Concern Driven Software Development

Omar Alam

School of Computer Science, McGill University, Montreal, Canada
Omar.Alam@mail.mcgill.ca

Abstract Model Driven Engineering (MDE) has achieved success in many areas of software development. However, despite its success, MDE faces some key challenges in practice. One of these challenges is *model reuse*. Modellers usually build models from scratch instead of reusing existing models. This makes modelling a more difficult task than coding, since modern programming languages provide libraries facilitating code reuse. This paper presents concern-driven software development – a new development process that focuses on concerns as primary artifacts. A concern groups related models serving the same purpose, and provides interfaces to facilitate reuse. We discuss the vision of concern driven development and discuss how concerns can be modelled at the design phase. We also discuss how we plan to address the challenge of model reuse in concern-driven development, by using concern libraries and tool support.

1 Introduction

This article presents concern-driven development (CDD), a development process that allows large-scale model reuse. In CDD, we focus on key challenges that MDE faces. One main challenge that we address is reusability. Models of evolving complex systems grow in size to an extent that individual model views are hard to understand and maintain. In addition, creating complex models from scratch is time consuming and the lack of model reuse leads to the creation of many redundant models. CDD aims at integrating Aspect-Orientation Modelling (AOM) [18] with MDE to overcome this challenge. AOM is a new paradigm that has been successfully used to separate crosscutting concerns (aspects) within software models, allowing the developer to reason about each concern individually. The model weaver then performs an elaborate model transformation which merges and composes related model elements from each concern to produce a final model (in which the concerns are linked). Previous research provides evidence that AOM can be used to build models of complex systems by composing together many interdependent simple concern models.

A *concern* addresses a domain of interest for the modeller. It encapsulates a set of models at potentially different phases of software development that pertain to some points of interest of a developer or stakeholder. A concern has a root phase, i.e. it starts to be relevant at a certain point during software development. For example, some concerns appear at the requirements phase, e.g. security, since they are of relevance to external stakeholders. Other concerns, for instance database integration, appear during the architecture or design phase. For the root phase and all subsequent phases, models are built to express the properties of relevance for that phase using the most appropriate formalism.

Models should be as general as possible, and provide a composition interface that allows a developer to specify how to customize the models to a specific context, as well as how to compose them with models of other concerns at the same level of abstraction. Concern models should also include all relevant variations/choices that are available to developers, if any, together with guidance on how to choose among those variations when moving from one phase to the next. Finally, a concern should also define the model transformations that link the models established at different levels of abstraction.

The rest of the paper is structured as follows. Section 2 presents concerns – the primary artefact of CDD and discusses the concern interfaces that facilitate reuse. Section 3 provides a methodology to realize concern-oriented reuse at the design phase. In particular, we discuss case studies, tool support and reusable concern library. Finally, in section 4, we draw some conclusions and provide a future outlook.

2 Concerns as a Unit of Reuse

Software reuse is a powerful concept that originated in the sixties, and is defined as the process of creating new software using existing software artifacts. To make software reuse applicable, reusing an artifact should be easier than constructing it from scratch. This entails that the reusable artifacts are easy to understand, find, and apply [12]. There are characteristics of software artifacts that facilitate reuse, e.g., grouping, encapsulation, information hiding, and well-defined interfaces.

2.1 Units of Reuse

Until now, we focused primarily on modelling concerns at the design phase, but we are planning to extend our study to other phases, particularly the requirement phase. The most popular units of reuse for software designs include classes, components, frameworks, design patterns, and software product lines (SPL). We extended a particular unit of reuse, namely, Reusable Aspect Models (RAM) [11], to illustrate concern-oriented software design. RAM is an aspect-oriented multi-view modelling approach for software design modelling. A RAM model consists of a UML package specifying the structure and the behaviour of a software design using class, sequence, and state diagrams.

Reusability is a key element in RAM. Each model has a well-defined *model interface* [3] (explained in more detail in the next subsection), which specifies how the design can be (re)used within other models. Having an explicit model interface makes it possible to apply proper information hiding principles [15] by concealing internal design details from the rest of the application. Thanks to aspect-oriented techniques, this is possible even if the encapsulated design details crosscut the rest of the application design. RAM also offers the modeller the possibility to create *model hierarchies*, which allows one RAM model to reuse the structure and behaviour of another RAM model within its design. Model composition techniques are used to flatten aspect hierarchies to create the final software design model.

2.2 Concern Interfaces

Units of reuse such as the ones discussed above typically either explicitly or implicitly define *interfaces* that detail *how* the unit is supposed to be reused. We classify these interfaces here into three kinds: *usage*, *customization, and variation interfaces*. We discuss in details the specific concern interfaces related to the extended version of RAM in [5].

Usage Interface: *The usage interface* for units that are used in software design *specifies the design structure and behaviour that the unit provides* to the rest of the application. In other words, the usage interface presents an abstraction of the functionality encapsulated within the unit to the developer. It describes *how* the application can trigger the functionality provided by the unit.

