Model-Based Reuse of APIs using Concern-Orientation

Matthias Schöttle
School of Computer Science
McGill University
Montreal, QC, H3A 0E9, Canada
Matthias.Schoettle@mail.mcgill.ca

***Abstract*—Despite the promises of Model-Driven Engineering (MDE) to address complexity and improve productivity, no widespread adoption has been observed in industry. One reason this paper focuses on is reuse, which is essential in modern software engineering. In the context of MDE, poor availability of reusable models forces modellers to create models from scratch. At the same time, reusable code artifacts, such as frameworks/APIs are widespread. They are an essential part when creating software. This paper presents model-based reuse of APIs, which makes use of concern-driven development (CDD) to raise the level of abstraction of APIs to the modelling level. The interface (API) of a framework is modelled using a feature model and design models for each feature enabling their reuse in MDE. Additional information is embedded, such as impacts and protocols, to assist the developer in the reuse process. We discuss how this enables reuse at the modelling level, the required tool support and future work.**

I. Introduction

Model-Driven Engineering (MDE) has been around for some time. Despite its promises to address challenges, such as complexity, maintenance and productivity, no widespread adoption has been observed in industry. The reasons are broad and range from non-technical (cultural, social etc.) to technical ones (e.g., lack of tool support) [1], [2].

Methodical reuse of software artifacts is essential in modern software engineering [3], [4]. Reusable code artifacts, such as class libraries and frameworks/APIs, which are abundantly available and widespread on the web, are often well maintained by continuous maintenance and improvements. They are bundled with extensive documentation, mostly textual in the form of API documentation, tutorials etc., but also code examples (snippets or runnable demos). However, API documentation can still fall short in being ambiguous, outdated or incomplete [5]. Furthermore, in order for a developer to decide which reusable artifact to choose over another to accomplish a specific task, typically documentation or question-and-answer websites need to be consulted to discover their impacts on high-level properties, such as performance, memory footprint, and so on, where the information is informally contained.

The challenge, that this paper focuses on, is reuse of existing reusable artifacts at the modelling level. For example, existing code might not be acccessible at the modelling level, making it more difficult to bridge from the models to the code. Another reason that could prevent someone from using MDE can be that there are not enough models to reuse. While there is a lack of evidence to support this claim, MDE approaches mainly focus on forward-engineering (creating models from the beginning of the software development process) or extracing models from code (reverse engineering) to continue with a MDE philosophy after.

Concern-Driven Development (CDD) aims to address this by introducing a new unit of reuse called *concern* [6]. A concern encapsulates different models of a domain of interest at different levels of abstraction. For instance, the concern provides at least the different variations and the impact these have on high-level system qualities.

In this paper we describe how we use CDD to address the aforementioned problems and present model-based reuse of APIs, which raises the level of abstraction of APIs to the modelling level. A developer can thus define a model interface for an existing reusable code artifact without having to model the code artifact itself. This raises the existing reusable artifact to the modelling level, thus enabling the (re)use of an API within an application that is modelled using MDE. In addition, it provides the ability to take advantage of the high-level and formal information encapsulated within a concern. The advantages of MDE are exploited to augment the interface with this information, which assists a user when reusing an API.

The remainder of the paper is structured as follows. Section II introduced concern-orientation and the three reuse interfaces it promotes. Section III illustrates our idea based on an example framework called Minueto. Tool support is essential in the context of MDE and especially CDD. Therefore, Section IV explains the required tool support that is necessary for model interfaces of APIs, and the last section draws some conclusions and gives an overview of future work.

II. Background

In contrast to the focus of classic Model-Driven Engineering (MDE) on models, the main unit of abstraction, construction, and reasoning in Concern-Driven Software Development (CDD) is the *concern* [6]. CDD seeks to address the challenge of how to enable broad-scale, model-based reuse. A concern is a unit of reuse that groups together software artifacts describing properties and behaviour of a domain of interest to a software engineer at different levels of abstraction. A concern provides a three-part interface. The *variation interface* describes required design decisions and their impact on high-level system qualities, both explicitly expressed using a feature

model and impact models in the concern specification. The *customization interface* allows the chosen variation to be adapted to a specific reuse context, while the *usage interface* defines how the functionality encapsulated by a concern may eventually be used.

