Concern-Driven Software Development

Omar Alam Jörg Kienzle
School of Computer Science, McGill University,
Montreal, QC H3A 0E9, Canada
Omar.Alam@mail.mcgill.ca, Joerg.Kienzle@mcgill.ca

Gunter Mussbacher
Department of Electrical and Computer Engineering,
McGill University, Montreal, QC H3A 0E9, Canada
Gunter.Mussbacher@mcgill.ca

Technical Report, School of Computer Science, McGill University, January 2015, CS-TR-2015.1

Categories and Subject Descriptors D.2.10 [*Software Engineering*]: Design; I.6.5 [*Simulation and Modeling*]: Model Development

Keywords software concern line, concern-driven development, reuse, variation interface, customization interface, usage interface.

Abstract

This paper describes the vision of Concern-Driven Development (CDD), a novel software development paradigm that extends model-driven engineering with best practices from reuse, advanced modularization techniques, goal modelling, and software product line research. CDD advocates the use of a three-part interface to describe units of reuse, i.e., concerns. The variation interface describes required design decisions and their impact on high level system qualities, both explicitly expressed using feature models and goal models. The customization interface allows the chosen variation to be adapted to a specific reuse context, while the usage interface defines how a customized concern may eventually be used. When a concern is reused, the modeller first uses the variation interface to select the feature configuration that has the desired impact on stakeholder goals and system qualities, then adapts the concern to the context of the application under development with the help of the customization interface, and finally accesses the concern's functionality through its usage interface. We argue that, if CDD is successfully adopted on a large scale, it will transform the software engineering discipline by enabling software engineers to specialize to a greater degree and hence align the practice of software engineering closer to what is done in other engineering disciplines.

1. Introduction

Engineering is defined as "creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works" that "respect an intended function, economics of operation or safety to life and property" [1]. Some engineering disciplines, e.g., civil engineering, have matured over hundreds of years to a point where standard components and standard processes exist that guide the engineer to weigh different design choices on their merits and choose the solution that best matches the requirements.

Software engineering as a discipline aims at ensuring that software is built systematically, rigorously, measurably, on time, on budget, and within specification. Complex modern software-based systems often require many stakeholder groups (e.g., scientists, software developers, and end-users with specialized domain knowledge) to work in a coordinated manner on different aspects of the system. Each aspect – called *concern* in this paper – is a domain with its own specialized knowledge space. A major challenge that stakeholders face is bridging the wide gap between the domain-specific concepts they use to express their problems and the programming languages and technologies used to implement solutions. Software reuse, although successful at the programming language in form of libraries, components, frameworks, and services, is still very limited at higher levels of abstraction. As a result, software engineers must manually bridge the gap between problem and solution for each project, which introduces costly accidental complexities [2]. Software modelling techniques, for example the Unified Modeling Language (UML) [3], are of limited use, because the crosscutting nature of most concerns is an obstacle for classical (object-oriented) modularization techniques to encapsulate a recurring modelling concern in a reusable way.

Some engineering disciplines have specialized greatly over the years, e.g., civil engineers usually practice as construction engineers, transportation engineers, hydraulic engineers, environmental engineers, and so on. In contrast, software engineers are still expected to master all concerns of a software-based system, which is becoming more and more difficult considering the complexity of modern software-based systems.

This paper outlines concern-driven development as a visionary new software development paradigm inspired by the ideas of [4] that builds on the disciplines of model-driven engineering, software product lines [5], goal modelling, and advanced modularization techniques offered by aspect-orientation to define flexible software modules that enable broad-scale model-based software reuse. In contrast to Software Product Lines, where reuse is planned for and takes place within the boundaries of the domain of the product line, CDD aims at enabling reuse across concerns (i.e., *Software Concern Lines*). Concerns are units of reuse that are typically developed in isolation. Reuse between concerns is made possible by means of well-defined interfaces.

In the remainder of the paper, Section 2 presents the main concepts of CDD. Section 3 discusses the flexible form of modularity that concerns provide, while the last section draws the conclusions.