For instance, for classes the usage interface is the set of all *public* class properties, i.e., the attributes and the operations that are visible and accessible from the outside. For components, the usage interface is the set of services that the component provides (i.e., the *provided interface*). For frameworks, design patterns, and SPLs, the usage interface is comprised of the usage interfaces of all the classes that the framework/pattern/SPL offers.

Customization Interface: Typically, a unit of reuse has been purposely created to be as general as possible so that it can be applied to many different contexts. As a result it is often necessary to tailor the general design to a specific application context. *The customization interface* of a reusable software design unit *specifies how to adapt the reusable unit to* the *specific needs* of the application under development.

For example, the customization interface of generic or template classes allows a developer to customize the class by instantiating it with application-specific types. For components, the customization interface is comprised of the set of services that the component expects from the rest of the application to function properly (i.e., the *required interface*). The developer can use this information at configuration time to plug in the appropriate application-specific services. The customization interface for frameworks and design patterns is often comprised of interfaces/abstract classes that the developer has to implement/subclass to adapt the framework to perform application-specific behaviour.

Variation Interface: The variation interface captures the variations offered by a concern as well as the impact of a selection of variations on high-level goals such as non-functional application properties. Variations are best expressed at a high level of abstraction, where details of the variation can be ignored and the focus can be on the relationships among offered variations. Feature modelling [10] and goal modelling of URN standard [9] have addressed these modelling requirements, and hence we make use of feature and goal modelling techniques in the specification of the variation interface. Software Product Lines use feature models to express variations [16]. However, in SPL, the application of feature models is limited to express variation of different products.

3 Methodology

To reach our goal, we will work on case studies, concern library and tool support.

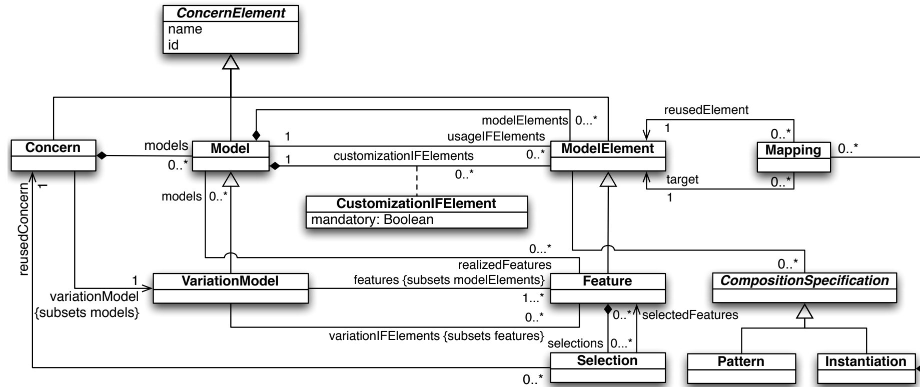

Figure 1. Concern Metamodel

3.1 Case studies

The bCMS [6] was proposed as a common case study for the AOM community and the models of participating approaches are available in a public repository [1]. We are planning to model the entire bCMS using CDD, including non-functional requirements, design variations, and communication protocol. Although the focus of our work is at design phase, we model the requirement-level concerns using AoURN [13]. AoURN is a notation that uses the use case maps, allowing the modeller to capture the interaction scenarios that a system supports as well as the impact these scenarios have on high level goals. At the design level we will use the extended version of RAM that supports CDD as discussed earlier.

We are also planning to apply our concern-driven approach to an industrial case study – slot machines [7]. A slot machine is used in the casino gambling industry and its system contains significant number of crosscutting concerns. Both bCMS and slot machines depends on other concerns related to networking, security, workflow middleware [4] and data structures, making them good candidates to study concern dependencies and interactions.

3.2 Tool support

Figure 1 shows a metamodel that a potential modelling approach should extend to support concern-oriented reuse. The *Concern* is added as a new root model element that groups together one or more aspects. The concern exposes a *VariationInterface*, which defines a set of features *(Feature)*. Concern groups *Models* which may include usage interface (*usageIFElements*) or customization interface elements (*customizationIFElements*). A concern can reuse other concerns through instantiation (*Instantiation*) and establishing mappings (*Mapping*) to model elements of the instantiated concern. We extended the RAM metamodel to support the metamodel depicted in Figure 1 [5]. RAM provides a multitouch-enabled tool for agile software design modelling aimed at developing scalable and reusable software design models [2]. The tool gives the designer access to a library of reusable design models encoding essential recurring design concerns. We are currently extending the RAM tool to support concern-oriented design models.

3.3 Library of Design Concerns

One of the main success factors for the programming language Java is the existence of an extensive standard library that contains commonly used code, such as searching algorithms, graphical user interfaces, and data structures. Bran Selic, one of the founders of MDE, has recently said [17]: "People complain about UML being too big. I say Java is even bigger. But still people don't complain about Java, because it has an extensive library." A similar observation was made by Jon Whittle, who found in his large-scale quantitive study that model reuse is not a common practice among the MDE practitioners, and companies that abandoned MDE found that without model reuse, modelling is too time consuming [19].