Building a concern is a non-trivial, time-consuming task, typically done by or in consultation with a domain expert (subsequently called the *concern designer*). On the other hand, reusing an existing concern is extremely simple, and essentially involves three steps for the *concern user*:

1) Selecting the feature(s) of the concern with the best impact on relevant goals and system qualities from the variation interface of the concern.
2) Adapting the general models of features of the concern that were selected to the specific application context based on the customization interface by mapping them.
3) Using the functionality provided by the selected concern features as defined in the usage interface within the application.

The feature model's features are organized within a tree structure, where a relationship between the parent and child feature determines constraints for valid selection. A feature can be *mandatory* or *optional*, or all child features can be within an *XOR* or *OR* relationship. Different models at different levels of abstraction can provide a realization model for each feature. For instance, the design of a concern can be provided by design models that entail class, sequence and state diagrams. Each feature provides the functionality for just that feature, contributing to the design of the parent feature's realization model. In order to accomplish this, aspect-oriented techniques are used.

In general, MDE approaches rely heavily on tool support. Tool support is even more important in the context of CDD, in particular for the concern user. The tool needs to guide the user for selecting variations (making valid selections) and evaluating impacts (allowing the user to perform trade-off analysis between different selections). In addition, the tool needs to hide the complexity of the composition of models and provide validation to ensure proper customization and usage.

III. Concern-Oriented Interfaces for APIs

This section provides an overview of our approach to raise the level of abstraction for APIs to the modelling level and is illustrated using an example framework called Minueto [7], [8]. Minueto is a Java framework, which provides an abstraction layer on top of Java 2D to simplify the creation of 2D multi-platform games. It takes care of the difficult technical parts of game programming in order to allow a developer (the user of the framework) to focus on the game logic. The framework provides different window modes, shapes and event handling and is shipped with different documentation, such as API documentation (based on *Javadoc*), *How To* and Frequently Asked Questions (FAQ) and several runnable code examples that showcase how to accomplish specific functionality.

A. Concernification

Concernification is the process of creating a concern interface for an existing reusable artifact (API) in order to facilitate its use within MDE [9]. The concern interface provides the variations and other aspects of this artifact from the user's perspective, i.e., for a user that reuses that artifact. Therefore, the user-perceived features of the API need to be determined and organized using their relationships within a feature model. Each feature covers a specific subset of the API that is required to successfully use that particular feature. We *concernified* the Minueto framework by hand using the provided runnable code examples. Each example covers at least one use case. However, only a subset of the complete API is covered, which in addition required us to consider the API documentation in order to identify and place them appropriately. Figure 1 shows the feature model of Minueto, which was obtained through several iterations.

This confirms what we described in the previous section, that in order to create a concern, thorough knowledge is required. In the ideal case, this is done by a domain expert, for instance one or several framework developers, who have deep knowledge of the reusable artifact. Considerable effort is required, but we deem this acceptable, if the API is reused a lot. Furthermore, the concern interface can evolve over time. Adding one additional feature requires a lot less effort. This also supports the use case of open source development, where anyone can contribute.

B. Designing an API Subset

In order to facilitate tailoring of the API to the user's needs, the API needs to be decomposed into the determined features. This allows that the user is presented with a subset of the API based on the desired features, which the user selected when reusing the API through the concern interface. Based on the user's selection, each subset of the API located in the design models of selected features is combined.

In the previous section, when the features were determined, a mapping between the used API classes and its corresponding feature was established. Based on this, the API is split across the features, which means that a feature can contain more than one class. However, more fine-grained separation might be desired, for instance, when a specific operation of a class belongs to a different feature. Then, the operation can be moved out of that class and the class with just that operation is added to another design model of a different feature. When combining design models, classes with the same name will be merged.

Figure 2 shows the Minueto feature *Windowed*, which provides the ability to create a game in windowed mode. *Windowed* extends the design realization model of *Surface* by only contributing this functionality, i.e., the interface `MinuetoWindow` is already defined in *Surface* including all its operations.

