Model-Based Reuse of Framework APIs

Bridging the Gap Between Models and Code

Matthias Schöttle

A thesis submitted to McGill University in partial
fulfillment of the requirements of the degree of

Doctor of Philosophy

School of Computer Science
McGill University
Montréal, Québec, Canada

October 2019

Abstract

Reuse is considered key to software engineering and is very common at the implementation level. Many reusable libraries and frameworks exist and are widely reused. However, in the context of Model-Driven Engineering (MDE) reuse is not very common. Most modelling approaches do not support reuse, requiring a user to start their modelling activity either from scratch or copy and paste pieces from other models.

This thesis provides a bridge for reusable units between implementation and modelling. We apply the principles of Concern-Oriented Reuse (CORE), a next-generation reuse technology, to lift existing frameworks up from the programming level to the modelling level. The level of abstraction of the API of existing frameworks is raised to the modelling level to facilitate their reuse within design models that are integrated within an MDE process. In addition, the benefits of the higher level of abstraction are exploited to formalize otherwise informally provided information, such as which features the framework provides, the impact of each feature on high-level goals and non-functional qualities, how to adapt the framework to the reuse context, and how the API of each feature is to be used. This thesis defines an automated algorithm that analyses the code of a framework and example code that uses the framework to produce an interface that 1) lists the user-perceivable features of the framework organized in a feature model, and 2) modularizes the API of the framework API according to each feature. The algorithm is implemented and validated on two small frameworks and the Android Notifications API along with an empirical user study.

To smoothen the transition from a high-level abstraction to a low level of abstraction, i.e., from models to code, this thesis addresses the difficulty caused by the finality of signatures. We identify and discuss four difficult situations for defining high-level interfaces at the modelling level, and present evidence that shows that these situations also exist at the implementation level. The *signature extension* approach is introduced to CORE allowing interfaces to encompass diverse implementation variants and to be evolved at a fine level of granularity across groups of features. We re-design two reusable concerns to show that the approach addresses the four difficult situations.

Abrégé

La réutilisation est essentielle dans l'ingénierie logicielle. La réutilisation de librairies et de frameworks (cadre d'application) est très utilisée lors du niveau de l'implémentation. Il existe un nombre important de frameworks et ils sont réutilisés fréquemment. Cependant, dans le contexte de l'ingénierie dirigée par les modèles (Model-Driven Engineering (MDE)), la réutilisation de modèles n'est pas très présente. La plupart des méthodes de modélisation ne supportent pas la réutilisation et demandent à l'utilisateur de soit commencer la modélisation de zéro ou de reprendre des morceaux d'autres modèles.

Cette thèse crée un lien entre l'implémentation et la modélisation pour des unités réutilisables. Nous utilisons les principes de la réutilisation par préoccupation (Concern-Oriented Reuse (CORE)), une nouvelle génération de technologies de réutilisation, pour monter les frameworks du niveau de l'implémentation vers la modélisation. Le niveau d'abstraction de l'interface de programmation d'application (API) de framework existant est monté au niveau de la modélisation pour faciliter leurs réutilisations dans un modèle de conception qui est intégré dans le processus MDE. De plus, les avantages d'un niveau d'abstraction plus élevé sont utilisés pour formaliser de l'information qui autrement n'existe que d'une manière informelle. Celle-ci nous permet de savoir quelles fonctionnalités le framework propose, l'impact de chaque fonctionnalité sur des buts de haut niveau et des qualités non-fonctionnelles, comment adapter le framework dans un contexte de réutilisation et comment l'API de chaque fonctionnalité doit être utilisée. Cette thèse définit un algorithme automatisé qui analyse le code d'un framework et les exemples de code qui utilisent l'API pour produire une interface qui 1) liste les fonctionnalités utilisateur du framework organisée dans un modèle de fonctionnalité, et 2) modularise l'API du framework selon chaque fonctionnalité. L'algorithme est implémenté et validé avec deux petits frameworks ainsi qu'avec une étude utilisateur empirique de l'API de notification d'Android.

Pour faciliter la transition depuis un haut niveau d'abstraction vers un niveau d'abstraction plus bas, c'est-à-dire, des modèles vers le code, cette thèse aborde les difficultés causées par la finalité des signatures de méthode. Nous identifions quatre situations complexes lors de la création d’interfaces haut niveau au niveau du modèle et présentons des preuves qui montrent que ces situations existent également au niveau de l’implémentation. La méthode de l’extension de signature est introduite à CORE, permettant aux interfaces d’intégrer plusieurs variantes d’implémentation et d’évoluer dans le détail à travers un groupe de fonctionnalités. Nous redéfinissons deux concerns réutilisables pour montrer que cette méthode répond à ces quatre situations complexes.

Acknowledgements

Here it finally is! A long journey is coming to an end and there are many people I would like to thank for their support, without which this thesis could not have been completed.

First and foremost, I express my sincerest gratitude to my advisor Jörg Kienzle for his trust, guidance, patience, friendship, and generous support. He gave me the freedom and flexibility to explore my own interests and also to lead the development of our TouchCORE tool. I am very grateful for his constant optimism and confidence in me, and his patience with my perfectionism. He always kept the big picture in mind and brought me back when I was stuck in the details. I very much enjoyed the intense intellectual discussions, sense of humour, and discussions outside of research. I also truly appreciate the privilege to attend conferences and many Bellairs workshops that allowed me to meet many smart people and gave a stimulating and enriching experience.

I am grateful and thank the members of my committee, Jin Guo, Gunter Mussbacher and Houari Sahraoui, as well as my external examiner Krzysztof Czarnecki, and my internal examiner Clark Verbrugge, for reviewing my work and their valuable feedback and comments. I also thank Alexandre Denault and Michael Hawker for taking time out of their busy schedules for the interviews, as well as the participants of the Android Notifications API study for providing their feedback.

During these years I was fortunate to work with many colleagues and form friendships in the Software Engineering Lab. Specifically, I would like to thank Berk Duran for his continued friendship and bringing life into the lab. I thank everyone for making our lab a great place and all the interesting discussions, lunches and fun we have had together: Berk, Céline, Julien, Max, Nirmal, Nishanth, Omar, and Rohit. I had the opportunity to lead the development of the TouchCORE tool and work with and co-supervise many students: Andrea, Céline, Emmanuel, Franz-Philippe, Nirmal, Nishanth, and Rohit. During the summers we were often joined by interns from France that made it a lot of fun (although not that productive for research :-)). I wish to thank (in order of appearance): Laura, Thomas, Cécile, Romain, Jehan, Arthur, and Josué. I also thank everyone else of the CORE group for the fruitful and intense discussions and feedback during our group meetings. Special thanks goes to Arthur for his help with translating the abstract to French.

I would like to express my gratitude to my friends and family for their love, support, patience and understanding. I especially thank my parents for their support and understanding when I came to visit but had to work. I also thank the parents of my wife for their hospitality during parts of the thesis writing. Special thanks goes to my friend Harald for his genuine interest in my work and progress, and his help in disseminating the invitation of my user study.

I thank my son Finn for his joy, smiles and playful energy.

And finally, words can not express the gratitude I have for my wife Stéphanie. Without her, this thesis would not have been possible. Thank you for embarking with me on this journey and your wonderful love, support, and patience. Thank you for all your sacrifices, and your devotion in looking after Finn. Thank you for your constant encouragement and always believing in me when I doubted myself.

Related Publications

Earlier versions of parts of this thesis were published as listed below. The first author is the main author who contributed the majority of the work to the publication.

- Matthias Schöttle and Jörg Kienzle. **Concern-Oriented Interfaces for Model-Based Reuse of APIs**. In *Proceedings of the 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2015)*, pp. 286–291. IEEE Press, 2015. (Chapter 3)

- Matthias Schöttle and Jörg Kienzle. **On the Difficulties of Raising the Level of Abstraction and Facilitating Reuse in Software Modelling: The Case for Signature Extension**. In *Proceedings of the 11th International Workshop on Modelling in Software Engineering (MiSE 2019, co-located with ICSE)*, pp. 71–77. IEEE Press, 2019. (Chapter 8)

- Matthias Schöttle, Omar Alam, Franz-Philippe Garcia, Gunter Mussbacher, and Jörg Kienzle. **TouchRAM: A Multitouch-Enabled Software Design Tool Supporting Concern-Oriented Reuse**. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity (MODULARITY 2014)*, pp. 25–28. ACM, 2014. (Section 5.6.1)

- Matthias Schöttle, Nishanth Thimmegowda, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. **Feature Modelling and Traceability for Concern-driven Software Development with TouchCORE**. In *Companion Proceedings of the 14th International Conference on Modularity (MODULARITY Companion 2015)*, pp. 11–14. ACM, 2015. (Section 5.6.1)

- Matthias Schöttle, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. **On the Modularization Provided by Concern-Oriented Reuse**. In *Companion Proceedings of the 15th International Conference on Modularity (MODULARITY Companion 2016)*, pp. 184–189. ACM, 2016. (Section 2.4)

The following publications were produced in parallel to the research described in this thesis and are not directly related to this thesis.

- Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. **Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering**. Computer Languages, Systems & Structures 54 (2018), pp. 139–155.

- Céline Bensoussan, Matthias Schöttle, and Jörg Kienzle. **Associations in MDE: A Concern-Oriented, Reusable Solution**. In *Proceedings of the 12th European Conference on Modelling Foundations and Applications (ECMFA 2016)*, pp. 121–137. Springer International Publishing, 2016.

- Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoît Combemale, Julien DeAntoni, Jacques Klein, and Bernhard Rumpe. **VCU: The Three Dimensions of Reuse**. In *Software Reuse: Bridging with Social-Awareness – 15th International Conference (ICSR 2016)*, pp. 122–137. Springer International Publishing, 2016.

- Wisam Abed, Matthias Schöttle, Abir Ayed, and Jörg Kienzle. **Concern-Oriented Behaviour Modelling with Sequence Diagrams and Protocol Models**. In *Revised Selected Papers of the International Workshops on Behavior Modeling – Foundations and Applications (BM-FA)*. Lecture Notes in Computer Science, vol. 6368, pp. 250–278. Springer-Verlag New York, 2015.

- Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty H. C. Cheng, Philippe Collet, Benoit Combemale, Robert B. France, Rogardt Heldal, James Hill, Jörg Kienzle, Matthias Schöttle, Friedrich Steimann, Dave Stikkolorum, and Jon Whittle. **The Relevance of Model-Driven Engineering Thirty Years from Now**. In *Proceedings of the 17th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*, pp.183–200. Springer International Publishing, 2014.

- Romain Alexandre, Cécile Camillieri, Mustafa Berk Duran, Aldo Navea Pina, Matthias Schöttle, Jörg Kienzle, and Gunter Mussbacher. **Support for Evaluation of Impact Models in Reuse Hierarchies with jUCMNav and TouchCORE**. In *Proceedings of the MODELS 2015 Demo and Poster Session co-located with ACM/IEEE 18th International Conference* *on Model Driven Engineering Languages and Systems (MODELS 2015)*, pp. 28–31. CEUR-WS.org, 2015.

- Nishanth Thimmegowda, Omar Alam, Matthias Schöttle, Wisam Al Abed, Thomas Di'Meco, Laura Martellotto, Gunter Mussbacher, and Jörg Kienzle. **Concern-Driven Software Development with jUCMNav and TouchRAM**. In *Proceedings of the Demonstrations Track of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*. CEUR-WS.org, 2014.

- Matthias Schöttle, Omar Alam, Gunter Mussbacher, and Jörg Kienzle. **Specification of Domain-specific Languages Based on Concern Interfaces**. In *Proceedings of the 13th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2014)*, pp. 23–28. ACM, 2014.

- Matthias Schöttle and Jörg Kienzle. **On the Challenges of Composing Multi-View Models**. In *Proceedings of the First Workshop On the Globalization of Modeling Languages (GEMOC 2013) co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*. 2013.

- Matthias Schöttle, Omar Alam, Abir Ayed, and Jörg Kienzle. **Concern-Oriented Software Design with TouchRAM**. In *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, pp. 51–55. CEUR-WS.org, 2013.

- Omar Alam, Matthias Schöttle, and Jörg Kienzle. **Revising the Comparison Criteria for Composition**. In *Proceedings of the Fourth International Comparing Modeling Approaches Workshop 2013 co-located with the ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*. CEUR-WS.org, 2013.

- Wisam Al Abed, Valentin Bonnet, Matthias Schöttle, Engin Yildirim, Omar Alam, and Jörg Kienzle. **TouchRAM: A Multitouch-Enabled Tool for Aspect-Oriented Software Design**. In *Proceedings of the 5th International Conference on Software Language Engineering (SLE 2012)*, pp. 275–285. Springer Berlin Heidelberg, 2012.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

List of Acronyms

AOM	Aspect-Oriented Modelling
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
API	Application Programming Interface
CI	Customization Interface
CORE	Concern-Oriented Reuse
DAG	Directed Acyclic Graph
EBNF	extended Backus-Naur form
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
FAQ	Frequently Asked Questions
GRL	Goal-Oriented Requirement Language
MDE	Model-Driven Engineering
MOF	Meta-Object Facility
OO	Object-Oriented
OOP	Object-Oriented Programming
RAM	Reusable Aspect Models
SoC	Separation of Concerns
SPL	Software Product Line
UI	Usage Interface
UML	Unified Modeling Language
VCU	Variation, Customization and Usage (Interfaces)
VI	Variation Interface

Part I

Prologue

1 Introduction

Methodical reuse of software artefacts is considered key to software engineering [65, 70]. Instead of creating all functionality from scratch, common and recurring functionality is reused. This functionality is typically packaged into frameworks or libraries with the purpose of making it reusable. They are usually well maintained, continuously improved, and come with good quality textual documentation and different forms of code examples. Many are publicly available, available as open source and have large communities. In addition, companies are interested in reusing existing software artefacts in order to amortize development costs by increasing quality, productivity, and time-to-market [66, 74].