2. Concern-Driven Software Development

MDE [6] is a unified conceptual framework in which software development is seen as a process of *model production*, *refinement,* and *integration*. To reduce the accidental complexity and the effort needed to move from a problem domain to a software-based solution, MDE advocates the use of different modelling formalisms, i.e., modelling languages, to represent and analyze the system from multiple points of view. For each level of abstraction, the modeller uses the best formalism that concisely expresses the properties of the system that are important to that level, and that the concepts used in the language are close to the problem domain at hand. During development, high-level specification models are refined or

combined with other models to include more solution details, such as the chosen architecture, data structures, algorithms, and finally even platform and execution environment-specific properties. The manipulation of models is achieved by means of model transformations. Model refinement and integration continues until a model (or code) is produced that can be executed.

However, MDE on its own is not a silver bullet. To reduce the complexity of reasoning and analyzing the problem and constructing a software-based solution, traditional software engineering principles such as decomposition, interfaces, information hiding, levels of abstraction, and reuse are key. Unfortunately, the crosscutting nature of most concerns is an obstacle for classical modularization techniques that apply the aforementioned principles in practice. To be effective, a *flexible notion of modularity* is required, that allows to separate and package concerns in a reusable way, and allows advanced composition mechanisms to introspect modules and compose them to build a usable (i.e., analyzable, simulatable, and/or executable) representation of the solution.

Concern-driven development (CDD) is a further development of MDE that proposes a novel software module – the *concern* – and uses it as the main unit of modularization, abstraction, construction, and reasoning.

2.1 Concern

A concern is any domain of interest to a software engineer. It can be but does not have to be a crosscutting concern as advocated by aspect-oriented software development. A concern is a unit of modularization that encapsulates a *set of models* describing all properties of that concern required to sufficiently understand and use the concern. Typically, the models within a concern span multiple phases of software development and levels of abstraction. Each concern has a root phase, where the concern manifests itself for the first time. Some concerns appear in early phases of software development, e.g., broadly scoped system properties with functional, non-functional, or even intentional characteristics. Some concerns appear in later phases of software development, e.g., solution-specific concerns such as specific communication protocols, concrete authentication algorithms, and design patterns.

In CDD, models are built for the root phase and all follow-up phases using the most appropriate modelling formalisms to express the properties of the concern that are relevant during each phase. Consequently, a concern is described by many modelling notations, which may be object-oriented in nature (e.g., based on UML), but typically also need to offer other language mechanisms (e.g., aspect-oriented features) in order to handle the crosscutting nature of certain properties of the concern. Within a concern, *model transformations* link models across different levels of abstraction. Finally, a concern also encapsulates all relevant variations that are available to software engineers at a given phase, together with guidance on how to choose among those variations by specifying the impact of each choice on stakeholder goals and system qualities.

2.2 Concern Interface

The key concept of CDD promoting modularity is the *three part interface* [7] that every concern must provide:

- The *Variation Interface* describes the available variations of the concern and the impact of different variants on high-level stakeholder goals, qualities, and non-functional requirements. The variations are typically represented with a *feature model* [8] that specifies the individual features that a concern offers, as well as their dependencies such as optional, alternative, requires, and excludes. The impact of choosing a feature can be specified with goal models (e.g., with GRL, which is part of the User Requirements Notation (URN) standard [9], or the NFR framework [10]). For example, a security concern may offer various means of authentication, from *password-based* to *biometrics-based solutions*, each with differing impacts on the *level of security* as well as *cost* and *end-user convenience*. These qualities have to be weighed, when determining which authentication variant is most appropriate in the current application context.
- The *Customization Interface* describes how a chosen variant can be adapted to the needs of a specific application. Each variant of a concern is described as generally as possible to increase reusability. Therefore, some elements in the concern are only *partially* specified and need to be complemented with concrete modelling elements of the application that intends to reuse the concern. The customization interface is hence used when a specific variant of a reusable concern is *composed* with the application. For example, a security concern may define a generic *User* as a partial class that needs to be merged with the concrete application classes that describe the actual users of the system, e.g., *Administrator* or *Employee*.
- The *Usage Interface* describes how the application can finally access the structure and behaviour provided by the concern. For example, the usage interface of the design model of a concern is typically comprised of all *public* classes and methods made available by the concern. For the security concern this might include an *authentication* operation that an administrator can invoke in order to gain access to restricted behaviour.