We are building a library of commonly reusable concerns in our tool. Currently, we have modelled 9 design patterns presented in [8], workflow middleware, and models related to security and networking middlewares. We continue to add more reusable models to the library.

4 Summary and Outlook

This article presents concern-driven development (CDD) that allows modellers to easily reuse existing models. Concern is a broad unit of reuse that groups related models and provides the modeller with three types of interfaces that facilitate reuse – usage, customization, and variation interfaces. We discussed how we applied CDD at the design phase by extending RAM to support the concern metamodel. Our approach will also provide a library of commonly reusable concerns and a tool support.

In future work, we plan to more tightly couple the AoURN/SPL tool jUCMNav [14] and the RAM tool TouchRAM [2], so that feature configurations selected in jUCMNav are automatically communicated to the RAM weaver. Furthermore, we are investigating how to automate the generation of the combined variation interface of two concerns (e.g., when an aspect of one concern reuses features of another concern). We are also working on larger case studies involving transaction and security concerns, and plan to undertake controlled experiments with designers regarding the usability and cost-effectiveness of concern-oriented software design.

If adopted on a large scale, we believe that concern-orientation has the potential to revolutionize software design reuse. It enables the creation of standard design concern libraries. Vendors can sell design concerns that target specific domains. Developers can become specialists responsible for the maintenance and evolution of specific design concern libraries. Ultimately, libraries, reuse, and specialization would provide a clear structure to software development, and as a result align the practice of software engineering closer to what is done in other engineering disciplines.

References

1. ReMoDD Model Repository. URL: http://www.cs.colostate.edu/remodd/ (2011)

2. Al Abed, W., Bonnet, V., Schöttle, M., Alam, O., Kienzle, J.: TouchRAM: A multitouch-enabled tool for aspect-oriented software design. In: 5th International Conference on Software Language Engineering - SLE 2012. pp. 275 – 285. No. 7745 in LNCS, Springer (October 2012)
3. Al Abed, W., Kienzle, J.: Information Hiding and Aspect-Oriented Modeling. In: 14th Aspect-Oriented Modeling Workshop, Denver, CO, USA, Oct. 4th, 2009. pp. 1–6 (October 2009)
4. Alam, O., Kienzle, J.: Designing with inheritance and composition. In: 3rd International Workshop on Variability and Composition. pp. 19–24. ACM (2012)
5. Alam, O., Kienzle, J.: Concern-oriented software design. In: Proceedings of the ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems. Springer (to be published)
6. Capozucca, A., Cheng, B.H., Georg, G., Guelfi, N., Istoan, P., Mussbacher, G.: Requirements Definition Document for a Software Product Line of Car Crash Management Systems. URL: http://cserg0.site.uottawa.ca/cma2011 (2011)
7. Fabry, J., Zambrano, A., Gordillo, S.E.: Expressing aspectual interactions in design: Experiences in the slot machine domain. In: MoDELS. pp. 93–107. Lecture Notes in Computer Science, Springer (2011)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison Wesley, Reading, MA, USA (1995)
9. International Telecommunication Union (ITU-T): Recommendation Z.151 (11/08): User Requirements Notation (URN) - Language Definition (2008)
10. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (November 1990)
11. Kienzle, J., Al Abed, W., Klein, J.: Aspect-Oriented Multi-View Modeling. In: AOSD 2009. pp. 87 – 98. ACM Press (March 2009)
12. Krueger: Software reuse. CSURV: Computing Surveys 24 (1992)
13. Mussbacher, G., Amyot, D.: Extending the User Requirements Notation with Aspect-Oriented Concepts. In: 4th International SDL Conference on Design for Motes and Mobiles – SDL'09. pp. 115–132. Springer-Verlag (2009), http://dl.acm.org/citation.cfm?id=1812885.1812896
14. of Ottawa, U.: jUCMNav website: http://softwareengineering.ca/jucmnav (2013)
15. Parnas., D.L.: A technique for software module specification with examples. Communications of the ACM 15(5), 330–336 (1972)
16. Pohl, K., Metzger, A.: Variability management in software product line engineering. In: Proceedings of the 28th international conference on Software engineering (ICSE '06). pp. 1049–1050. ACM (2006)
17. Selic, B.: "the theory and practice of modeling language design". Tutorial given at MODELS 2012, Innbruck, Austria, Oct. 1st 2012
18. Solberg, A., Simmonds, D.M., Reddy, R., Ghosh, S., France, R.B.: Using aspect oriented techniques to support separation of concerns in model driven development. In: COMPSAC (1). pp. 121–126. IEEE Computer Society (2005), http://dblp.uni-trier.de/db/conf/compsac/compsac2005-1.html#SolbergSRGF05
19. Whittle, J.: "the truth about model-driven development in industry - and why researchers should care". http://www.slideshare.net/jonathw/whittle-modeling-wizards-2012/ (2012)