Model-Driven Engineering (MDE) [21,32] 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 aims to reduce accidental complexity and helps dealing with the complexity of the problem to be solved. MDE improves development productivity as well as the quality of the end product [50, 75, 122].

Although MDE technology has been available for more than two decades, adoption of MDE in industry is slow and not widespread [49, 105, 124]. One of the reasons is that reuse at the modelling level is not common. We see this being caused by several factors. First, hardly any modelling language offers all the language features required to build reusable models, i.e., language constructs for expressing modularization, encapsulation and composition. As a result, reuse at a higher-level of abstraction, i.e., a reusable architecture or design, typically takes the form of a pattern or style, and not a reusable model with a clear reuse interface. Second, it is currently not well understood how reuse should integrate with the top-down philosophy of MDE, where first platform-independent "higher-level" models are being built and refined to platform-specific "lower-level" models, from which eventually executable code is generated. Reuse could happen within a level of abstraction to build complex reusable units by combining reusable units of lower complexity, for example, when building the design of a bank application a reusable unit providing the design for authentication functionality could be reused. Also, reuse could happen across levels of abstraction, for example, when reusing within a design a programming language framework.

Due to the above, model libraries or repositories for reusable models are very uncommon. Models are either designed from scratch or bits and pieces are copy-and-pasted from other models. While some model repositories exist [22, 117], they mainly focus on collecting example models or do not have a large community behind them.

At the implementation level, however, modern programming languages already provide *out-of-the-box* a large number of classes for different functionality. On top of that, reusable code artefacts in the form of libraries and frameworks are abundant, well maintained and widely and readily available to developers. Availability of frameworks and libraries has risen dramatically since open-source software adoption has increased [107]. There exist large repositories of reusable code artefacts. For instance, the *Central Maven Repository*[1], which is primarily used for Java in combination with the *Maven* build system, contains more than 290,000 unique artefacts[2]. *npm* is a package manager for JavaScript, its package registry contains more than 1 million unique packages[3].

However, reuse at the implementation level has problems as well. Most frameworks come as a monolithic code block. Developers need to find out and understand how to adapt the reusable entities to their own needs, and then how to use the usage interface (API) correctly. While frameworks generally provide many different artefacts that explain the framework along with the code itself, such as documentation, API reference, tutorials, *How Tos*, demos, etc., the manner in which this information is presented is typically unstructured, scattered and informal. Furthermore, documentation can be ambiguous or incomplete [120]. Many frameworks come with a large feature set, and developers spend a lot of time understanding how to use them properly. While tutorials show how to use them in a general way, it is not always clear how to correctly reuse the knowledge in the tutorial for a specific development project. Also, even though a developer might only want to use a small feature set of a framework, he is nevertheless confronted with the complete API. This can lead to incorrect reuse, which can have serious consequences, e.g., in security-critical cases can expose vulnerabilities [45]. Reuse at the implementation level by means of frameworks also hinders separation of concerns, as the business and application logic and the glue code for the reused artefact are often tangled. Lastly, evolution of frameworks is difficult, as either binary compatibility needs to be maintained, or the framework users are required to update all glue code. If the required effort is too high, developers do not update to newer versions [89].

[1]https://search.maven.org
[2]https://search.maven.org/stats
[3]http://www.modulecounts.com

1.1 Problem Statement

There clearly exists a gap between the modelling and the programming worlds. While at the modelling level reuse is not common, the higher level of abstraction allows developers to more easily deal with the complexity of the problem domain and reason about it. At the implementation level, there exist many high quality class libraries and frameworks ready to be reused at the code level. In standard MDE it would require considerable time and effort to make the same functionality available to the modeller. In addition, re-creating equivalent functionality at the model level is error prone. Since the classes and methods of a framework are already implemented in code, they should therefore not be re-defined at the modelling level, but their functionality should nevertheless be accessible from within design models. At the same time, the higher level of abstraction should be used to formalize information of reusable artefacts to make it easier for users to reuse.

A step towards improved reuse is provided by the Concern-Oriented Reuse (CORE) approach, a next-generation reuse technology [4]. Its main unit of abstraction is the *concern*. A concern provides a three-part interface to document the functional variations, how to customize the chosen functionality of the concern to the reuse context, and how to use the provided functionality of the concern. CORE makes it possible to build reusable models and collect them in a reusable model library. However, the library of reusable concerns is currently small, making it difficult to create software systems without re-creating a large part of functionality, which would normally be accomplished by reusing existing frameworks.

Therefore, this thesis attempts to answer the following research question:

> *How can existing functionality available at the implementation level be reused at the modelling level with minimal effort and the higher level of abstraction be exploited to benefit reuse?*

1.2 Thesis Contributions

This thesis provides a solution to bridge the gap between the modelling and implementation level by providing a connection between the design at the modelling level and the code at the implementation level.

We propose *Concernification* as a new approach that raises the level of abstraction of reusable code artifacts (frameworks) to the modelling level. The benefits are two-fold. First, existing functionality that is provided through frameworks and libraries and can already be reused at the implementation level can now be reused at the modelling level as well. Second, reuse of frameworks is simplified, as the concern interface presents a high-level, formal, organized view of the user features that a framework provides to the user, and is modularized in such a way that it can expose only the subset of the framework's API that a user needs. Modularizing the API exposed the need for an incremental refinement of interfaces across several variants. This thesis also proposes *Signature Extension* to deal the finality of method signatures present in Object-Orientation.

Concretely, the thesis makes the following contributions which are organized into two parts:

Part I Concernification:

- A new approach called *Concernification* which applies the principles of Concern-Oriented Reuse (CORE) to framework reuse. *Concernification* raises the level of abstraction of reusable code artefacts (frameworks) to the modelling level. The concern interface formalizes and documents what the framework provides to the user. It provides a high-level, formal, organized view of the user features that a framework provides, and only exposes a subset of the framework's API tailored to the user's needs. By explicitly modelling the three interfaces (variation, customization, and usage) of a concern a framework is formalized from the user's perspective. This provides several benefits to the user of a framework, such as documenting the variations, guiding the user on making a choice based on the impact on high-level goals, providing a tailored API, documenting the generic elements that need to be customized, expressing the usage protocol of the API, and providing "glue code" that is always required.
  - A definition of the steps necessary to successfully create a concern interface of an existing framework.
  - A demonstration of how the concern interface for an existing framework is determined and created. This is illustrated using an existing framework called *Minueto*. We created this concern interface of *Minueto* by familiarizing ourselves with the API and the various available resources that are provided along with the framework.
  - A qualitative study with the two developers of *Minueto*. The developers are the domain experts of the framework. This study presents validation of the accuracy of our *Minueto* concern interface. We also report on insights gained from interviewing the two developers in relation to concernification and variations of feature models.
- A complete description of an algorithm to automatically create an initial concern interface of a framework. The algorithm discovers the features of the framework, organizes them in a feature model and modularizes the API in accordance with the features. The qualitative study with the Minueto developers confirmed our intuition of which information from a framework and its code documentation to use to determine the concern interface. The *automated concernification* algorithm uses structural relationships of the API and the usage of the API from examples provided with a framework to determine an initial concern interface.

  - An implementation of the *automated concernification* algorithm using the backend components of the TouchCORE tool. To facilitate fully the concernification of existing code, the TouchCORE tool is extended with support for importing existing code as implementation classes into a design model, generating code from design models, and traceability to understand which feature an API element belongs to in a composed model.
  - A validation of the *automated concernification* algorithm on three frameworks. Specifically, *Workflow* was designed as a concern first which means that the feature model with the features and their corresponding API is known. The *Minueto* concern interface was validated with the developers and therefore allows comparison to the automatically determined concern interface. To evaluate whether the algorithm provides accurate results on a third and bigger framework that is used in industry, it is performed on the notifications part of the Android platform. The results are validated using a qualitative study in which Android app developers familiar with the notifications API of Android provided feedback.

Part II: Signature Extension

- A new approach called *Signature Extension* which supports the incremental refinement of interfaces across several variants. It helps to overcome the rigid nature of signatures by allowing declared signatures to be extended with additional parameters.
  - An identification and description of four difficult situations that are caused by the finality of method signature declarations.
  - An empirical study on the Java Platform API that highlights the existence of these difficult situations in the Java API. In addition, the thesis outlines various workarounds that address these difficult situations proposed for different programming languages.
  - A novel way of extending method signatures called *Signature Extension* allowing method signatures to evolve across different variations. The *Signature Extension* approach makes it possible to extend method signatures structurally and behaviourally.
  - An extension of class diagrams as defined in the Reusable Aspect Models (RAM) language in CORE to support the structural extension of method signatures. This includes an extension to the class diagram composition to deal with evolving signatures.
  - An application of the *Signature Extension* approach to two reusable concerns.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides relevant background in the context of Model-Driven Engineering (MDE), Concern-Oriented Reuse (CORE) and Reusable Aspect Models (RAM).

The first part of this thesis presents *Concernification* that makes it possible to raise the abstraction level of frameworks to the modelling level. Chapter 3 introduces the idea of concernification, its benefits and steps. Furthermore, *Concernification* of a framework is shown based on an example of the *Minueto* framework. Chapter 4 then reports on a qualitative study with the two developers of *Minueto* to validate the concern interface determined in Chapter 3. To automate the process of concernification, Chapter 5 presents an algorithm to automatically concernify the API of a framework based on the structural relationships of the API and the usage of the API in examples. Chapter 6 evaluates the accuracy of the *Automated Concernification* algorithm using three frameworks. Chapter 7 presents an overview of existing work related to *Concernification* as well as *Automated Concernification*.

The second part of this thesis presents *Signature Extension* that enables the fine-grained abstraction of method signatures at the modelling level. Chapter 8 identifies the need for *Signature Extension* by discussing four difficult situations that are caused by the finality of signatures once declared. In addition, an empirical study is discussed that shows evidence of difficult situations in the Java Platform API, and workarounds in programming languages to overcome some difficulties are described. Chapter 9 presents the *Signature Extension* approach as well as the extension of class diagrams with support for structural signature extension. Chapter 10 presents evidence that *Signature Extension* overcomes the four identified difficulties by re-designing two reusable concerns.

Finally, Chapter 11 summarizes this thesis and discusses potential avenues for future work.

2 Background

This chapter introduces the methodologies, concepts and approaches that this thesis is based on and that are relevant for the understanding of this thesis.

We introduce the idea of *Model-Driven Engineering* in Section 2.1. We then discuss in Section 2.2 *metamodelling*, an essential process in building modelling languages and tools. In Section 2.3, we describe *Separation of Concerns* in software development in general, as well as advanced separation of concern techniques in the form of *aspect-orientation*. This thesis is in the context of *Concern-Oriented Reuse* (CORE), which we introduce in detail in Section 2.4. Finally, in Section 2.5, we describe *Reusable Aspect Models* (RAM) as one modelling language that is integrated into CORE. Both CORE and RAM are explained using a running example to showcase the various models that are involved.

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) [32] advocates the use of models as the primary artefact at every stage of the development lifecycle. It attempts to provide a way to deal with the increasing complexity of software development and promises to improve productivity and software quality. Models are used as an abstraction of the system under development.

MDE advocates the use of different modelling formalisms, so that the most appropriate modelling notation is chosen to reason about the system under development depending on the needs and level of abstraction. To move towards an executable implementation, high-level, problem-centric models are gradually refined or transformed to progressively integrate solution and platform details.

Reasoning at a higher level of abstraction also allows the inclusion of different stakeholders to participate and gather early feedback. This facilitates the detection of errors early in the lifecycle. Furthermore, the use of domain-specific notations allows domain expert themselves to work on the solution of the problem instead of using experts that are knowledgeable in the solution domain.

While various forms of modelling have long been used, such as sketches, informal diagrams, etc., these are often used in a *model-based* sense [21], i.e., as a form of documentation or communication of ideas, but not (or only indirectly) connected to the source code of the system that is being built. Such a situation occurs, for example, when a class diagram defining the domain model of a system is being used as a blueprint for developers when they start their implementation effort. A *model-driven* process, however, can use models as an input for a model-to-text transformation to produce the implementation (or parts thereof) in an automated way. This is also known as code generation. To support the use of precise models during the development process, MDE tools are usually built using metamodels which define the valid concepts and structure that models are created with. This allows one to write tools and transformations that can support any model built using the supported metamodel.

2.2 Metamodelling

Metamodelling [21, 108] is the process of defining a formal definition of the syntax for modelling languages. A metamodel describes the structure a model must adhere to, i.e., the abstract syntax of a modelling language. Models can then be created that conform to the metamodel. This means that the metamodel is an abstraction of the model. The elements of the model are instances of elements in the metamodel.

