Concern-Oriented Interfaces for Model-Based Reuse of APIs

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

***Abstract*—Reuse is essential in modern software engineering, but limited in the context of MDE by the poor availability of reusable models. On the other hand, reusable code artifacts such as frameworks and libraries are abundant. This paper presents an approach to raise reusable code artifacts to the modelling level by modelling their API using concern-oriented techniques, thus enabling their use in the context of MDE. Our API interface models contain additional information, such as the encapsulated features and their impacts, to assist the developer in the reuse process. Once he has specified his needs, the model interface exposes only the API elements relevant for this specific reuse at the model level, together with the required usage protocol. We show how this approach is applied by hand to model the interface of a small GUI framework and outline how we envision this process to be performed semi-automatically.**

I. Introduction

Model-Driven Engineering (MDE) advocates the use of different modelling formalisms during software development, so that the right level of abstraction is chosen to reason about the system under development depending on the needs. To move towards an executable implementation, high-level, problem-centric models are gradually refined or transformed to progressively integrate solution and platform details. MDE can also simplify the management, understanding and reasoning about complex systems, because different modelling notations naturally push developers to express properties of a system according to different points of interest, which promotes the principle of separation of concerns.

Although MDE technology has been available for more than a decade, adoption of MDE in industry is slow [1]. One of the possible reasons, which our paper focuses on, is that model reuse between different development projects is difficult. Typically, models for a system under development are created from scratch, rather than reusing already existing models. However, methodical reuse of software artifacts is considered key to software engineering [2]. Especially companies are interested in reusing existing software artifacts in order to amortize development costs by increasing quality and productivity [3]. Furthermore, model libraries or repositories for reusable models are very uncommon. While some model repositories exist [4], they mainly focus on collecting example models or do not have a large community that provides them.

This stands in contrast to reusable code artifacts in the form of class libraries and frameworks, which are abundant, readily available on the web, and are often well maintained by the user community, continuously improved, and come with good quality textual documentation and different forms of code examples (tutorials, demos etc.). However, documentation can still be ambiguous or incomplete [5]. Class libraries of modern programming languages provide hundreds of classes that encapsulate commonly used algorithms, data structures, and mechanisms for input and output. Application frameworks are also composed of classes, but usually focus on providing reusable structure and behaviour related to a specific domain (e.g., graphical user interfaces, persistence, ...). Most often, frameworks impose an application architecture, drive the execution control flow, and require the programmer to tailor the framework to their needs and integrate the application's behaviour by implementing interfaces or extending classes provided by the framework.

In this paper, we propose and outline an approach based on concern-oriented technology that allows a developer to define a model interface for an API. Doing this has many benefits. First, although the code artifact itself is not modelled, it can be (re)used in an application that is developed following MDE principles. This allows the developer to reason about the system under development at multiple levels of abstraction and from multiple points of view, and still take advantage of the functionality and quality of the reused code. Second, while framework documentations typically only include the API, textual documentation and some code examples that demonstrate the use of the API, our concern-oriented model interface has the following additional advantages:

- provides a high-level, formal, organized view of the user features that a framework provides, together with dependency constraints among them using a feature model,
- provides guidance to the user on how different alternatives that the framework offers impact the non-functional properties and qualities of the system that is being built,
- only exposes a subset of the framework's API tailored to the user's needs, reducing the API complexity exposed to the user to a minimum,
- clearly marks the generic elements (classes, operations) of the framework that the user needs to customize to application-specific elements,
- expresses the usage protocol of the different classes exposed in the framework's API formally, and
- model checking can ensure that the application models

make correct use of the framework's API, i.e., they implement all the required interfaces and adhere to the correct usage protocol.

The remainder of the paper is structured as follows. Section II briefly introduces concern-orientation and the three reuse interfaces it promotes, which we use to build the model interface for frameworks. It also overviews Minueto, a small game development framework, that we use in Section III to illustrate our idea. Section IV describes how building concern-oriented model interfaces for frameworks could be automated, and the last section draws some conclusions.