2.3 Concern Reuse Process

Building a concern is a non-trivial, time consuming task, typically done by or in consultation with a domain expert. Deep understanding of the nature of the concern is required to be able to identify its different features (and capture them in a feature model), to model the common properties and differences of all features of a concern at all relevant levels of abstraction (by building requirements models and design models that (i) realize the features of the concern using the most appropriate modelling notations and (ii) are eventually refined into executable specifications), and to express the impact of the different variants on high level stakeholder goals and qualities (using goal models).

On the other hand, reusing an existing concern is extremely simple, and essentially involves 3 steps:

1. The concern user must first select the feature(s) with the best impact on relevant stakeholder goals and system qualities from the variation interface of the concern based on provided impact analysis. Based on this configuration, the modelling tool then merges the models that realize the selected features to yield new models of the concern corresponding to the desired configuration. Depending on the root phase of the concern, the merging may involve requirement models and/or design models. For composition at the requirements level with goal and workflow/scenario models, the interested reader is referred to [11] for more details. For details on how this composition is done for design concern models with structural and behavioural descriptions based on class, sequence, and state diagrams, see [7].
2. Next, the concern user has to adapt the generated detailed models to the application context by mapping customization interface elements to application-specific model elements. Again, depending on the root phase of the concern, this step might require customizing requirement models and/or design models.
3. Finally, a software engineer can use the functionality provided by the selected concern features which are exposed in the usage interface within his own application models. In requirements models, this may mean including workflow segments exposed in the concern's usage interface in the application's workflow

models. In design models expressed using sequence diagrams, for instance, using a concern may involve instantiating a class exposed in the concern's usage interface and/or calling one of its public operations.

3. Concerns as Units of Modularization

The goal of the first part of this section is to reflect on the properties of a single concern and the powerful modularization support it provides. The second part discusses concern hierarchies and the modular, abstraction-level preserving reasoning they enable.

3.1 Encapsulation and Adaptation of a Concern

Information hiding [12] is the activity of consciously deciding what parts of a software module should be exposed to the outside, i.e., the "rest" of the application, and what parts should be hidden from external use. To allow a developer to state what is internal and what is external to a module, modelling or programming languages typically provide constructs to define a module's *interface*. The problem with classical interfaces of reusable software modules is that they usually tend to encapsulate the state and behaviour so strongly that it is often impossible to adapt the module to the needs of a developer.

Information hiding, if employed well, gives the developer the ability to hide the internal workings of a software module behind a well defined interface. As a result, anyone who wishes to interact with a module only needs to know what the module does and is not dependent on the details of how it does it. This is a critical property, because it allows a developer to modify the internal state and behaviour of the module, e.g., improve an algorithm or use a different data structure, without affecting how the outside world interacts with the module. A developer can even go to the extent of replacing the module with a completely different one that supports the same interface.

For reusable software modules, though, complete encapsulation is not desired. Reusable modules are designed in a very general way, which typically means that they need to work with state or execute behaviour that is defined by the application context (i.e., the application that uses them). Concerns as advocated by CDD go even further, because they aim at modularizing different available variations or solutions of a problem. While each variation could be used, each one has different properties and impacts. Choosing the variant that is most adequate can only be done once the high-level stakeholder goals and non-functional requirements of the application context are known.

This is why concerns offer three interfaces. Combined together they provide the flexible modularization required to encapsulate domain knowledge, support information hiding and allow fine-grained adaptation to application-specific contexts.

The variation interface presents the available variants that a concern encapsulates to the user in a concise way. The user can make an informed choice by consulting the impacts of each configuration on non-functional requirements, stakeholder goals, and qualities. When a variant is selected, the concern performs a complex internal adaptation on the encapsulated implementation that typically involves structural and behavioural weaving. However, this adaptation is completely hidden from the user.