Going further, the metamodel is defined by the meta-metamodel. Usually, the meta-metamodel can be described in itself and as such, there is no further need for defining languages at even higher levels of abstraction. Essentially, this culminates in a four layer approach, where each layer is usually numbered from 0 to 3 (or vice-versa). Going bottom-up, the real-world subject is represented by the *instance model*. Usually, the instance model is an abstraction of the real world subject, because it describes only the properties of interest. The instance model is an instance of *model*, which is an instance of the *metamodel*. The metamodel in turn is an instance of the *meta-metamodel*.

Figure 2.1 shows the four layers of metamodelling on the left-hand side, and the following example on the right-hand side. In this example, an instance model describes the book *1984* by *George Orwell*. The model describes books in general, i.e., that they have a title and author. The metamodel that the *Book Model* is described with can be a UML class diagram [81] (i.e., metamodel), which is based on the *Meta-Object Facility* (MOF) [80] (i.e., the meta-metamodel). The core part of MOF is defined in *Essential MOF* (EMOF). Its de facto reference implementation is *Ecore*, which is provided by the *Eclipse Modeling Framework* (EMF) [109]. A similarity of the four layer approach can be drawn to grammars in that a grammar (metamodel) is defined using EBNF (i.e., meta-metamodel).

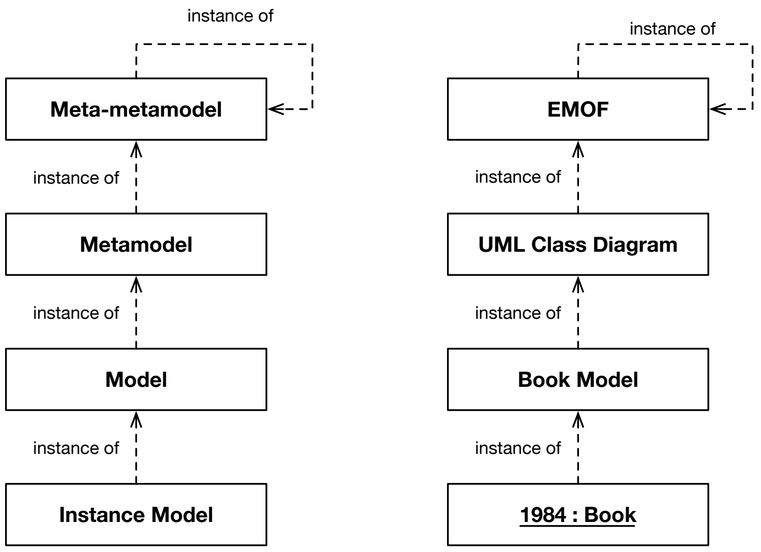


Figure 2.1: Layers of Metamodelling

2.3 Separation of Concerns

Separation of concerns (SoC) [28, 29] is a key design principle in software development where a program is separated into distinct parts with as little overlap as possible. A concern is any piece of interest and can be seen as a feature of a program. SoC can be achieved by striving for encapsulation (using information hiding) and modularity [20, 85]. This helps reduce the complexity of software construction as well as software maintainability, and can enable the reuse of functionality.

Object-Orientation, for example, provides a way to decompose a system into units of primary functionality called *modules* (classes). The ability to distinguish between public (accessible from the outside) and private members of a class provides encapsulation and hides the internals of a class to the outside. From the outside, the interface (API) provides the contract to the outside world. However, this only provides decomposition along a single dimension (objects) [111] and does not allow the separation of cross-cutting concerns. Certain concerns of a system, such as logging, caching, authentication, etc., cross-cut the entire system. Their code is scattered across all affected modules of the system. This results in code tangling, i.e., the code of cross-cutting concerns is intermixed with business logic.

2.3.1 Aspect-Orientation

As a result of dealing with cross-cutting concerns, advanced separation of concern techniques have been proposed. These techniques allow the separation of concerns along additional dimensions. A well-known methodology is *Aspect-Oriented Software Development* (AOSD) [41], which addresses the identification, specification and separate expression of cross-cutting concerns in the software development lifecycle.

Cross-cutting concerns can be expressed separately, and it can be specified where in the system they should be applied. In AOSD, concerns are usually called *aspects*. To retrieve the final system, all aspects are combined by composition at the specified places.

While aspects can be applied at different levels of abstraction, they evolved from first being introduced at the implementation level, known as *Aspect-Oriented Programming* (AOP) [59]. *AspectJ* [58] is the most widely-known language providing aspects for Java, however, most aspect-oriented programming languages share common concepts on how aspects are defined and applied.

So called *join points* define, for a given language, where in a program an aspect can be applied. In AspectJ, for example, join points include when a method is executed or called, a field is read or modified, an object is instantiated, etc. A developer can use *pointcuts* to designate the concrete join points of a program where the aspect should be applied. The *advice* defines the structure and/or behaviour that should be added at a matched pointcut. Most commonly, it is possible to define whether additional behaviour should be added *before*, *after*, or *around* the identified *join point*.

2.3.2 Aspect-Oriented Modelling

Applying aspect-oriented principles only at the source code level creates a gap between the earlier phases of the software development process and the implementation phase. Thus, *Aspect-Oriented Modelling* (AOM) emerged enabling aspect-oriented techniques to be applied at higher levels of abstraction [96]. Aspects play a role within the complete software development process. This makes it possible to bridge the gap between different phases and facilitates traceability of concerns across the software development lifecycle.

AOM techniques use different kinds of patterns (a pointcut is one of them) and different ways of composition. They can generally be divided into *symmetric* and *asymmetric* approaches. Asymmetric approaches make a distinction between cross-cutting concerns and the base of the application. Symmetric approaches, however, do not have this distinction and the whole system is composed from aspects.

2.4 Concern-Oriented Reuse (CORE)

Concern-Oriented Reuse (CORE) [4] is a next-generation reuse approach inspired by the ideas of multi-dimensional separation of concerns [111]. CORE builds on the ideas of MDE, software product lines (SPL) [86], goal modelling [53], and advanced modularization techniques offered by aspect-orientation. CORE seeks to address the challenge of how to enable broad-scale, model-based software reuse [65, 70].

In contrast to the focus of classic MDE on models, the main unit of abstraction, construction, and reasoning in CORE is the *concern*. A concern is a unit of reuse that groups together software artefacts (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. It contains all models required for the domain of interest and the models may span all phases of the software development lifecycle, i.e., requirements, analysis, design, testing and implementation models. A concern in CORE is broader and usually provides different variants of the functionality it encapsulates. CORE advocates developing an application by reusing as many already existing concerns as possible. In order to realize the functionality of a concern, a concern itself can reuse the functionality of lower-level concerns. This creates a concern hierarchy.

In the context of reuse, at least two clearly distinct software development roles arise. The *designer* of the reusable unit is an expert of the domain of the development issue that the unit addresses. She has a deep understanding of the nature of the issue, is able to identify variations of the problem and can therefore potentially identify user-relevant variations or *features*. Because the designer elaborates and implements the solution artifacts, she knows the exact implementation details, their properties and qualities, and the trade-offs that she decided to make. However, the designer does not know in what contexts and how exactly the reusable unit may be used in the future. Therefore, in addition to realizing the solutions, the designer strives to make the reusable unit as versatile and generic as possible, so that the solutions can be applied in a wide variety of reuse contexts. This might again involve coming up with multiple, functionally equivalent, yet different variants of realizations in terms of qualities and non-functional properties, e.g., varying memory footprint or performance. Finally, the designer needs to modularize and package the reusable unit to make it available to others, e.g., in form of a library or framework. There is no doubt that building a reusable unit is a challenging, non-trivial, time consuming task for the *designer*.

A *user* of a reusable unit on the other hand is an expert of the application he is developing. He is aware of the specific requirements of the system he is working on. At some point, the user might become aware that the software needs to deal with a specific development issue for which an existing reusable unit is available. The user typically knows very little about the complexity of the recurring development issue, and even less about the implementation details of different solutions to the issue offered by the reusable unit. To make reuse possible and safe, the user needs to be able to determine whether the reusable unit is applicable to their system. He has to be able to determine which solution, in case the reusable unit offers more than one, is most appropriate for the specific application context. He needs to be able to customize the reusable unit to his specific reuse context, and then must use the reusable unit correctly.

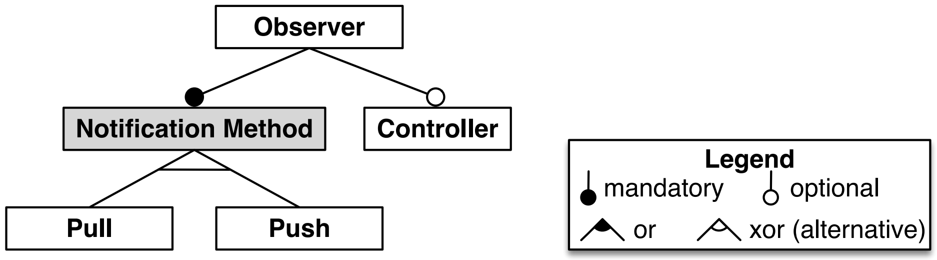


Figure 2.2: The Feature Model of the *Observer* Concern.

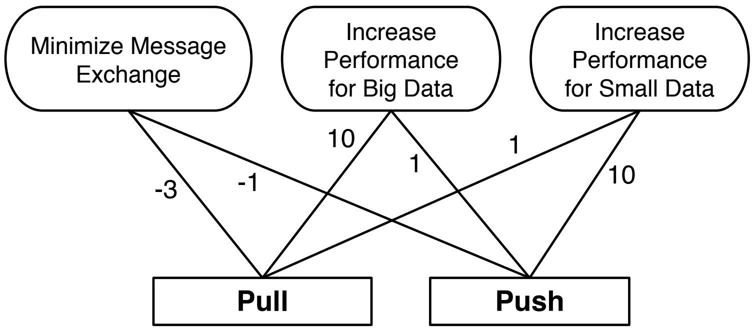

Minimize Message Exchange
Increase Performance for Big Data
Increase Performance for Small Data
-3
-1
10
1
1
10
Pull
Push

Figure 2.3: The Impact Model of the *Observer* Concern.

2.4.1 Three Interfaces of Reuse

In order to facilitate reuse of artefacts, a concern provides a three-part interface [63]. The *variation interface*, *customization interface*, and *usage interface* formalize the boundaries between concerns. They contain all the information a concern user needs to know in order to reuse a concern created by a concern designer.

2.4.1.1 Variation Interface

The *variation interface* (VI) describes the required design decisions and their impact on high-level system qualities, both explicitly expressed using feature and impact models in the concern specification. The feature model [57] describes the available functional variants and design choices the concern offers. The features are structured in a tree with non-root features having a relationship to their parent (*mandatory*, *optional*, *XOR* or *OR*). Additionally, cross-tree constraints (*requires* or *excludes*) support the definition of additional relationships. For example, Figure 2.2 shows a concern design of the *Observer* design pattern [44]. It has two alternative variations (XOR) for *notification methods*, *push-based* or *pull-based*.

The impact model is expressed using a variant of the *Goal-Oriented Requirement Language* (GRL) [53]. It captures the high-level goals and non-functional properties and describes how the features of the feature model impact these goals. The weights of the contributions of the features are specified using relative values [34]. The concern designer as the domain expert knows the details of the implementation and establishes these values. For some high-level goals, such as performance, measurements or benchmarks can be performed to determine them. For example, as Figure 2.3 shows, the push and pull-based notification methods of the *Observer* concern impact the high-level goals *Minimize Message Exchange* and *Increase Performance*. While a pull-based notification method minimizes the message exchange, the push-based one increases the performance for small data.

2.4.1.2 Customization Interface

The *customization interface* (CI) allows the chosen variation of a concern to be adapted to a specific reuse context. In order to facilitate reuse of concerns, the concern designer cannot fully specify everything of the concern as it depends on the context in which it is reused. Therefore, such elements in the models can only be specified partially. They are placeholders and need to be completed by the concern user. Essentially, the customization interface shows to the concern user the elements that need to be customized. For example, to use the *Observer* concern in the design, it provides partial classes for the *subject* and *observer*. A concern user needs to specify which elements in the application should play these roles. In essence, the *Observer* concern provides the required structure and behaviour for the functionality to observe subjects and receive notifications when they are modified.

2.4.1.3 Usage Interface

The *usage interface* (UI) defines how the functionality encapsulated by a concern may be used. In order to ensure information hiding, the details of the design are encapsulated from the outside. The UI specifies which model elements are visible and accessible from the outside to trigger the functionality provided by the concern. In this way, the usage interface is the API of the concern. For example, a design model's usage interface is typically comprised of all the public classes and methods. In the *Observer* concern, the usage interface might include the methods to start and stop observing a subject that can be called on the observer.

2.4.2 Reuse Process

As outlined above, building a concern is a non-trivial, time-consuming task, typically done by or in consultation with a domain expert. The initial effort to create a concern that is reusable is high. 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. The concern user may do trade-off analysis between the different possible configurations to find the solution that best satisfies the relevant goals.

2. Adapting the general models of features of the concern that were selected to the specific application context based on the customization interface. The partial elements of the chosen variant must be mapped to the concrete elements of the application.