II. Background

A. Concern-Driven Development

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 (models and code, henceforth called simply models) describing properties and behaviour related to any 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 feature 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 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. (3) Using the functionality provided by the selected concern features as defined in the usage interface within the application.

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 do 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.

We use TouchCORE[1] [7], [8] a multi-touch enabled, concern-oriented software design modelling tool that supports feature and impact models, as well as class, sequence and state diagrams.

[1]http://touchcore.cs.mcgill.ca

B. Minueto

Minueto [9], the framework we are using in this paper to illustrate our idea, is a framework written in Java that touts itself as a game SDK. It provides an abstraction layer on top of Java 2D to simplify the creation of 2D multi-platform games by taking care of the difficult technical parts of game programming related to graphics and input handling, thus allowing the users to focus more on the game logic. The framework provides different window modes, shapes, hardware graphics acceleration and transparency, and integrates event handling with the render loop. Minueto (we use the current available version 2.0.1) is shipped with several documentation artifacts:

- A *How To* section on the Minueto website [10] provides a quick introduction to the framework, and presents details on how certain tasks can be accomplished;
- Several runnable code examples show how to use specific functionality provided by Minueto;
- The API documentation (based on *Javadoc*);
- A list of Frequently Asked Questions (FAQ) that explain common issues encountered by users.

III. Concernifying an API

[6] introduces CDD and the three concern interfaces, and illustrates concern-oriented software design with a simple example. The idea put forward in this paper is to create a concern interface for an *existing code artifact* so that it can be (re)used in the context of MDE. We call this *concernification*, and propose a process that an MDE practitioner or the API designer, that wants to build a bridge between the modelling world and a (or his) framework, can follow to encapsulate this framework within a concern and define the three interfaces.

As described in the previous section, the variation interface of a concern declares the distinctive user-visible aspects and characteristics of the software that a concern modularizes and encapsulates using a feature model. In our case, each feature encapsulates some specific use of the API from the user's perspective. Successful concernification of an API therefore requires the identification of all user-perceived features of a framework and their relationships: *mandatory* or *optional*, *XOR* or *OR*, and cross-tree constraints (*requires* or *excludes*).

Once the features are determined, the API that is being concernified needs to be decomposed according to its features. In CDD, each feature is described by a design model that realizes the feature, and therefore each such design model should contain the subset of the API related to the feature. Hence, it is necessary to identify which API elements (e.g., classes and methods) belong to which feature. To do this, the parent-child relationships between features (and hence also between realization models) need to be taken into account: child models can add additional operations to the classes defined in parent models, and add new classes.

A. Concernifying Minueto

We created a concern of the Minueto framework by hand with the intention to observe key points that will allow the automation of this process. Minueto is a small framework that consists of 60 classes and interfaces in total, of which 32 classes and 8 interfaces are public. In total there are 253 public and protected methods (~6 on average per class).

As described in the previous section, in order to create a concern, thorough knowledge is required, which requires consultation with a domain expert. The authors of this paper therefore first invested time to familiarize themselves with Minueto. Several sources of information were used. This includes the creation of a complete class diagram of the framework to gain an understanding of the big picture. Furthermore, the different forms of documentation outlined in the previous section were studied. Finally, the source code was used to gain deep knowledge and experiment with the examples. Minueto provides 30 small runnable examples that explain how to use the API. Each example mostly showcases one use case (such as drawing a specific shape, handling input etc.).

We constructed the feature model using the examples along with the class diagram. However, the examples do not cover all classes and operations. To be able to integrate the missing elements into the feature model, additional information, such as their API documentation was considered. The feature model was constantly refined based on discussions between the authors, starting at an initial version that was very close to the class hierarchy to the final version that—from our point of view—reflects the user's perspective well.

1) Documenting and Organizing Features: Figure 1 shows the resulting feature model. The main functionality is divided into three clusters. The feature *Surface* provides different kinds of window modes, *GraphicalElement* the various elements that can be drawn and *Interactive* provides event handling. The remaining features provide optional functionality. The features with a grey background are used solely for structuring. They do not have a design model that realizes them, and hence do not contain any classes or methods of the API.