The customization interface (of the selected variant) then allows the user to compose application-specific structure and behaviour with the reusable concern's internal structure and behaviour. This specification takes the form of a simple mapping that establishes relationships between the application context and the entities exposed in the customization interface. Once this is done, structural and behavioural weaving is used again to adapt the encapsulated implementation in an automated way to the application context.

Finally, the resulting usage interface works just like a classic interface. It exposes the minimal structural and behavioural knowledge required by the application to trigger the functionality offered by the concern when needed, and at the same time encapsulates the specific implementation/solution details.

3.2 Concern Hierarchies and Information Hiding

As already mentioned in the introduction, complex applications consist of many intertwined, interacting concerns, and CDD advocates to develop an application by reusing as many already existing concerns as possible. The same principle applies to the development of a concern itself. As explained at the beginning of Section 2, a concern encapsulates *sets of models* that describe relevant properties at all levels of abstraction required to sufficiently understand the concern. Typically, a requirements concern, e.g., security, needs to comprise not only models that specify different ways of achieving security (authentication, role-based access control, encryption, etc.), but also different ways of realizing them (password-based authentication vs. biometrics, etc.). Even for a given realization, there are different possible implementation architectures (centralized password server vs. local, distributed databases, etc.). It comes with no surprise that low-level design solutions, such as various design patterns, transaction controls, or resource allocation are quite general solutions that can be reused in many contexts.

To fully reap the benefits of reuse, it is therefore important to allow the creation of *concern hierarchies*. To increase scalability and avoid duplication of effort, a high-level concern (or to be more precise, a feature of a high-level concern) should be able to reuse the functionality (structure / behaviour / properties) of a lower-level concern when appropriate. Doing so creates a concern hierarchy where a concern at a higher level of abstraction reuses other lower-level concerns. Similarly, a more domain-specific or solution-specific concern can reuse other more general concerns. For example, an authentication concern that encapsulates a variety of authentication means and protocols could be used within the context of a concern that provides authentication services for distributed systems connected with a local area network.

Concern hierarchies allow the developer to modularize the application into different layers of abstraction. But these layers again have to be flexible. In order to successfully reduce complexity, the layers should allow for separate reasoning, and hide the complexity of the lower levels from the upper levels. On the other hand, the layers can not be completely opaque, since the structure and behaviour of most lower-level concerns crosscuts the structure and behaviour of the upper levels (and the application). At the very least, the qualities of the upper level depends heavily on the qualities of the reused concerns at the lower levels.

Concern-orientation addresses this problem by allowing the concern designer to precisely specify how the interface of the concern being built is affected by the interfaces of the lower-level concerns that are reused.

3.2.1 Composing the Usage Interface

For the usage interface, standard information hiding principles are applied by default. In other words, the accessible structure and behaviour of the lower level concern are not included in the usage interface of the concern being built, unless explicitly reexposed by the developer. In those rare situations, which occur when the lower-level concern directly provides functionality that the concern under construction wants to offer, it is nevertheless often necessary to rename the reexposed functionality to reflect the change in level of abstraction. For example, a concern encapsulating many variants that implement the *Observer* design pattern might internally reuse the *Association* concern to associate a subject with its observers. In this case, the `getAssociatedObjects` functionality of the *Asso-*

ciation concern might be reexposed as a `getObservers` functionality in the usage interface of *Observer*.

3.2.2 Composing the Customization Interface

For the customization interface, the default rule is that the customization interface of the concern being built is a union of the new customization elements introduced by the concern and the customization elements of the lower-level concerns that have not been customized, i.e., that were not mapped to specific elements in the current concern. This makes it possible to grow or shrink the customization interface within concern hierarchies, depending on the intent of the designer. A "more specific" concern, for instance, would abstain from introducing new customization elements, and map some of the lower-level customization elements to specific elements of the concern under construction.