Similarly, the feature *Keyboard* (sub-feature of *Interactive*) illustrated in Figure 3 provides a keyboard handler. In addition, it provides the required operations to register and unregister

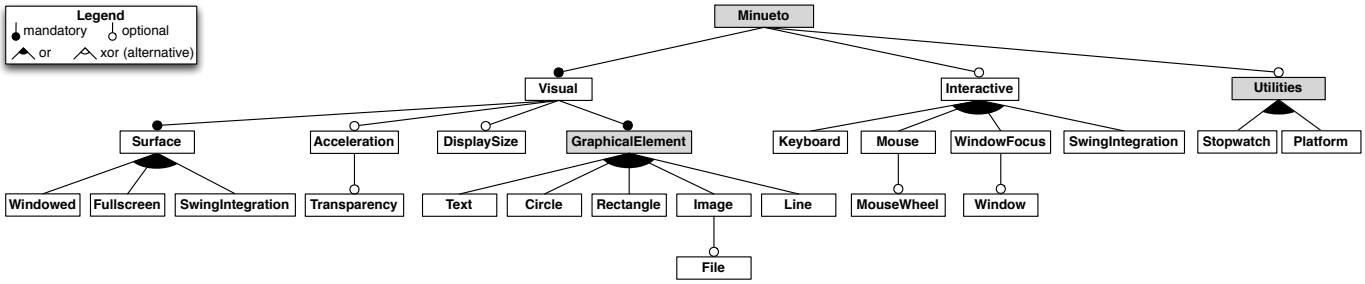

Figure 1. Hand-Made Minueto Feature Model–(features with white background contain parts of the interface, in contrast to those with a grey one that only provide a logical grouping of child features)

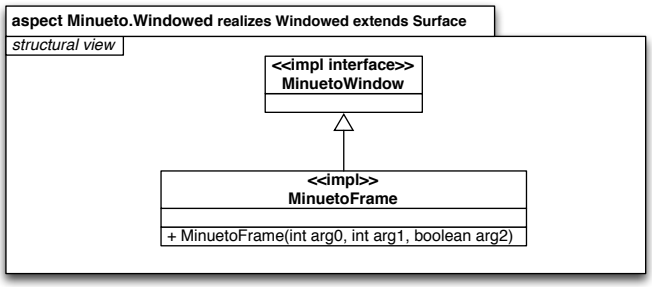

Figure 2. The Interface of the Feature *Windowed*

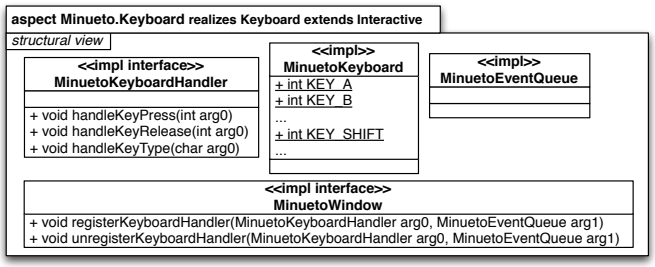

Figure 3. The Interface of the Feature *Keyboard*

the handler from a window, which is not present in the `MinuetoWindow` interface of the visual feature group. These operations were separated from that interface and the interface class with just those operations added to the *Keyboard* feature.

When a user selects *Windowed* and *Keyboard*, both those features and their parents are part of the selection and their design realization models will be combined. Figure 4 shows the resulting composed model, which presents the subset of the API based on the user's desired selection. It was obtained by combining the models of *Windowed*, *Surface*, *Visual*, *Keyboard* and *Interactive*. Classes with the same name are merged by combining their operations and attributes. For instance, the `MinuetoWindow` class now also contains the operations for keyboard handler registration. The user can then use these classes and operations within the application model.

C. Benefits

The previous section described how the variations of an API can be modelled and decomposed in order to provide only the interface of interest to the user. It also provides a compact and organized view of the user-perceivable features provided by an