3. Using the functionality provided by the selected concern features within the application as defined in the usage interface.

CORE advocates developing a concern by reusing as many existing concerns as possible. That way, a concern designer often also plays the role of a concern user for another concern. To that effect, CORE provides a reusable concern library to which concern designers can contribute their reusable concerns. Concern users can then reuse these concerns for their purposes. Ideally, an open-source community will emerge in which concerns are continuously refined based on feedback and contributions in a similar way that can be observed in the open-source software world.

2.4.3 Composition

In CORE, each feature is realized by one or more models. Each realization model provides the realization of that feature for one of the views onto the concern. For example, design models provide the detailed design in realizing features using class and sequence diagrams to model the structure and behaviour. In general, CORE advocates an additive approach, i.e., realization models of children features build upon the realization model of their parent feature. Therefore, the child realization model extends the parent realization model.

When a concern user chooses a set of features of a concern, all their realization models (and their ancestors) are pair-wise composed together. This is performed for each language using the homogenous composition operator of the language. The composition algorithm also maintains tracing information, which keeps track of which elements of the composed model belong to which feature, which can help the user in understanding the composed model.

2.4.4 Delaying Decisions

Software product line development (SPL) [86] is an approach that is beneficial when developing a collection of similar software systems—a family of products—that share some commonalities and differ in a well-defined set of features exposed by the SPL. Reuse [65, 70] in the context of SPL is therefore planned in advance. When the needs of a user are clear, the set of features corresponding to the needs of the user is selected and a specific application—called product in SPL terms—is derived.

Choosing the best variant works in the context of SPL, as at the moment of product derivation all requirements are known. This is different in the case of concerns where the functionality that is being designed will be reused itself. As such, a concern designer reusing another concern does not know the final requirements of the application and can therefore not always make all decisions in advance, as it depends on the context in which it is being reused.

As such, CORE has been designed in such a way that decisions can be delayed as long as possible [64]. Only those features that are known to be required need to be selected. Features that could be useful, but a decision can not be made at the time of initial reuse, can be *re-exposed*. A partial configuration contains feature selections and features that have been re-exposed. CORE can handle partial configurations, i.e., CORE will allow the concern designer that is reusing some other concern to customize and use the functionality provided by the selected features of the reused concern, while leaving additional decisions up to the concern user of the concern that is being designed. Those users can then make an additional decision if additional requirements are known or choose to further delay the decision(s), if needed.

For example, a concern reusing the Observer concern outlined in Section 2.4.1 might not be able to make the decision between a *pull-based* or *push-based* observer as this affects the performance and therefore chooses to delay this decision.

2.4.5 CORE Metamodel

To provide a thorough understanding of CORE, we present here the CORE metamodel [100] highlighting relevant parts pertinent to this thesis. The full metamodel is shown in appendix A.

CORE provides a common framework for concerns and reuse using the three interfaces (see Section 2.4.1). It has *extension points* in the form of abstract classes[1] where language designers can plug in to *corify*[2] their modelling language. To this aim, the modelling language metamodel is required to sub-class the abstract classes of the CORE metamodel. This is either accomplished by adding the `COREModelElement` class as a super-class of an existing class or by introducing a new sub-class in the language metamodel, which is then associated with existing classes of the language. To avoid name conflicts between metamodels, all classes in CORE are prefixed with "CORE".

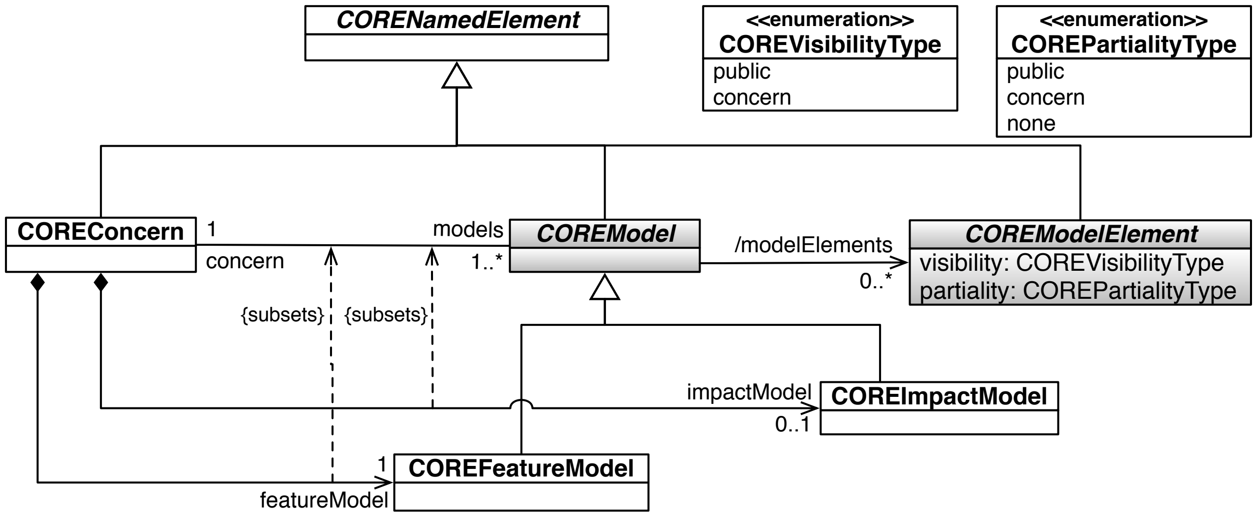


Figure 2.4: General Overview of a Concern in the CORE Metamodel.

2.4.5.1 Concern and Models

Figure 2.4 shows the main concepts of the CORE metamodel. At the centre is the `COREConcern`, which groups together `COREModels`. `COREFeatureModel` and `COREImpactModel` are concrete models contained in a concern, because they define the variation interface. The `COREModel` class may be sub-classed in order to *corify* a language. The language must provide `COREModel-Elements`. The only requirement CORE imposes on model elements is that they provide a partiality (to denote generic, incomplete elements) and visibility (to denote elements that are accessible to the outside). These are required to be able to define the customization and usage interfaces.

The `COREFeatureModel` is highlighted in Figure 2.5. It contains all the `COREFeatures` that are part of the feature tree of a concern. With the exception of the root feature, all other features have a concrete `COREFeatureRelationshipType` (*XOR*, *OR*, *mandatory*, or *optional*) denoting the relationship to their parents. In the case of *OR/XOR*, all siblings share the same relationship type. The `requires` and `excludes` associations of `COREFeature` realize the cross-tree constraints (*requires* or *excludes*).

Figure 2.6 provides an overview of the impact model. The `COREImpactModel` contains `COREImpactNodes`, which represent the main impacts (high-level goals). The `COREContri-bution` is the relationship between impact nodes denoting how the incoming impact node con-

[1]Denoted in grey in the figures to follow.

[2]The process of sub-classing the CORE metamodel within a language to be able to use it within the CORE approach.

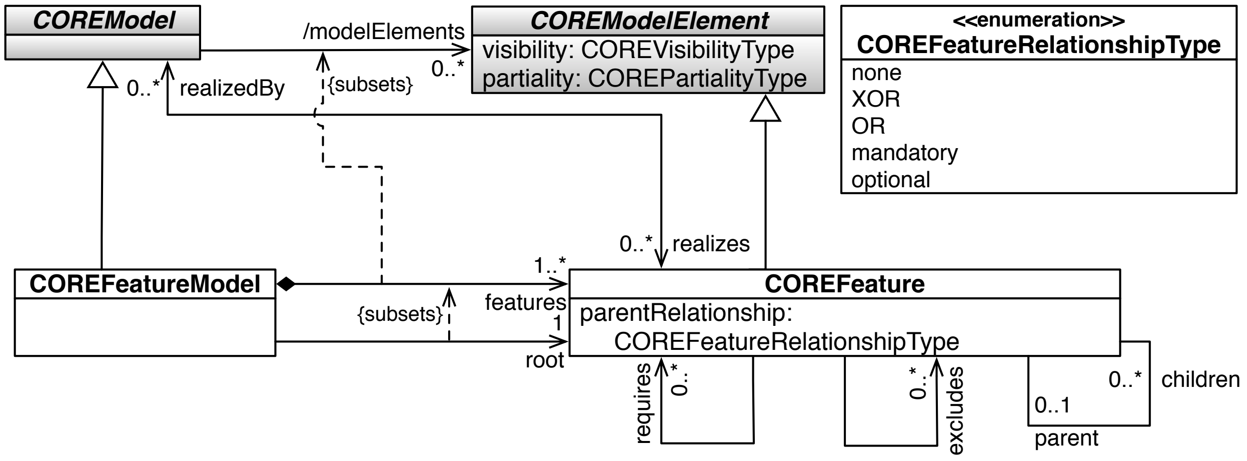


Figure 2.5: The Feature Model part of the CORE Metamodel.

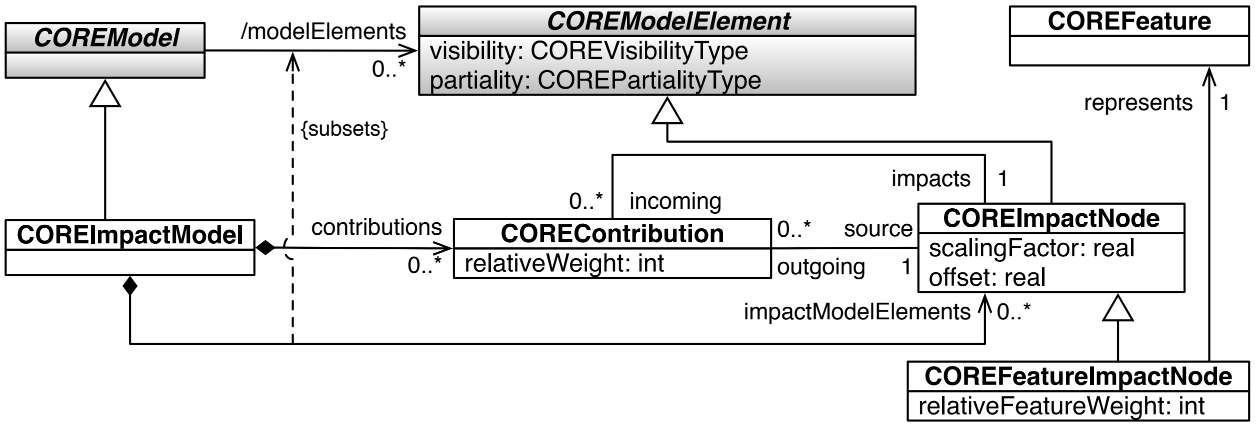


Figure 2.6: The Impact Model part of the CORE Metamodel.

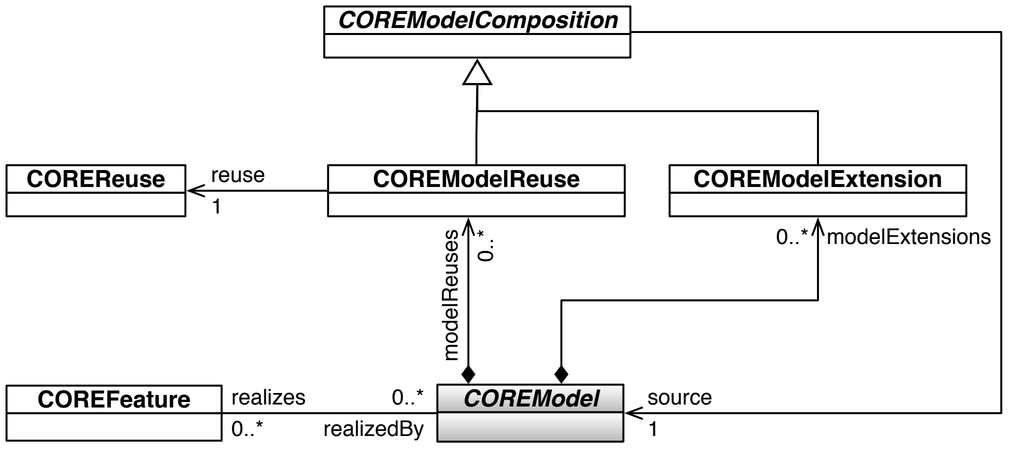


Figure 2.7: The Realization Models and their Compositions in the CORE Metamodel.

tributes to the target impact node. Goal models are rarely used in isolation, and in CORE they are used in combination with feature models. Features that are selected from the feature model have an impact on the high-level goals. Therefore, the `COREFeatureImpactNode` is the impact node representing a `COREFeature`. A simple impact model has one `COREImpactNode` node as the high-level goal, and two or more feature impact nodes contributing to the high-level goal.

2.4.5.2 Feature Realizations and Model Compositions

As described in Section 2.4.3, Figure 2.7 shows that each feature can be realized by models. If a feature is not realized by any models, it is usually used as a grouping feature, e.g., to group together alternative features. A feature is realized by one model of each modelling language. However, there exist feature interactions where the structural or behavioural realizations need to be adapted if the realizations of two or more features conflict with each other. These conflicts are resolved by providing realization models that realize more than one feature. During the reuse process (see Section 2.4.2), when the conflicting features are selected, the conflict resolution model is used during the composition instead of the individual realization models.