3.2.3 Composing the Variation Interface

The rules for constructing the variation interface of the concern being built from the variation interfaces of the concerns being reused are the most interesting. The considerations that have to be taken into account are multiple:

1. Choosing the best variant of a concern is only possible once all non-functional requirements and desired qualities are known. This might not be the case within a concern hierarchy, and only known when the complete application is being built.
2. A concern encapsulates all possible variants that can be useful in any context. When used in a specific context, some of these variants might not be applicable.
3. The qualities of the concern being built are affected by the qualities of the concerns being reused.

To address 1 and 2, the designer of the concern under construction should be able to *explicitly reexpose* all features of the lower-level concern that provide the required functionality (and indirectly exclude those features that do not have the desired properties) in its variation interface to defer the decision of which specific variant to use to the next level. To address 1, 2, and 3, specific algorithms to compose impact models of the high-level concern with those of the lower-level concern are needed.

Such composition algorithms must differentiate two distinct knowledge spheres. The first pertains to which feature configuration of a concern offers the best possible solution for a specific stakeholder goal or system quality. This can only be answered by concern specialists, because they are the experts of the concern's domain, and must hence be encoded in the concern's impact model. The second relates to the impact of a reused concern on the current concern that is reusing this concern. This can only be answered by the modeler who is tasked to build the current concern and must hence be expressed in the current concern's impact model. When goal models are used to describe impact models, then the best possible solution is identified by the stakeholder goal or system quality being evaluated to the maximum evaluation value. Based on this implicit understanding, the modeler of the concern under construction can then combine and describe the impacts of all reused concern on itself in a coordinated fashion. Note that, in the future, goal models could be replaced by more sophisticated models that allow for more accurate predictions of the composed concern's behaviour (e.g., performance models based on queuing theory could be used instead of goal models).

4. Conclusion

This paper presents the vision of concern-driven software development, a novel software development paradigm that extends model-driven engineering with best practices from reuse, aspect-orientation, and software product line research. The paper presents the flexible modularity that concerns offer. The 3-part interface of a concern reduces complexity by hiding unnecessary internal concern details from the concern user while still exposing the encapsulated design variants of the concern together with their impacts on non-functional system properties and allowing customization to specific application contexts.

The overhead of developing concerns is significant. The vision, however, is that if CDD is successfully adopted on a large scale, it will transform the software engineering discipline as a whole. CDD would enable software engineers to specialize, i.e., to become concern specialists. In companies selling concern libraries, security concern specialists, e.g., would solely concentrate on maintaining and evolving the models within the security concern, i.e., adding new security requirements, solutions, techniques and platforms as they become relevant. Within a company developing applications, a security concern expert would focus on composing the security concern with the other application concerns. Ultimately, concern libraries, concern reuse, and concern 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] Engineer's Council for Professional Development, 1947.

[2] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *Future of Software Engineering*, FOSE '07, pp. 37–54, IEEE, 2007.

[3] Object Management Group, *Unified Modeling Language: Superstructure (v2.4.1)*, December 2011.

[4] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., "N degrees of separation: Multi-dimensional separation of concerns.," in *ICSE'1999*, pp. 107 – 119, IEEE CS, May 1999.

[5] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[6] Douglas C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, pp. 41–47, 2006.

[7] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-oriented software design," in *International Conference on Model-Driven Engineering Languages and Systems - MODELS 2013*, vol. 8107 of *LNCS*, pp. 604–621, Springer, 2013.

[8] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, CMU, 1990.

[9] International Telecommunication Union (ITU-T), "Recommendation Z.151 (10/12): User Requirements Notation (URN) - Language Definition," approved October 2012.

[10] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*. Springer, 2000.

[11] G. Mussbacher, D. Amyot, and J. Whittle, "Composing goal and scenario models with the aspect-oriented user requirements notation based on syntax and semantics," in *Aspect-Oriented Requirements Engineering*, pp. 77–99, Springer Berlin Heidelberg, 2013.

[12] D. L. Parnas, "A technique for software module specification with examples," *Communications of the Association of Computing Machinery*, vol. 15, pp. 330–336, May 1972.