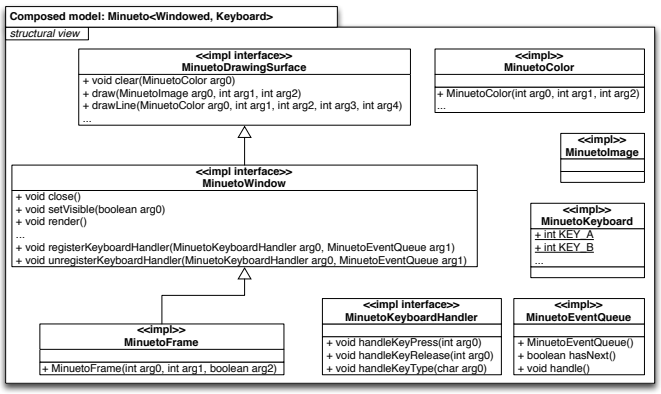

Figure 4. Generated Interface of the Selected Features *Windowed* and *Keyboard*

API. However, a concern interface can provide the following additional benefits:

- Traceability can be provided, which allows the user to see which elements belong to which feature in the composed model of the API.
- Impacts on high-level system qualities can be incorporated to give guidance to the user when selecting features on how they impact non-functional properties.
- Usage protocols can be incorporated to formally specify the protocol of different classes on how they can be used. Correct use of the framework can then be ensured in the application models.
- Partial structure and behaviour can be integrated, which forces the user to provide mappings to application-specific elements using the customization interface. For instance the fact that a certain interface needs to be implemented.
- Other artifacts that usually are shipped with an API or are relevant for reuse could be integrated in the future (as a different kind of model).

These help decrease the risk of a user making mistakes when reusing an API, which can often lead to bugs or vulnerabilities in software systems.

IV. Tool Support

As described in Section II, tool support is crucial for concern-orientation. It is important for both concern designer and user to be provided with proper tool support. Especially

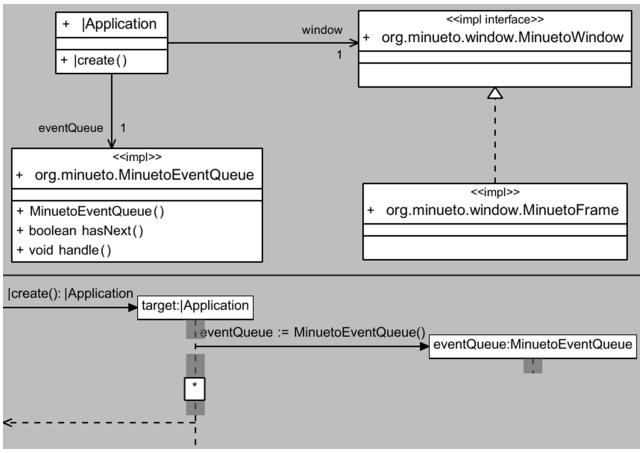

Figure 5. Example of Structure and Behaviour in TouchCORE

for the latter it is essential to hide the complexity of CDD and provide guidance throughout the reuse process following the three steps outlined in Section II. In addition, support to model existing API elements (classes, operations etc.) in order to differentiate them from application-specific elements is required.

We use TouchCORE [10], a multi-touch enabled, concern-oriented software design modelling tool that supports feature and impact models, as well as class, sequence and state diagrams. TouchCORE has recently been extended to support concern-orientation [11], [12] and has been further extended to add support for impact models and their evaluations, as well as improvements to the concern reuse support. Therefore, using TouchCORE, it is possible to do the full concernification as described in the previous section. TouchRAM, the predecessor of TouchCORE, already provided support to create design models using class, sequence and state diagrams, and support to use aspect-oriented techniques to address separation of concerns [13], [14].

In order to be able to import *implementation classes* that are defined by a programming language class library (such as those provided by Java) or a framework, it is possible to import them into a design model [15]. The modeller can choose, which implementation class to import. At first, the class is empty. We decided to only show the ones used by the user, since it might have a large amount of operations. The modeller can then import those operations that are needed. Whenever an operation uses another class from the same framework or programming language (e.g., as a return or parameter type), it is automatically imported. In addition, the hierarchy (both extends and interface implementations) of implementation classes is now supported, which wasn't possible until recently. This means that on import, if the imported class is a super or sub class of an existing implementation class within the class diagram, the hierarchy is shown accordingly. An example, as seen in TouchCORE, is shown in the top part of Figure 5.

The structure illustrated in Figure 5 presents an example of the partial structure and behaviour mentioned as one of the benefits in the previous section. In this case, the modeller needs to provide a mapping for `|Application` and its constructor to corresponding elements of the model. This class will then have the associations to the implementation classes. Furthermore, using aspect-oriented techniques, we advice the constructor. An event queue is instantiated before the actual behaviour (represented by the box containing the asterisk) of an application's constructor as shown in the bottom part of Figure 5, because it is required to provide interactivity.