2) Specifying Impacts: In CDD, impacts are specified with impact models such as the one shown in Figure 2 for Minueto. It contains all high-level goals and qualities that experienced API developers or users identified as relevant to consider. The goals are depicted with rounded rectangles, and are connected to each feature of the API that has an impact on the goal with a weighted link. The weights are specified using relative values.

The simplified impact model for Minueto, for example, states that from a performance point of view, the best set of features to use is to select *Fullscreen* and *Acceleration*, but to not select *Transparency*. This selection is not too demanding on system requirements, but it does require the person playing the game to have admin privileges on the machine (in order to switch to fullscreen).

3) Tailoring the API to the User's Needs: When a user wants to use Minueto, he will first be presented with the feature model and requested to make a selection. Based on the user's selection of desired features, the API pieces contained in the realization models of all the selected features are composed to yield an API of Minueto that only contains the classes and methods that the user needs.

To illustrate the result of the composed interface based on a user selection, we present two individual models and the resulting composition. Figure 3 shows the *Windowed* feature. It provides the `MinuetoFrame` class, which allows the creation of a GUI application in window mode. Since it is a subclass of an internal abstract class (and implements the `MinuetoWindow` interface), it declares a constructor and only additional functionality. The remaining methods are defined in the model of the parent *Surface*.

Furthermore, Figure 4 shows the *Keyboard* feature. It provides the required interface to be implemented and its utility class. Furthermore, it extends `MinuetoWindow` and adds the corresponding methods to (un)register a keyboard handler. The event queue is provided by the parent feature *Interactive*.

The result of the composed models is illustrated in Figure 5. It was obtained by merging the models that realize *Window*ed, *Surface*, *Visual*, *Interactive* and *Keyboard*. Classes that exist in multiple models are merged, i.e., combining their operations and attributes. For example, the `MinuetoWindow` class now contains the operations required to register a keyboard handler.

In our TouchCORE tool, the model weaver keeps track of the origin of each model element. This makes it still possible for the user to highlight which part of the generated API belongs to which feature, if needed.

4) Specifying Usage Protocols: Our model interfaces make it possible to specify usage protocols. We use protocol models, which are similar to state diagrams, but support composition and allow a protocol machine to refuse transitions. For further information, the interested reader is referred to [11].

Figure 6 illustrates the usage protocol of the interface `MinuetoWindow`, which describes the render loop that is executed repeatedly to update a window's content. Before any drawing operations can be called, the window needs to be visible. Then, the window has to be either cleared or at least something has to be drawn into it before it can be rendered. This cycle repeats, until the window is closed.

Specifying a usage protocol avoids potential user mistakes. Furthermore, if protocols are specified, the MDE tool can use model checking to ensure that the API is used correctly.

5) Guaranteeing Correct Reuse: To further facilitate correct reuse, we can exploit the *customization interface* provided by CDD. Partial elements can be added to a design model realizing a feature, forcing the user to provide a mapping from the partial elements in the customization interface to the appropriate model elements in his application. For example, the use of the feature *Keyboard* requires a user to implement the `MinuetoKeyboardHandler` interface. By providing a partial class `|KeyboardHandler` and a partial operation for each method that needs to be implemented, the user is forced by the MDE tool to do so. This reduces again the possibility for making mistakes, and therefore allows the user to focus more on the logic of the application under development.

IV. Towards Automated Concernification

Ideally, the developer of an API bundles the source or binary code of the API already with a concern-oriented model