In order to support composition, there exist two `COREModelCompositions` (see Figure 2.7). The intra-concern composition—for extending the realization model of the parent feature—is accomplished using the `COREModelExtension` class. Inter-concern composition is done through the `COREReuse`. Each model contains its own model composition for a reuse in the form of the `COREModelReuse`.

Figure 2.8 highlights the reuse and the feature configurations. Reuses are specified at a concern-wide level. A `COREConfiguration` entails the feature selection and re-exposed features (see Section 2.4.4) for a reuse. In addition, models within the same concern hierarchy, i.e., realiza-

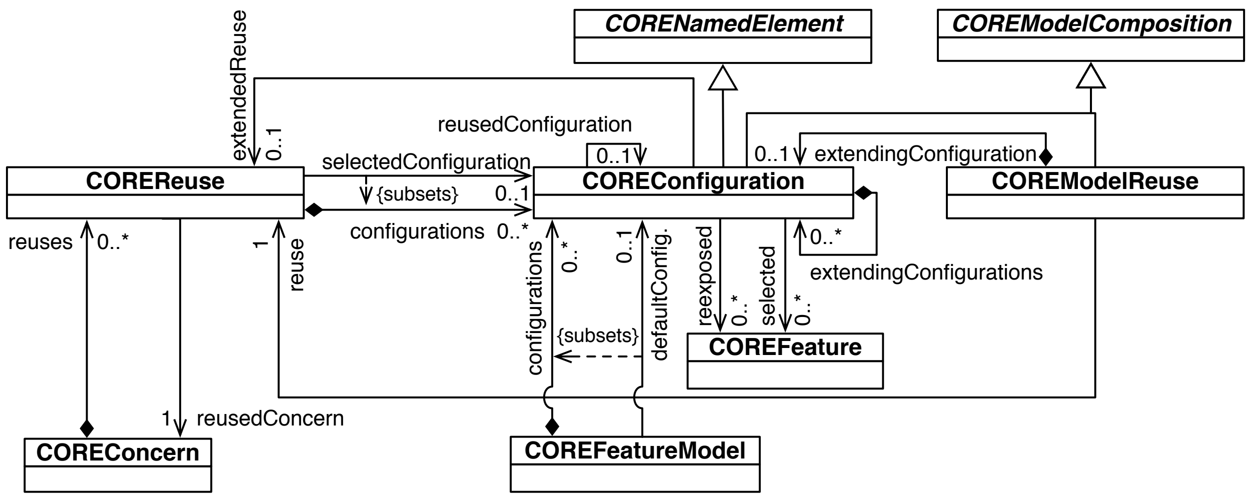


Figure 2.8: The Reuse part of the CORE Metamodel.

tion models that extend another higher-level realization model of the concern, can refine an existing reuse (see `extendingConfiguration` association in Figure 2.8). For example, it might be that to realize a sub-feature, a feature that was re-exposed earlier needs to be explicitly chosen. Similarly, delaying of decisions across concerns is supported via the `extendedReuse` and `reusedConfiguration` associations, as shown in Figure 2.8. Furthermore, a concern can provide pre-defined configurations that are contained in the `COREFeatureModel`. This allows the concern designer to provide different selections for various scenarios and gives the concern user a convenient way to make a decision.

Finally, model element compositions must be specified to support the customization interface. `COREModelElements` that are concretized during reuse and any elements that are refined within the same concern hierarchy are specified using `COREModelElementCompositions`, as shown in Figure 2.9. Depending on the modelling language this can either be provided through a `COREPattern`, describing a way to match elements of the source model of the composition, or defined through a concrete mapping from the source model element to the target model element (in the model the `COREModelComposition` is contained in). Figure 2.9 also provides an example of a concrete `CORELink`. Impacts from a reused concern might relate to high-level goals of the reusing concern. Therefore, impact models provide the `COREWeightedLink` to bind the high-level goal of the reused concern with the affected goal of the reusing concern [5, 34].

2.4.5.3 Model Element Cardinalities

Furthermore, `COREMappingCardinality` provides ways for the concern designer to provide cardinalities (multiplicities) to model elements within a realization model [17]. The cardinality

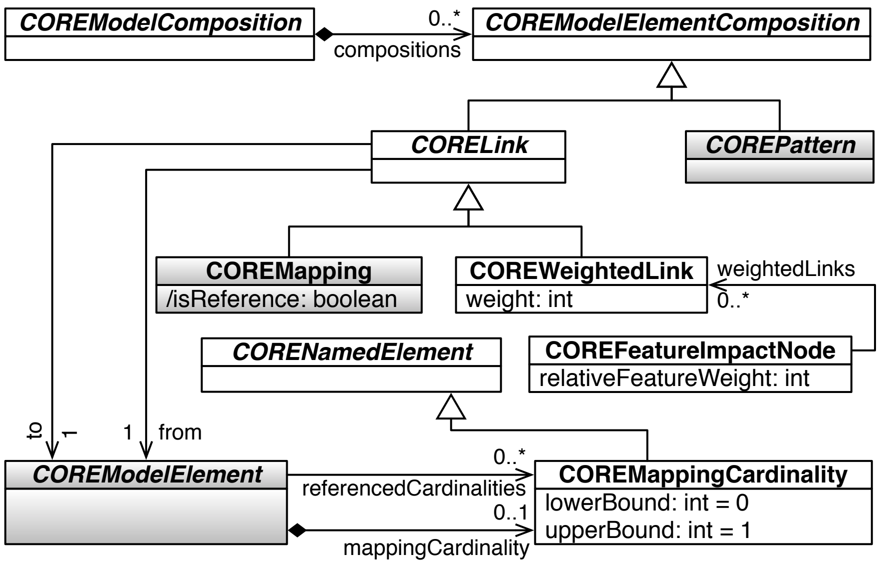


Figure 2.9: The Model Element Compositions part of the CORE Metamodel.

specifies how often the element can be mapped within the same composition. This is especially important if there are elements that need to be mapped a certain number of times for every mapping of another element. For example, each observer in the *Observer* requires that its `update` method is mapped as many times as the `modify` method of the subject.

2.5 Reusable Aspect Models

CORE provides a framework for reuse using the VCU interfaces. As we described in the previous section, modelling languages can extend the CORE metamodel to use them within the CORE approach. One such modelling language is *Reusable Aspect Models* (RAM) [60, 61].

RAM is an aspect-oriented multi-view modelling approach for software design modelling. It offers detailed design using three modelling notations based on UML class, sequence and state diagrams [81]. A RAM model supports structural modelling using class diagrams (*structural view*), behavioural modelling with sequence diagrams (*message view*), and protocol modelling with state diagrams (*state views*). The structural view and message views describe the object-oriented design and their behaviour, while the state views describe the operation invocation protocol for class instances. RAM is a compositional approach using aspect-oriented techniques allowing models to build on each other, which means that models can be kept at reasonable sizes. This results in decreased complexity of a single model.

RAM has been *corified* to provide design models for CORE [4]. Therefore, its metamodel

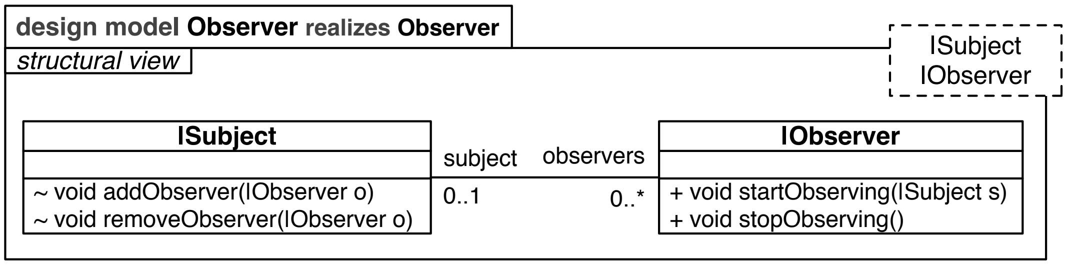

design model Observer realizes Observer
structural view
ISubject
IObserver
ISubject
~ void addObserver(IObserver o)
~ void removeObserver(IObserver o)
subject
observers
0..1
0..*
IObserver
+ void startObserving(ISubject s)
+ void stopObserving()

Figure 2.10: The *Structural View* for the *Observer* Feature of the *Observer* Concern.

extends the CORE metamodel as advocated by CORE (see Section 2.4.5). For instance, the root element of RAM—the `Aspect`—sub-classes the `COREModel`. Model elements such as `Class`, `Operation`, `Attribute`, etc. sub-class `COREModelElement`. Specific mappings are provided within RAM already, therefore `ClassifierMapping`, `OperationMapping` etc. sub-class `COREMapping`.

2.5.1 Structural View

The structure of aspects is defined in a structural view which defines all classes together with their attributes and operations, as well as any associations among classes. The structural view is based on UML class diagrams, with the additional possibility of marking classes, attributes and operations as partial by prefixing their name with a vertical bar: '|'.

Partial model elements are included in the concern's customization interface, and designate model elements that are general from the point of view of the current concern, which means that they must be mapped to application-specific model elements before the concern can be used. Furthermore, *concern-partial* denotes an element that must be mapped within the same concern by a model lower in the realization hierarchy.

Similarly, public classes and operations are part of the usage interface. They are the model elements that another model can use and invoke. Besides the common visibility modifiers (*public* (+), *private* (−) and *protected* (#)), the *concern-private* (~) visibility[3] restricts access to model elements within the same concern.

For the *Observer* concern shown in Section 2.4.1, the structure of the design realization model of the base feature *Observer* is shown in Figure 2.10. Figure 2.11 shows the structural extension of the base design for the *Push* feature.

[3]This is similar to the *package* visibility in OOP.

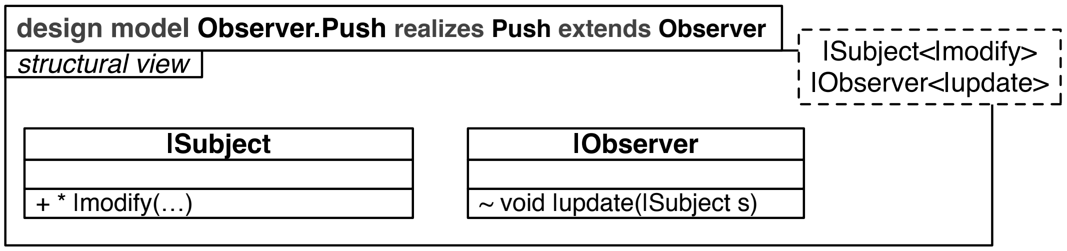

design model Observer.Push realizes Push extends Observer
structural view
ISubject<Imodify>
IObserver<Iupdate>
ISubject
+ * Imodify(...)
IObserver
~ void Iupdate(ISubject s)

Figure 2.11: The *Structural View* for the *Push* Feature of the Observer Concern.

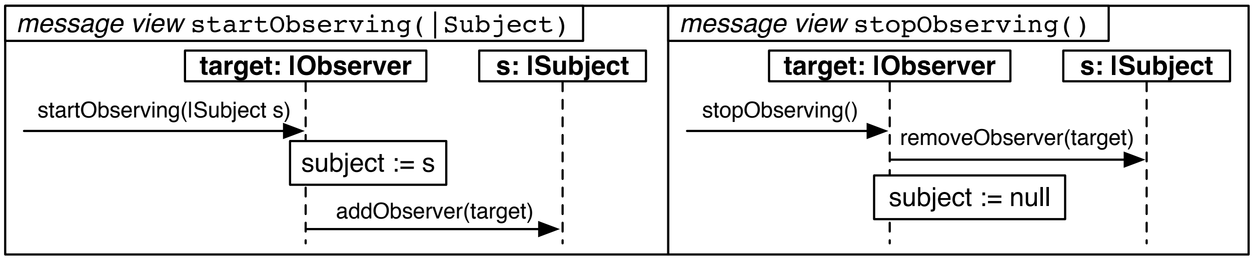


Figure 2.12: The *Message Views* for the *Observer* Feature of the *Observer* Concern.

2.5.2 Message View

Message views describe the sequencing of message interchanges that occur between instances of classes of the model. There must be at least one message view for each public operation defined by a class in the structural view. Message views extend a subset of UML sequence diagrams [97]. In contrary to UML, no textual descriptions are possible. Instead, the message views refer to elements defined in the structural view making it necessary for them to be defined there first [101].

Figure 2.12 shows the behaviour of the design realization model of the base feature *Observer*. In contrast to UML sequence diagrams, message views provide means to define behaviour that is closer to the implementation. This facilitates the generation of code from it. For example, temporary properties can be defined and used, values assigned to properties (see Figure 2.12), and a try-catch combined fragment operator allows one to catch exceptions. For unsupported programming statements, an execution statement can be added to the message view which allows one to specify code as a string. This string is directly used when generating code.