V. Conclusion and Future Work

This paper presented an overview of concern interfaces for APIs in order to facilitate the reuse of reusable code artifacts on the modelling level by raising their level of abstraction to it. The variations of an API are presented to the user in a concise way, showing the user-perceived features, from which the user can choose. When selecting, the user is able to do a trade-off analysis using impacts on high-level system qualities. Once a selection is made, the customized API (a subset of the API) is presented to the user showing only the relevant parts of the interface required for the desired features. Concernification of an API allows to incorporate additional information, which supports the user upon reuse using the concern interface. This also helps to reduce mistakes by the user during reuse, for instance, not calling a certain operation.

In the future, we plan to *concernify* a larger, more relevant framework, such as the Android SDK, in order to ensure the feasibility of the approach. Ideally, the concern interface is bundled along with the API's code. Because of this, and the fact that it is more difficult to concernify a larger framework with thousands of classes by hand, we intend to perform this (semi-)automatically. This way, a feature model can be presented to the user, which, if necessary, can then be adjusted. To validate the automated concernification, we plan to empirically evaluate it to gather evidence on whether (i) the automated detection works well and (ii) this actually helps users. We believe that even more information/artifacts than those described in this paper can be incorporated more formally into a concern interface of APIs, which would improve API reuse and distribution as a whole.

References

[1] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical Assessment of MDE in Industry," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, (New York, NY, USA), pp. 471–480, ACM, 2011.

[2] B. Selic, "What will it take? a view on adoption of model-based methods in practice," *Software & Systems Modeling*, vol. 11, no. 4, pp. 513–526, 2012.

[3] C. W. Krueger, "Software Reuse," *ACM Comput. Surv.*, vol. 24, pp. 131–183, June 1992.

[4] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies," *Empirical Software Engineering*, vol. 12, no. 5, pp. 471–516, 2007.

[5] G. Uddin and M. P. Robillard, "How API Documentation Fails," *Software, IEEE*, vol. 32, pp. 68–75, July 2015.

[6] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-Oriented Software Design," in *MODELS 2013*, pp. 604–621, Springer, 2013.

[7] "Official Minueto Website." http://minueto.cs.mcgill.ca/.

[8] A. Denault and J. Kienzle, "Minueto, a Game Development Framework for Teaching Object-Oriented Software Design Techniques," in *FuturePlay 2006*, 2006.

[9] M. Schöttle and J. Kienzle, "Concern-Oriented Interfaces for Model-Based Reuse of APIs," in *MODELS 2015*, 2015. *to be published*.
[10] "TouchCORE Website." http://touchcore.cs.mcgill.ca.
[11] N. Thimmegowda, O. Alam, M. Schöttle, W. A. Abed, T. Di'Meco, L. Martellotto, G. Mussbacher, and J. Kienzle, "Concern-Driven Software Development with jUCMNav and TouchRAM," in *Demonstration at MODELS*, 2014.
[12] M. Schöttle, N. Thimmegowda, O. Alam, J. Kienzle, and G. Mussbacher, "Feature Modelling and Traceability for Concern-Driven Software Development with TouchCORE," in *Companion Proceedings of MODULARITY 2015*, pp. 11–14, ACM, 2015.
[13] W. Al Abed, V. Bonnet, M. Schöttle, O. Alam, and J. Kienzle, "TouchRAM: A multitouch-enabled tool for aspect-oriented software design," in *SLE 2012*, no. 7745 in LNCS, pp. 275 – 285, Springer, 2012.
[14] M. Schöttle, "Aspect-Oriented Behavior Modeling In Practice," M.Sc. Thesis, Department of Computer Science, Karlsruhe University of Applied Sciences, September 2012.
[15] M. Schöttle, O. Alam, F.-P. Garcia, G. Mussbacher, and J. Kienzle, "TouchRAM: A Multitouch-enabled Software Design Tool Supporting Concern-oriented Reuse," in *Companion of Modularity:2014*, pp. 25–28, ACM, 2014.