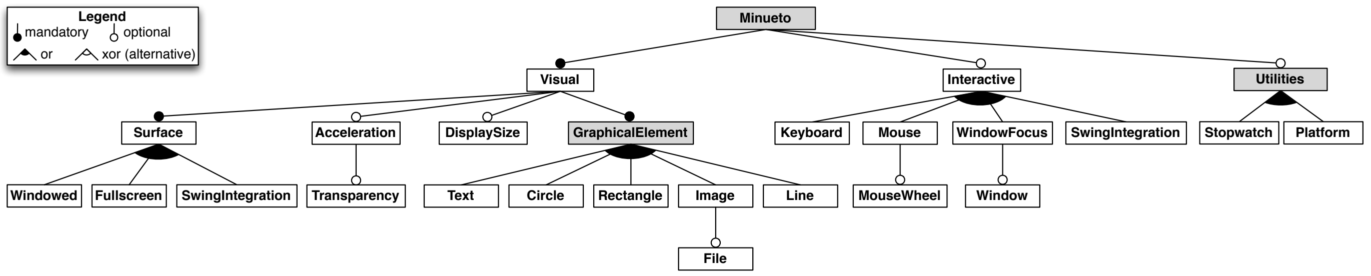


Figure 1. Hand-Made Minueto Feature Model–(features with white background contain parts of the interface, whereas those with a grey one do not)

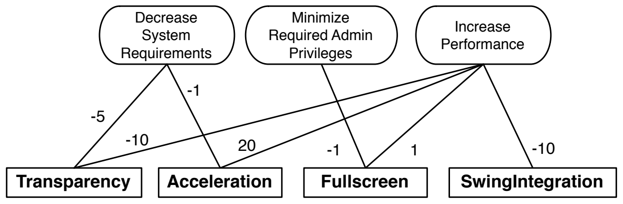


Figure 2. Minueto Impact Model

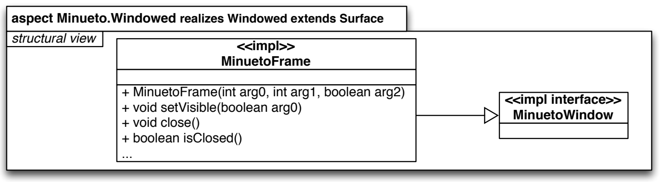

aspect Minueto.Windowed realizes Windowed extends Surface
structural view
<<impl>>
MinuetoFrame
+ MinuetoFrame(int arg0, int arg1, boolean arg2)
+ void setVisible(boolean arg0)
+ void close()
+ boolean isClosed()
...
<<impl interface>>
MinuetoWindow

Figure 3. The Interface of the Feature *Windowed* (sub-feature of *Surface*)

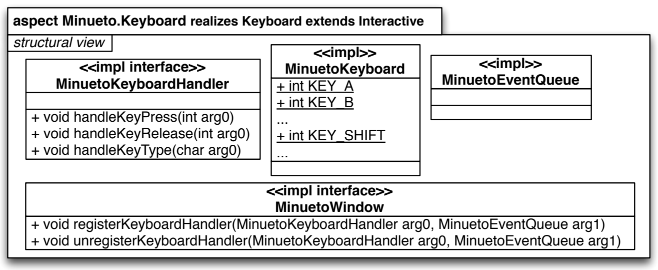

aspect Minueto.Keyboard realizes Keyboard extends Interactive
structural view
<<impl interface>>
MinuetoKeyboardHandler
+ void handleKeyPress(int arg0)
+ void handleKeyRelease(int arg0)
+ void handleKeyType(char arg0)
<<impl>>
MinuetoKeyboard
+ int KEY_A
+ int KEY_B
...
+ int KEY_SHIFT
...
<<impl>>
MinuetoEventQueue
<<impl interface>>
MinuetoWindow
+ void registerKeyboardHandler(MinuetoKeyboardHandler arg0, MinuetoEventQueue arg1)
+ void unregisterKeyboardHandler(MinuetoKeyboardHandler arg0, MinuetoEventQueue arg1)

Figure 4. The Interface of the Feature *Keyboard* (sub-feature of *Interactive*)