To support aspect-oriented composition of behaviour, aspect message views provide an advice that is woven into the behaviour of operations. Figure 2.13 shows the behaviour that needs to be woven into methods that modify the subject in order to notify all registered observers. The location of the behaviour of the affected operation is denoted by the *original behaviour* (shown as a box containing a '*'). I.e., the behaviour is specified relative to this position, which makes it possible

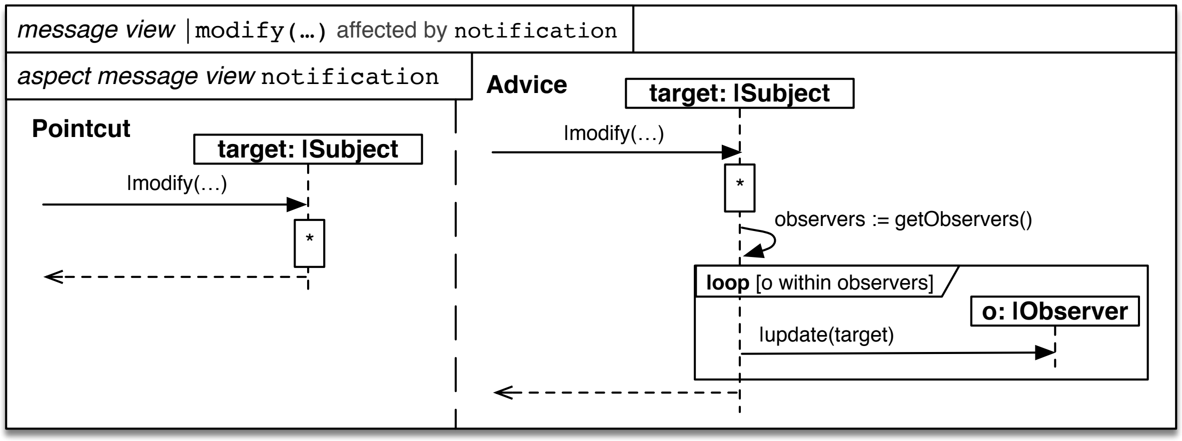

message view |modify(…) affected by notification
aspect message view notification
Pointcut
target: ISubject
Imodify(…)
*
Advice
target: ISubject
Imodify(…)
*
observers := getObservers()
loop [o within observers]
o: IObserver
Iupdate(target)

Figure 2.13: The *Aspect Message View* for Notifying Observers.

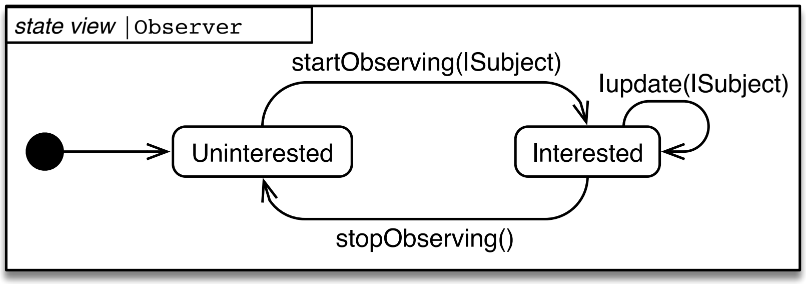


Figure 2.14: The *State View* for the `Observer` Class of the Base Feature of the *Observer* Concern.

to add behaviour *before*, *after* or *around* the original behaviour.

2.5.3 State View

State Views are similar to state diagrams and build on protocol models [72], which support composition and allow a protocol machine to refuse transitions. State Views allow the modeller to specify the different states an object can be in and a protocol that defines what method calls the object accepts in each state. State views serve as a mechanism to perform model checking or verification on whether message views comply with the defined state views. The modelled message views must conform to this protocol. For each class specified in the structural view that defines operations a state view has to be provided. One transition for each operation has to be at least contained in the state view.

Figure 2.14 shows the state view for the `Observer` class, which specifies that `stopObserving()` may only be called if `startObserving(|Subject)` was called prior. As is done with

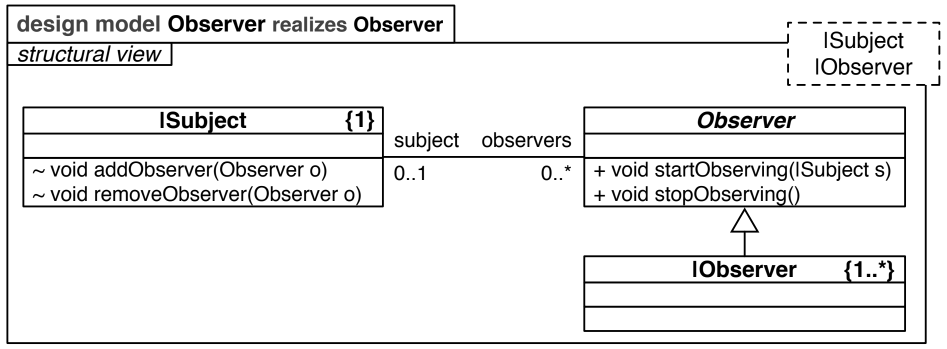

design model Observer realizes Observer
structural view
ISubject
IObserver
ISubject {1}
~ void addObserver(Observer o)
~ void removeObserver(Observer o)
subject
observers
0..1
0..*
Observer
+ void startObserving(ISubject s)
+ void stopObserving()
IObserver {1..*}

Figure 2.15: The *Structural View* for the *Observer* Feature of the *Observer* Concern with Mapping Cardinalities.

message views, state views reference operations from the structural view in their transitions. In addition, guards can be specified to define conditions for triggering a transition. Because protocol models support composition, several state machines can be specified for the same class. As a result, each state machine can be kept concise, and reasoning can be applied to each model locally. At the end, all state machines are composed together using parallel composition [72].

2.5.4 Mapping Cardinalities

As briefly outlined in Section 2.4.5, the CORE metamodel provides a `COREMappingCardinality` for all `COREModelElements`. In general, CORE allows the modeller to create multiple mappings for the same source element. For example, a user of the *Observer* concern might have a subject with multiple operations modifying it or needs several observers to handle update notifications in various places. The designer of a concern is the domain expert and knows how the model is intended to be extended or customized. By default, a model element has a cardinality of `{0..1}`, a partial model element has a cardinality of `{1}`.

The current design of the *Observer* as shown in Figure 2.10 does not support multiple kinds of observers. To allow multiple observers, the designer of *Observer* can adjust the structural design by introducing a super-class for observers that contains common structure and behaviour. Figure 2.15 shows the design of the base feature of *Observer* with mapping cardinalities. It is meant to be used for one subject which may be observed by one or more observers. The partial sub-class `|Observer` therefore has a cardinality of `{1..*}`.

Similarly, for the *Push* design realization model (see Figure 2.11), the designer wants to allow the ability to have multiple operations that modify the subject and an update method for each of those. The designer therefore defines a cardinality of `{m=1..*}` for operation `modify` of `Subject` and references this cardinality for operation `Observer.update`: `{m}`.

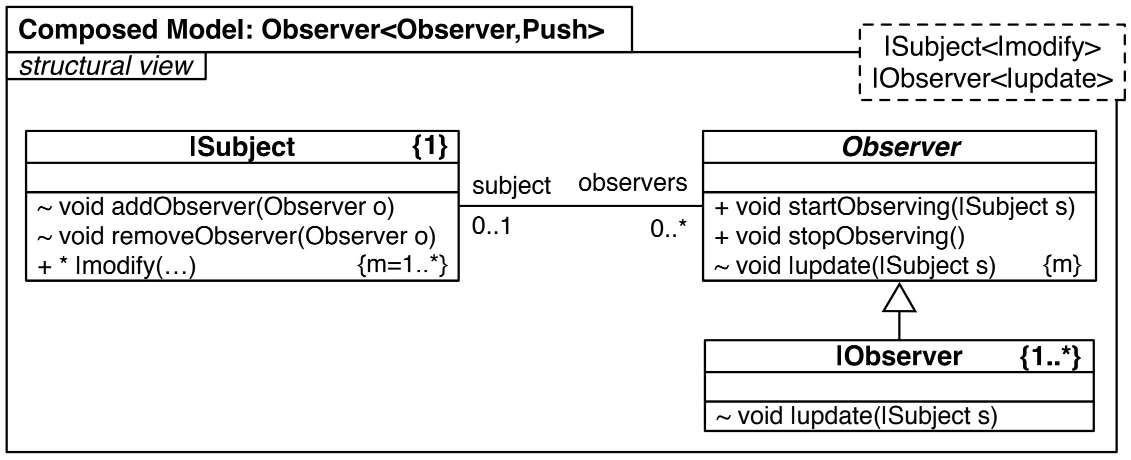

Composed Model: Observer<Observer,Push>
structural view
ISubject<Imodify>
IObserver<Iupdate>
ISubject {1}
~ void addObserver(Observer o)
~ void removeObserver(Observer o)
+ * Imodify(…) {m=1..*}
subject
0..1
observers
0..*
Observer
+ void startObserving(ISubject s)
+ void stopObserving()
~ void Iupdate(ISubject s) {m}
IObserver {1..*}
~ void Iupdate(ISubject s)

Figure 2.16: The Composed Model of the *Observer* Concern including Mapping Cardinalities.

2.5.5 Composition

As explained in Section 2.4.3, when a user selects features for reuse, the realizing models of the selected features (and their ancestors) are composed (woven) together. For example, if a user chooses the feature *Push* of the *Observer* concern (see Figure 2.2), automatically the parent feature *Observer* is selected as well. Since the design model of *Push* extends the design model of *Observer*, they are woven together. For the structural view, the classes that are mapped are merged together, unmapped classes are copied into the resulting structural view.

The composed structural view of *Observer<Push>* is shown in Figure 2.16. The classes also show the mapping cardinalities as outlined in the previous section. Message and state views are only copied when reusing a concern, and only woven into the final application to deal with complex dependencies (i.e., the diamond problem) [97]. For example, *Observer* might be reused in a *Stock* application in order for a window to get notified once the stock price changes. The modeller would map `|modify` to `setPrice` of the `Stock` class. Because `|modify` is affected by the *notification* aspect message view (see Figure 2.13), its behaviour is woven in at the matched join point defined by the pointcut and location of the original behaviour box. As a result, the behaviour to notify observers is added at the end of the behaviour of the `setPrice`.

2.6 TouchCORE Tool

In general, MDE approaches rely heavily on tool support. Tool support is even more important in the context of CORE, 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.

There exist tools for feature modelling (such as *FeatureIDE*[4]) and UML modelling (such as *Papyrus*[5]). Some tools combine feature models with other artefacts. For example, *Clafer*[6] combines feature models with class models [18]. *Clafer* is targeted at the early stages of software development, i.e., domain models are mainly used, and it provides reasoning support. The TouchCORE tool implements the principles of CORE and uses design models as defined in RAM. The contributions presented in this thesis are implemented within the TouchCORE tool.

TouchCORE[7] [1] is a multi-touch enabled, concern-oriented software design modelling tool that supports feature and impact models, as well as class, sequence and state diagrams. The TouchCORE tool was developed primarily to assist the concern designer when creating a concern with its three interfaces and the underlying design models. It also assists the concern user when reusing a concern by ensuring proper customization and usage. Finally, it hides the complexity of the aspect-oriented model composition and can execute the composition algorithms to visualize the composed design models.

The foundation of the backend of TouchCORE are the metamodels of CORE and RAM that define the abstract syntax for CORE and RAM models. The metamodels are defined using the *Eclipse Modeling Framework* (EMF). EMF allows the definition of a structured data model and generate the required Java code from it. EMF provides facilities to serialize models in XMI (*XML Metadata Interchange*). Further technologies from the *Eclipse Modeling Project*[8] are used, such as the *Object Constraint Language* (OCL) to define constraints for the metamodels.

[4]https://featureide.github.io
[5]https://www.eclipse.org/papyrus/
[6]https://www.clafer.org
[7]http://touchcore.cs.mcgill.ca
[8]https://www.eclipse.org/modeling/

Part II

Concernification: Raising the Abstraction Level of Frameworks

3
Concernification Overview

Concernification [102] is the process of creating a concern interface for an existing reusable artefact, such as a framework, that can be readily reused at the implementation level. It provides a way to bridge the gap between the design at the modelling level and the implementation level. *Concernification* makes it possible to lift existing reusable code artefacts up to the modelling level and exploit the benefits provided by the higher level of abstraction.

This chapter begins by motivating the need for concernification based on difficulties in reuse at the implementation and modelling level. Section 3.2 then introduces a general overview of concernification. It is followed by an overview of the benefits (Section 3.3) a concern interface provides, as well as the steps that are required to *concernify* an existing reusable code artefact (Section 3.4). Section 3.5 shows a concern interface based on the example of a reusable framework called *Minueto*. Finally, Section 3.6 shows how a concern interface can be reused and presents the size reduction of the API for different feature selections on the Minueto concern.

3.1 Motivation

Methodical reuse of software artefacts is considered key to software engineering [65, 70]. Instead of creating all functionality from scratch, common and recurring functionality is reused. However, at the modelling level reuse is not very common [123]. Models are often created from scratch or created by copy-and-pasting existing model fragments. This is due to the fact that there are basically no reusable models available, and because modelling languages and modelling tools do not have good support for reuse. Finally, creating reusable models is difficult, because models are defined at a higher level of abstraction, and typically, reuse is done in a bottom-up way by reusing specific artefacts depending on implementation choices. MDE advocates a top-down approach, where models are continuously refined with more detail, to eventually be able to generate code.

