multiple class models) may be defined within the same package that could be used to provide a limited measure of separation based on requirements. This support is limited for overlapping concepts (concepts that support multiple requirements) because, using UML, design elements that support the same concept, but have different views that necessitate different specifications, must be specified separately. Since there is no means of synthesising a complete design of incomplete pieces in UML, such elements will remain separate throughout the design cycle. Multiple generalization is another mechanism that could be used to combine multiple different structural and behavioural properties, designed to support different requirements. However, there are some difficulties with using this technique in an

attempt to separate support for different requirements into different classes. First, separation based on multiple generalization is not possible when there are overlapping concepts that support multiple requirements. Another issue is the practicality of the approach based on the possibilities relating to an explosion of the class hierarchy for each new requirement added.

Conceptually, Theme/UML has evolved from the work on subject-oriented programming [6]. Different subjects (similar to themes) may be or programmed to support separate requirements, be they functional (and conceptually overlapping) or crosscutting requirements. Defining composition relationships in Theme/UML is analogous to defining composition rules in subject-oriented programming.

The goals of Theme/UML and those of the role modelling work from the OORam software engineering method [8] are similar. OORam shows how to apply role modelling by describing large systems through a number of distinct models. Derived models can be synthesised from base role models, as specified by synthesis relations. Synthesis relations can be specified both between models and between roles within the models, much like our composition relationships. The synthesis process is equivalent to the synthesis of subjects defined with a merge interaction specification. The subject-oriented design model distinguishes itself with its notion of override integration, and more particularly, with the potential provided by composition patterns to provide for more sophisticated, complex possibilities for combination patterns.

Separation of concerns with Catalysis [3] is based on UML, using *horizontal* and *vertical slices* to separate a package’s contents according to concerns. Composition of artefacts is based on a definition of the UML *import* relationship, called *join*. The designer is instructed to form a new design containing

the simple union of design elements, with re-naming in the event of unintended name clashes. This approach is similar to the meaning of a merge interaction specification with property matching by name. Catalysis encourages a design strategy in which an initial design is gradually modified to produce a completed one, which is a single, fully integrated design. The subject-oriented design model supports a design strategy in which pieces (subjects) are identified and designed separately, and may remain separate in the completed design, though related by composition relationships. This enhances the traceability of requirements that lead to various artefacts. For example, Catalysis describes the rules and decisions a designer should (might) follow to form the result of joining two packages, while the original packages in Theme/UML are retained. Instead, a way of specifying the rules and decisions as annotations on the composition relationship(s) relating them is defined. Reusable design components are supported in Catalysis with template frameworks, containing placeholders that may be imported, with appropriate substitutions, into model frameworks. This is similar to merging reusable design themes that have no overlapping design elements, possibly using crosscutting themes.

From the perspective of crosscutting requirements, Theme/UML also closely relates to the aspect-oriented programming model that separates crosscutting behaviour into separate ‘aspects’ [7]. There are some interesting design approaches that are rooted in the aspect-oriented programming paradigm. Approaches that consider aspects for UML as their main concern broadly fall into two categories: those that extend UML constructs and semantics to deal with aspects, or those that use standard UML in new ways. To our knowledge, Theme/UML is the most prominent approach in the former category. An approach that avoids extending standard UML is described in [4] that instead, relies on advanced features of UML statecharts. Concurrent statecharts describe different behaviours, some of which may be crosscutting. This paper shows how to use standard (though advanced) features of statecharts, including event broadcasting, to achieve implicit weaving. This has the benefit of allowing use of standard UML tools, but the additional complexity of using UML constructs in novel ways, previously unintended ways that may be confusing to the designers. Other approaches exist that make extensive use of stereotypes to characterise the various aspect-oriented constructs, relying on descriptions of the semantics from programming models to map to the design [9] [10].

5 Summary

This paper describes an approach to aspect-oriented design using an extension to the UML called Theme/UML. Using Theme/UML, the design of the system is modularised into ‘themes’. A theme is any feature, or concern, or requirement of interest that has to be catered for in the system. Differ-

ent theme designs may overlap in two main ways – they may share core concepts, or they may define crosscutting behaviour. Theme/UML’s extensions to the UML centre around supporting the designer in specifying how overlapping themes relate to each other, and in the composition capabilities that are defined. To provide this capability, Theme/UML has defined a new kind of relationship, called a composition relationship that allows the designer identify those parts of the theme designs that relate to each other and therefore should be composed. For themes that crosscut others, this means identifying when and where in those other themes the additional behaviour should occur. For other kinds of overlaps, this means identifying elements in the theme designs that correspond to each other and saying how they should be integrated. Theme/UML also provides semantics for model composition based on the composition relationship specification. This supports a level of verification that allows the designer to consider the overall system design, including the composition specification, to ensure that it makes sense. These extensions have been defined as extensions to the UML metamodel.

References

- [1] E. Baniassad, S. Clarke. Theme. An Approach for Aspect-Oriented Analysis and Design. To appear in proceedings of ICSE, 2004.
- [2] S. Clarke. Extending standard UML with model composition semantics. In Science of Computer Programming, 2002.
- [3] D. D’Souza, A.C. Wills. Objects, Components and Frameworks with UML, The Catalysis Approach. Addison-Wesley, 1998.
- [4] T. Elrad et al. Expressing Aspects Using UML Behavioural and Structural Diagrams in Aspect-Oriented Software Development. Addison-Wesley (to appear.)
- [5] E. Gamma et al. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
- [6] W. Harrison, H. Ossher. Subject-oriented programming (a critique of pure objects). In proceedings of OOPSLA, 1993.
- [7] G. Kiczales et al. Aspect-Oriented Programming. In proceedings of ECOOP 1997.
- [8] T. Reenskaug, et al. Working with objects, The OORam Software Engineering Method. Manning Publications Co., 1995.
- [9] D. Stein et al. A UML-based Aspect-oriented Design Notation for Aspect. In proceedings of AOSD, 2002.
- [10] J. Suzuki, Y. Yamamoto. Extending UML with aspects: aspect support in the design phase. In proceedings of the Aspect-Oriented Programming Workshop at ECOOP, 1999.
- [11] P. Tarr et al. N-degrees of separation: Multi-dimensional separation of concerns. In proceedings of ICSE, 1999.