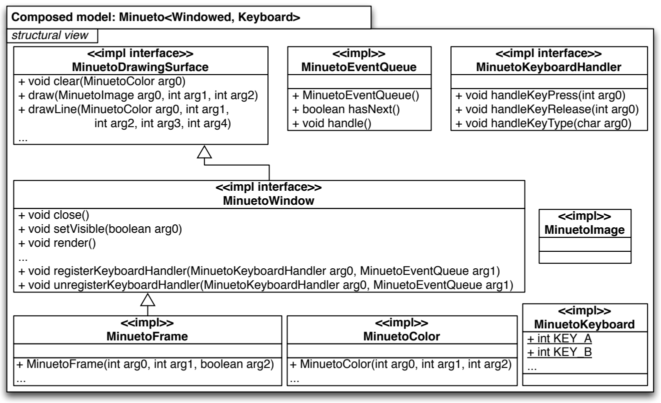

Composed model: Minueto<Windowed, Keyboard>
structural view
<<impl interface>>
MinuetoDrawingSurface
+ void clear(MinuetoColor arg0)
+ draw(MinuetoImage arg0, int arg1, int arg2)
+ drawLine(MinuetoColor arg0, int arg1, int arg2, int arg3, int arg4)
...
<<impl>>
MinuetoEventQueue
+ MinuetoEventQueue()
+ boolean hasNext()
+ void handle()
<<impl interface>>
MinuetoKeyboardHandler
+ void handleKeyPress(int arg0)
+ void handleKeyRelease(int arg0)
+ void handleKeyType(char arg0)
<<impl interface>>
MinuetoWindow
+ void close()
+ void setVisible(boolean arg0)
+ void render()
...
+ void registerKeyboardHandler(MinuetoKeyboardHandler arg0, MinuetoEventQueue arg1)
+ void unregisterKeyboardHandler(MinuetoKeyboardHandler arg0, MinuetoEventQueue arg1)
<<impl>>
MinuetoImage
<<impl>>
MinuetoFrame
+ MinuetoFrame(int arg0, int arg1, boolean arg2)
<<impl>>
MinuetoColor
+ MinuetoColor(int arg0, int arg1, int arg2)
...
<<impl>>
MinuetoKeyboard
+ int KEY_A
+ int KEY_B
...

Figure 5. Generated Interface of the Features *Windowed, Surface*, *Visual*, *Keyboard* and *Interactive*

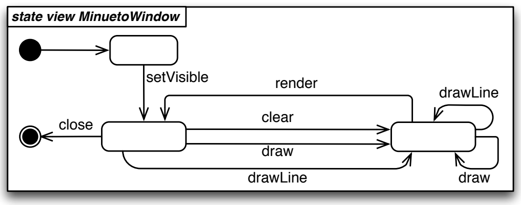


Figure 6. Usage Protocol of the `MinuetoWindow` Interface

interface. When this is not the case, an automated process of obtaining a concern interface from an existing API is needed.

We conducted experiments and determined an initial automated process that can be applied to object-oriented APIs that come with runnable code examples. The intuition behind the algorithm is that code examples usually exemplify how to use a specific feature of the API (or a subset of the features of the API). The detailed processing steps are as follows:

1) *Analyzing OO Tree Structure*: All public classes are extracted from the API, and clustered into tree structures based on their inheritance relationships. Subclasses and classes that implement an interface are potential candidates for sub-features, because they add to the superclass or provide a specialization. Each tree is converted to a feature model, where each feature has the name of the corresponding class in the hierarchy. Since user-perceived features of an API often involve the use of multiple classes, this step yields many features that are false positives. We will call them candidate features, or short *CF*.
2) *Analyzing CF Use*: To understand how CFs are used, we annotate each CF with the set of code examples in which the corresponding class was used in. Usage includes creation of instances, method calls, public field access, exception handling, and subclassing.
3) The relationships between the CFs in each feature model are determined by comparing the sets obtained for each CF in step 2. A set that is equal to the parent CF set leads to the parent-child relationship being mandatory. If a feature becomes mandatory and it has siblings, they become optional. If none of the child features turned mandatory, the sets of all siblings are intersected and their relationship determined as follows:
   - If all of them overlap, the relationship to the parent is OR, since it is possible to use them together;

- If all of them are disjoint, the relationship to the parent is XOR, because it appears that they should not be used together;
- In all other cases, additional intermediate features that do not correspond to a class have to be added representing disjoint subgroups. For example, if sibling CFs *B* and *C* intersect, but CF *D* is disjoint, an additional intermediate CF is added to group *B* and *C* using an OR-relationship. The relationship between the intermediate CF and *D* is set to XOR.