At the implementation level, common and recurring functionality is packaged into frameworks or libraries[1] with the purpose of making it reusable. They are usually well maintained, continuously improved, and come with textual documentation, different forms of code examples, and other artefacts. Many are publicly available, available as open source and have large communities. In addition, companies are interested in reusing existing software artefacts in order to amortize development costs by increasing quality and productivity [74]. However, reuse at the implementation level has problems as well. The code of a framework usually comes as one monolithic block. Even if a developer only wants to use a small feature set of a framework, he is nevertheless confronted with the complete API. Furthermore, developers need to find out and understand how to adapt the reusable entities for their own needs, and then how to use the usage interface (API) correctly. Research has identified many obstacles that developers face when learning and using APIs [33, 78, 91, 92, 120], such as ambiguous, incomplete or absent documentation and how to correctly use an API for a task. A study on programmer questions [48] revealed as one issue "De-localized concerns": "*An object-oriented design [...] often consists of a number of classes and interfaces, and the source code for a feature or concern may be scattered in multiple syntactic constructs. How to modularize such concerns is an interesting research topic.*"

In the following sections we describe the idea and benefits of *Concernification* that proposes to modularize the API of a framework according to features and to raise the level of abstraction of the API to the modelling level to benefit from the higher level of abstraction.

3.2 General Idea

In order to overcome the problems and limitations described in Section 3.1 and to bridge the implementation and modelling levels, we raise the level of abstraction of frameworks by lifting their APIs to the modelling level. This process is called *concernification* and produces as a result a concern interface for an *existing code artefact* [102], i.e., the existing code at the implementation level is reused and made available at the modelling level. In addition, the higher-level abstraction of the interface can be exploited to formalize information of a framework that is otherwise informally described (such as textual documentation). In particular, building on the VCU interfaces provided by CORE allows the concern interface to expose the user-relevant features and their impacts on non-functional goals and qualities in a variation interface, describe how to adapt the framework to the reuse context with a customization interface, and modularize the framework's functional API according to the user-relevant features with a usage interface. These benefits are described in more detail in the following section.

Therefore, concernification makes the framework available at the modelling level and allows it to be (re)used in the context of MDE by a modeller. However, the benefits are also available at the

[1] In the remainder of this thesis, the term *framework* will be used to refer to any reusable code artefact/library.

implementation level to the programmer. In both cases, the concern interface clearly documents the framework, which can help a user understand the framework better and help avoid mistakes.

3.3 Benefits

Framework developers generally provide different forms of information or documentation that describe what the framework provides and how to use it. Common forms are textual documentation, tutorials, API reference, *How Tos*, and executable examples. However, this information is often informal, scattered, and incomplete, making it difficult for the user to gain a clear understanding of the framework. The proposed concern-oriented model interface for frameworks formalizes the information and provides it in one place. The advantages are described in the following subsections.

3.3.1 Documents and Organizes Features

The *variation interface* of a concern provides a high-level, formal, organized view of the user-relevant features that a framework provides. As described in Section 2.4, 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 framework from the user's perspective. Successful concernification of a framework therefore requires the identification of all user-perceived features of a framework and their relationships (parent-child and cross-tree constraints).

3.3.2 Provides Impacts

The *variation interface* of a concern also 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. Impacts are expressed using impact models, a variant of goal models [54]. An impact model contains all high-level goals and qualities that experienced framework developers identified as relevant to consider. By specifying links with relative weights, the framework developer can express how each feature affects high-level goals and qualities. When reusing a framework, tool supported impact model evaluation allows the user to perform trade-off analysis between different feature selections [5].

3.3.3 Tailors the API to the User's Needs

The public classes and methods of a framework constitute the *usage interface* (API). Once a feature selection has been made by the concern user, only the subset of the framework API corresponding to the user's needs is exposed to the user, thus reducing the API complexity and cognitive load on the user to a minimum.

3.3.4 Provides Usage Protocols

The *usage interface* of a concern expresses the usage protocol of the different classes in the API formally. We use protocol models, which are similar to state diagrams, but support composition and allow a protocol machine to refuse transitions [2]. Specifying a usage protocol for a framework not only serves as an additional form of documentation, but can be exploited by a modelling tool or development IDE to ensure that the API of the framework is used correctly, i.e., that the operations provided by the framework are invoked in the correct order. This helps avoid user mistakes that are not detectable with static code analysis.

3.3.5 Guarantees Correct Reuse

The *customization interface* of a concern can be used to force the user to correctly adapt the chosen functionality to the reuse context. For example, choosing a certain feature might require a specific class to be extended or an interface implemented. To enforce this, a partial class and/or operation is added to the design model realizing the feature, which forces the user to provide a mapping from the partial element to some model elements in his application model.

3.3.6 Provides Glue Code

Finally, repetitive "glue code" that is required to use the API of a framework within an application can be provided in the concern of the framework itself. This "glue code" entails pre-defined structure and behaviour, and is composed with the application's structure and behaviour before executable code is generated. This lowers the amount of work required to reuse a framework and reduces again the possibility for making mistakes. Therefore, the user can focus more on the logic of the application under development.

3.4 Concernification Steps

Naturally, no existing framework provides a concern interface along with its other documentation. Ideally, such a concern interface would be created and maintained during the development of the framework. There exist a large number of reusable frameworks that are readily available for others to be reused. In order for an existing framework to be concernified, the following steps need to be performed for the different interfaces.

1. Variation Interface:

   (a) Determine the user-relevant features that a framework offers, i.e., the distinct framework features that a user can make use of.

   (b) Determine the inherent dependencies among the framework features, and create a feature model that documents the dependencies. Successful concernification of a framework 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*).
   (c) Determine the high-level goals the framework might impact and create impact models for each goal.

2. Customization and Usage Interface:
   (a) Determine which elements, i.e., public classes and methods, of the framework's API belong to which feature.
   (b) For each feature, create a design realization model that groups all API elements associated with the feature. As in CORE design models can extend other design models by adding additional classes and operations, API elements shared by multiple features should be put into shared parent design models.
   (c) Determine for each element in the API whether or not it is supposed to be in the customization and/or usage interface.
   (d) Determine and specify the usage protocols for the operations of each class in the design model.
   (e) Identify "glue code" required to use features and add structure and behaviour to the design realization model. Add affected classes and operations to the customization interface.

In general, performing *concernification* on an existing framework involves considerable effort. However, we argue that this high up-front effort makes sense, if the framework is reused a lot. Additionally, frameworks are typically long-lived. More information, such as new impacts, and additional features can be added over time. Here, collaborative, open source user communities can be exploited for this.

In the following chapters of thesis, we focus on steps 1a and 1b for the variation interface, and steps 2a, 2b and 2c for the customization and usage interface. The remaining steps are out of the scope of this thesis.

3.5 Concernification by Example

To illustrate concernification, we manually *concernified* the API of a small framework called *Minueto* [27]. Minueto is written in Java and 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 is shipped with several documentation artefacts:

- A *How To* section on the Minueto website [73] 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.

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. These 40 classes and interfaces have a total of 260 public and protected methods and constants (~6 on average per class).

In order to concernify a framework, in-depth knowledge of the framework is required. We therefore first invested time to familiarize ourselves with Minueto. The different forms of documentation outlined above were studied. We experimented with the 30 small runnable examples provided by Minueto that explain how to use different functionality of the framework. Each example mostly showcases one use case (such as drawing a specific shape, handling input, etc.). We further studied the source code to gain additional knowledge about the framework. Finally, we created a complete class diagram of the framework to gain an understanding of the big picture.

We then elaborated a feature model based on the examples and 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. We created a feature model, which evolved from a feature model that resembled the class hierarchy a lot to one that focused on the user-relevant features only. The feature model was iteratively refined until we reached the final version (see Figure 3.1) that—from our point of view—reflects the user's perspective well.

Since we considered the API during elaboration of the feature model, we assigned the API elements (classes and operations) to their corresponding features during this process. Some oper-

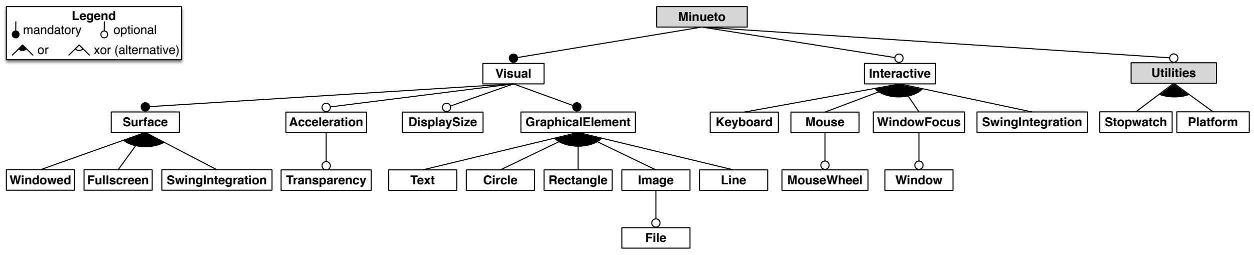


Figure 3.1: Hand-Made Minueto Feature Model

ations were moved out of their class as they belong to a different feature than the one introducing the class. Furthermore, we took note of any usage protocol being mentioned in the documentation and repetitive code that is always or often required and indicates potential glue code.

In the following sub-sections we illustrate the concern interface for Minueto based on this manual process according to the benefits described in Section 3.3.

3.5.1 Variation Interface of Minueto

This subsection describes in detail the variation interface of Minueto. We begin by describing the feature model which documents and organizes the features. We then discuss the impact model which describes how the features impact high-level goals.

3.5.1.1 Minueto Feature Model

Figure 3.1 shows the final feature model that we determined. The features with a grey background are solely for structuring purposes. They do not have a design model that realizes them, and hence do not contain any classes or methods of the API.

The functionality of Minueto is divided into three clusters. The mandatory feature *Visual* provides everything related to graphical visualization. It is divided into the mandatory feature *Surface* for different window modes, the mandatory feature *GraphicalElement* for different elements that can be drawn, and the optional features *Acceleration* (hardware acceleration and support for transparency) and *DisplaySize* (to retrieve the size of the display at runtime).

The second cluster is the optional feature *Interactive,* which provides the event queue and different kinds of event handlers. Lastly, the feature group *Utilities* provides stopwatch functionality for timing purposes, and a utility class to determine at runtime which operating system the application is executed on.

In total there are 26 features of which 4 are mandatory[2]. Two of them require at least one of their children to be selected as well, because they have an OR-relationship to their children. This

[2]The root feature is implicitly mandatory and is always selected.

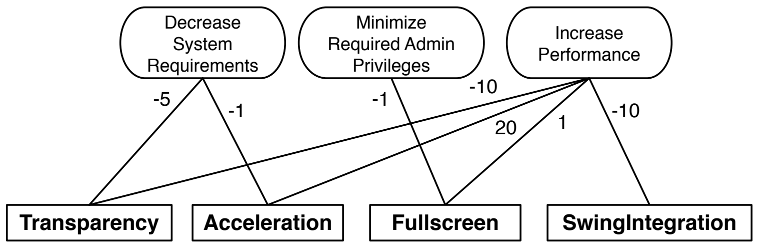


Figure 3.2: Minueto Impact Model

means the minimum number of features that can be selected is 6 features (e.g., *Minueto*, *Visual*, *Surface*, *Windowed*, *GraphicalElement*, *Text*). Note that there is no single correct feature model for a framework. The feature model shown in Figure 3.1 is *one possible* feature model for Minueto. Other feature models—even models with the exact same features and just structural differences—would be valid as well, as long as the complete Minueto API is modularized correctly. For example, *DisplaySize* and *Acceleration* (with its child *Transparency*) could also be grouped under *Utilities*, even though they are related to visualization. At the same time, one could argue that *Surface* (or *Visual*) should provide all means for different window sizes (*DisplaySize*). This depends on the structuring and granularity one wants to attain.

3.5.1.2 Impact Model

Based on the documentation provided with Minueto, we extracted some high-level goals. Figure 3.2 shows the non-exhaustive goals we determined. For example, it states that from a performance point of view, the best set of features to use is to select *Fullscreen* and *Acceleration*, but not to select *Transparency* and *SwingIntegration*. The latter features contribute negatively, because they reduce the performance due to increased resource usage. *Fullscreen*, however, requires admin privileges in order to create an application spanning the full screen. The individual weights contributing to the high-level goals were determined based on how they affect the high-level goal compared to other features contributing to the goal. Unfortunately, it is currently not possible to specify contextual information in the impact models used in CORE.

Choosing *Transparency* and *Acceleration* increases the hardware demands on the graphic card, and hence is the worst possible selection if one wants to keep the system requirements low. This means that any other feature can be selected but those two in order to achieve the best possible result for *Decrease System Requirements*.

If at any point in time more knowledge is acquired about how these or other features affect any of the goals, or if new goals are discovered or become relevant, the impact model can be updated to reflect that.