4) Then, all feature models are combined into the biggest one FM_{base}. The root CF of the to-be-merged feature model FM_{merge} is taken and combined according to the following rules:
   a) If the class corresponding to the root feature of FM_{merge} is used as the type of a method parameter in any of the classes represented in FM_{base}, then the closest common ancestor feature *A* covering all those classes is determined, and the entire FM_{merge} is marked to be merged with it. This means the root is marked to be merged with *A*, and any sub-features recursively. This might require adding new intermediate sub-features to adhere to the feature model relationship constraints (as in step 3).
   b) If the class is not used as a parameter type anywhere, it is marked to be added as an optional sibling feature of the root of FM_{base}.
5) Finally, for all marked CFs that were determined during step 4, those that are marked to be added or merged with the same CF and where the corresponding classes are located in the same package, are grouped using an intermediate parent feature with an OR relationship. The intermediate feature is added as an optional parent to CF. This step considers the fact that a developer of an API must have had a reason to create a package. Therefore, we group them under a common parent, assuming that they can be used for related functionality. All other ones that are not part of the same package are merged with the previously determined CF.

Applying this process to the Minueto framework using the 30 runnable examples yields promising results. The obtained feature model is shown in Figure 7. The upper set of feature models illustrates the result of our algorithm after applying steps 1-3, and the lower feature model shows the final feature model obtained using the combination rules described in 4 and 5. The features with a grey background indicate that a merge or grouping occurred.

A. Observations

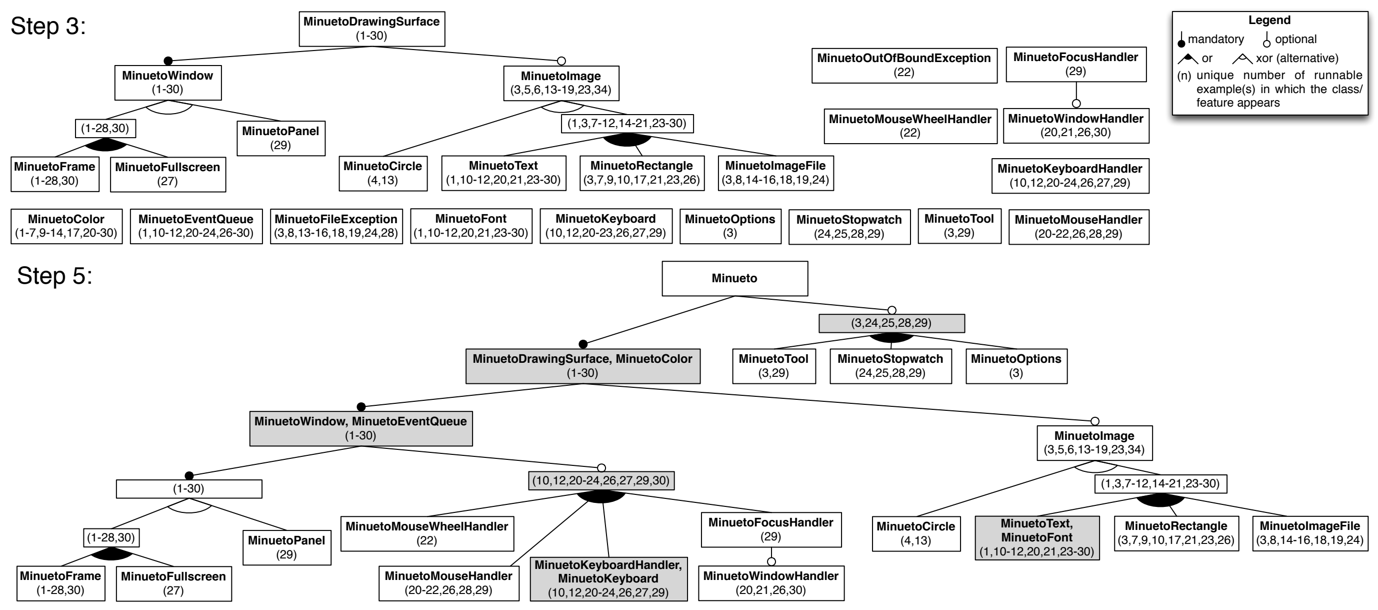
A comparison of the hand-made feature model (Figure 1) and the automatically generated feature model (Figure 7) reveals encouraging structural similarities. Even some of the sub-features that were manually introduced in the hand-made model show up in the generated one (*Surface*, *Interactive* and *Utilities*). The following paragraphs discuss the limitations of our prototype algorithm by commenting on the differences.

- The generated feature model might be missing features if the provided code examples are few or they do not cover the entire API. They need to be discovered by consulting the API documentation, and then inserted manually into the feature model depending on their purpose. For example, Minueto provides the ability to handle *Swing* events using the provided event queue. However, no example code covers this case.
- In the case where most code examples are very narrow, i.e., they focus on showcasing a single feature, there will be no examples that exemplify which features can be used in combination. This leads to false XOR relationships. For instance, `MinuetoCircle` is never used with any other graphical shape, and therefore shows up as an exclusive alternative to all other shapes in the generated feature model. Another case is `MinuetoPanel`. This is probably due to the fact that in both cases there is only one example that uses these classes.
- Currently, in step 2 of our automated process we consider only the usage of the entire classes. However, it could be beneficial to consider use at a finer grain, i.e., reason at the level of operations. For example, the method `drawLine(...)` of the interface `MinuetoDrawingSurface` can be invoked to draw a line. In the hand-made feature model (Figure 1), this single operation is modularized within a sub-feature *Line* of *GraphicalElement*. The features *DisplaySize*, *Platform*, *Acceleration* and *Transparency* represent additional examples where classes were split across features that reflect the user's perspective more appropriately.
- Even code examples of high quality APIs may contain errors. This reduces the accuracy of an automated extraction. For instance, as shown in Figure 7, the usage set of `MinuetoEventQueue` is a superset of the group of handlers. This is due to the fact that example 1 and 11 instantiate an event queue, but never use it to register a handler, and hence never use it at all at runtime. This mistake, which could have been prevented with a usage protocol interface (see subsection III-A4), prevented our algorithm to merge the *EventQueue* (`MinuetoEventQueue`) feature with the *Handler* subtree.

All of the above shows that the auto-generated feature model should be presented to the user to provide the opportunity to move, merge features and change relationships, if necessary. This includes renaming to reflect their meaning appropriately.

V. Conclusion and Outlook

This paper explained how to encapsulate existing code APIs behind a model interface, which not only makes them (re)usable in the context of MDE, but also results in several additional benefits. By providing a concern-oriented interface, it is possible to concisely communicate the features that an API provides to the user with a feature model. The impact model provides insight on how each feature affects high-level goals and qualities. This helps the user to make her selection, and once she decides, the API presented to the user at the



Step 3:
MinuetoDrawingSurface (1-30)
MinuetoWindow (1-30)
MinuetoImage (3,5,6,13-19,23,34)
(1-28,30)
MinuetoPanel (29)
(1,3,7-12,14-21,23-30)
MinuetoFrame (1-28,30)
MinuetoFullscreen (27)
MinuetoCircle (4,13)
MinuetoText (1,10-12,20,21,23-30)
MinuetoRectangle (3,7,9,10,17,21,23,26)
MinuetoImageFile (3,8,14-16,18,19,24)
MinuetoOutOfBoundException (22)
MinuetoFocusHandler (29)
MinuetoMouseWheelHandler (22)
MinuetoWindowHandler (20,21,26,30)
MinuetoKeyboardHandler (10,12,20-24,26,27,29)
MinuetoColor (1-7,9-14,17,20-30)
MinuetoEventQueue (1,10-12,20-24,26-30)
MinuetoFileException (3,8,13-16,18,19,24,28)
MinuetoFont (1,10-12,20,21,23-30)
MinuetoKeyboard (10,12,20-23,26,27,29)
MinuetoOptions (3)
MinuetoStopwatch (24,25,28,29)
MinuetoTool (3,29)
MinuetoMouseHandler (20-22,26,28,29)
Legend
mandatory
optional
or
xor (alternative)
(n) unique number of runnable example(s) in which the class/feature appears
Step 5:
Minueto
(3,24,25,28,29)
MinuetoDrawingSurface, MinuetoColor (1-30)
MinuetoTool (3,29)
MinuetoStopwatch (24,25,28,29)
MinuetoOptions (3)
MinuetoWindow, MinuetoEventQueue (1-30)
MinuetoImage (3,5,6,13-19,23,34)
(1-30)
(10,12,20-24,26,27,29,30)
(1,3,7-12,14-21,23-30)
MinuetoMouseWheelHandler (22)
MinuetoFocusHandler (29)
MinuetoCircle (4,13)
MinuetoText, MinuetoFont (1,10-12,20,21,23-30)
MinuetoRectangle (3,7,9,10,17,21,23,26)
MinuetoImageFile (3,8,14-16,18,19,24)
(1-28,30)
MinuetoPanel (29)
MinuetoFrame (1-28,30)
MinuetoFullscreen (27)
MinuetoMouseHandler (20-22,26,28,29)
MinuetoKeyboardHandler, MinuetoKeyboard (10,12,20-24,26,27,29)
MinuetoWindowHandler (20,21,26,30)

Figure 7. Automatically determined Minueto Feature Model after Step 3 (top) and Step 5 (bottom; grey features show where a merge or grouping took place).

modelling level is tailored to only show the structure and functionality relevant for the user. The customization interface is used to ensure correct adaptation of the API to the reuse context, and model-checking usage protocols can be exploited to enforce correct use of the API.

While *concernification* involves considerable effort, we believe that it is acceptable if the API is reused significantly. Ideally, it is performed by one or several designers of the reusable artifact, which reduces the effort as they already have deep knowledge of the artifact. Subsequently, evolution of an existing concern interface, for example to add a new feature, is a lot easier. APIs are typically long-lived, and hence impacts can also be added slowly over time. This could be done even in a collaborative, open source user community environment.

If this approach is embraced by the software engineering community, concern interfaces would become a new, formal form of API documentation provided in addition to the traditional, informal form, which can easily be ambiguous and incomplete [5]. This would allow MDE tools and programming IDEs to considerably streamline the reuse process, e.g., by reasoning about feature selections and calculating impacts, by optimizing selections, by providing traceability for features and their API elements, by allowing concern designers to define default/pre-made configurations (selections) for the user, by verifying correct customization and adherence to usage protocols, and by restricting code completion suggestions to those conforming to the usage protocol.

We have illustrated our idea by presenting a concern-oriented interface of a small GUI framework called Minueto, and outlined a prototype algorithm for extracting the features of an API automatically. While automated concernification is not a must, it would simplify adoption of our approach. We are therefore planning to extend our prototype algorithm by integrating related work on extracting features from documentation, e.g., using natural language processing techniques as described in [12]. How well such an algorithm performs on frameworks of bigger size, and how useful the concern-oriented interface actually is for MDE practitioners remains to be investigated. We are also planning to extend the customization interface to support customization through annotations, naming conventions and external files (XML). Furthermore, we plan to investigate how to generate tutorials based on a user's feature selection to more concisely communicate to the user how the selected features need to be used.

References

[1] J. Whittle, "The Truth about Model-Driven Development in Industry - and Why Researchers Should Care." http://www.slideshare.net/jonathw/whittle-modeling-wizards-2012/, 2012.

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

[3] 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.

[4] J. Cabot, "One (virtual) model repository to rule them all." http://modeling-languages.com/one-virtual-model-repository-rule/, 2014.

[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] 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.

[8] 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.

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

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

[11] W. A. Abed, M. Schöttle, A. Ayed, and J. Kienzle, "Concern-Oriented Behaviour Modelling with Sequence Diagrams and Protocol Models," in *BM-FA. Post-proc*, vol. 6368 of *LNCS*, pp. 250–278, Springer, 2015.

[12] C. Treude, M. Robillard, and B. Dagenais, "Extracting development tasks to navigate software documentation," *Software Engineering, IEEE Transactions on*, vol. 41, pp. 565–581, June 2015.