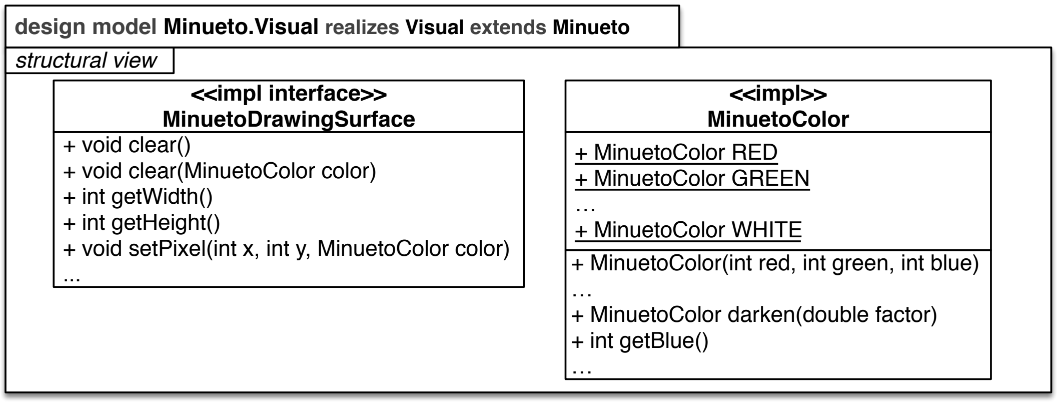

design model Minueto.Visual realizes Visual extends Minueto
structural view
<<impl interface>>
MinuetoDrawingSurface
+ void clear()
+ void clear(MinuetoColor color)
+ int getWidth()
+ int getHeight()
+ void setPixel(int x, int y, MinuetoColor color)
...
<<impl>>
MinuetoColor
+ MinuetoColor RED
+ MinuetoColor GREEN
…
+ MinuetoColor WHITE
+ MinuetoColor(int red, int green, int blue)
…
+ MinuetoColor darken(double factor)
+ int getBlue()
…

Figure 3.3: The Interface of the Feature *Visual* (sub-feature of *Minueto*)

3.5.2 Usage Interface of Minueto

This section describes in detail the usage interface of Minueto. We first discuss how the API is modularized across the features. We then discuss the usage protocol of different classes in the API.

3.5.2.1 Minueto API

The complete source of Minueto consists of 2443 lines of code. In total the Minueto API is comprised of 32 public classes (of which 0 are abstract), 8 public interfaces, 153 public methods, 18 protected methods, and 89 attributes/constants that are public. Each of these needs to be assigned to its corresponding feature. A class or method belongs to a feature when the behaviour offered by the class or method encodes functionality relevant for the feature. For example, creating a windowed game requires the user to instantiate the `MinuetoFrame` class. Furthermore, the user might want to set the window's position. Therefore, the *Windowed* feature needs to include the relevant classes and methods. If the *Windowed* feature is not selected, this means that the user does not see corresponding classes and methods and therefore does not need to think about them.

Following the Minueto feature model given in Figure 3.1, at the top is the *Minueto* feature. From the API's perspective this root feature is only grouping the sub-features and does not introduce any elements from the API. The feature *Visual* is responsible for providing general elements for visualization. Its realization model is shown in Figure 3.3. It introduces the main interface `MinuetoDrawingSurface` along with most of its methods and the `MinuetoColor` class with its methods and colour constants.

The realization model of the feature *Surface* extends the realization model of *Visual* (see Figure 3.4). It introduces the interface `MinuetoWindow` (sub-interface of `MinuetoDrawing-Surface`) along with its general methods. For example, to make the window visible and close it,

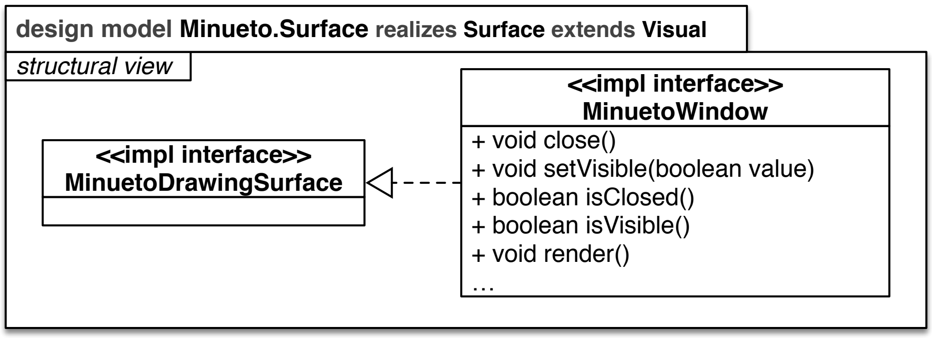


Figure 3.4: The Interface of the Feature *Surface* (sub-feature of *Visual*)

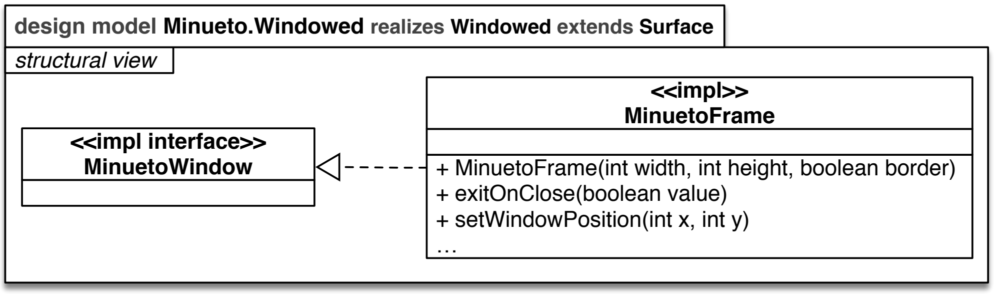

design model Minueto.Windowed realizes Windowed extends Surface
structural view
<<impl interface>>
MinuetoWindow
<<impl>>
MinuetoFrame
+ MinuetoFrame(int width, int height, boolean border)
+ exitOnClose(boolean value)
+ setWindowPosition(int x, int y)
...

Figure 3.5: The Interface of the Feature *Windowed* (sub-feature of *Surface*)

render the window, inquire on its visible or closed status, etc.

The leaf feature *Windowed* as one of the sub-features of *Surface* provides a concrete `Minueto-Window` implementation for windowed applications. Its realization model (see Figure 3.5) provides the `MinuetoFrame` class along with specific methods for windows, such as setting and getting the window's position, setting whether to automatically exit the application when the close button is pressed, etc.

Similarly, the *GraphicalElement* feature introduces the `MinuetoImage` class along with its transformation methods (i.e., `scale`, `crop`, `rotate`). In addition, the method `drawImage(-MinuetoImage, int, int)` from `MinuetoDrawingSurface` is placed in the *GraphicalElement* feature as it requires the `MinuetoImage` class, which is introduced here. The sub-features of *GraphicalElement* introduce each a specific sub-class of `MinuetoImage` to draw specific graphical elements. One notable exception is the feature *Line*, which solely introduces the method `drawLine(MinuetoColor, int, int, int, int)` from the `Minueto-DrawingSurface` interface. As it can be considered a graphical element as well—independent of how it is actually implemented—it is grouped as a sub-feature of *GraphicalElement*.

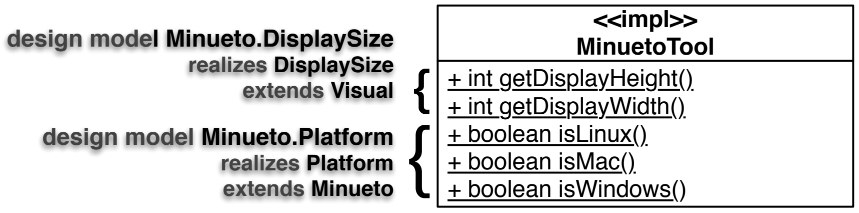


Figure 3.6: The Interface of the Class `MinuetoTool` and the Relation of its Methods to Features

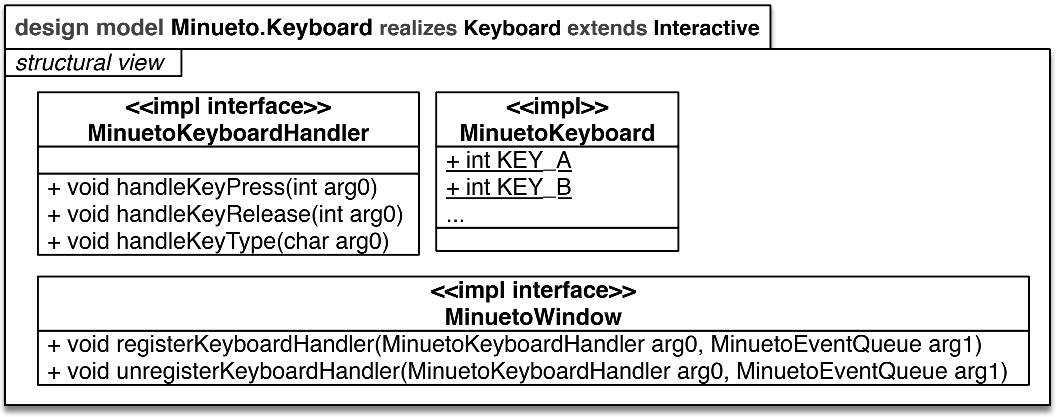

design model Minueto.Keyboard realizes Keyboard extends Interactive
structural view
<<impl interface>>
MinuetoKeyboardHandler
+ void handleKeyPress(int arg0)
+ void handleKeyRelease(int arg0)
+ void handleKeyType(char arg0)
<<impl>>
MinuetoKeyboard
+ int KEY_A
+ int KEY_B
...
<<impl interface>>
MinuetoWindow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

Figure 3.7: The Interface of the Feature *Keyboard* (sub-feature of *Interactive*)

The class `MinuetoTool` showcases a class whose static helper methods belong to different user features as well. Figure 3.6 shows the class with all its methods. The first two methods relate to retrieving the size of the display and are assigned to the feature *DisplaySize*. The remaining three methods allow a developer to find out which platform the application is run on. These are assigned to the feature *Platform*. In the same way, the methods of the class `MinuetoOptions` are split between the features *Transparency* and *Acceleration*.

The feature *Stopwatch* on the other hand contains one class with all its methods. The class `MinuetoStopwatch` provides timing functionality, for example, to measure the frame rate. Similarly, the feature *Interactive* provides the overall functionality required to handle events. It contains the `MinuetoEventQueue` class with its corresponding methods to determine whether there is an event in the queue and to get the next one. Typically, the code to handle events with the event queue is added to the render loop of the application.

The sub-features of *Interactive* provide the specific handlers and any corresponding classes and methods that are required. For example, Figure 3.7 shows the realization model of the feature *Keyboard*. It contains the interface `MinuetoKeyboardHandler` that needs to be imple-

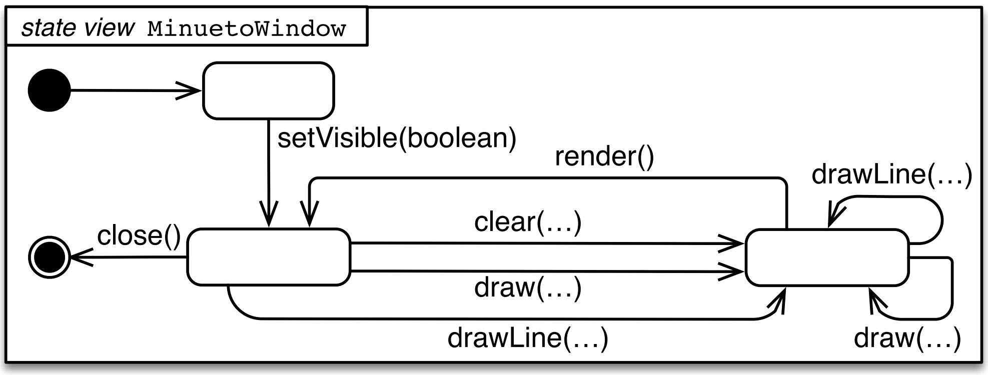


Figure 3.8: The Protocol Model of the Feature *Surface* for the `MinuetoWindow` Interface

mented, and the methods to register and unregister the `MinuetoKeyboardHandler` instance on the window. As such, the register and unregister methods from `MinuetoWindow` belong to the feature *Keyboard* and therefore are located in its realization model. The `MinuetoWindow` interface is located in the realization model of the mandatory feature *Surface*, therefore, they are always available. When using the *Keyboard* feature, the `register` and `unregister` methods are made available in addition.

3.5.2.2 Usage Protocol

Frameworks often require certain methods to be called and sometimes they need to be called in a certain order. This is usually described in textual documentation and the user is required to find and remember it when writing code. In CORE, the usage protocol of a framework can be formalized using protocol models [2], which are part of the usage interface. The user can view the protocol and understand in what order calls need to be made and how calls affect the state the class is in. Furthermore, the usage protocol can be used by model checkers or code analyzers to ensure that the API is used correctly by the user.

For example, the usage protocol for the `MinuetoWindow` interface provided in the realization model of the feature *Surface* is shown in Figure 3.8. It shows the required order in which the methods of the interface are supposed to be invoked as outlined by the example in Listing 3.1. For instance, it ensures that `render()` is only called after drawing something. Furthermore, the `clear(...)` method is optional and can only be called before drawing.

Usage protocols can also help with avoiding exceptions during runtime. Minueto, for example, defines several methods that throw exceptions during runtime if an instance is not in the required state or invalid values have been passed as parameters. For instance, `MinuetoFrame` and `MinuetoFullscreen` can throw a `MinuetoInvalidStateException` if a method

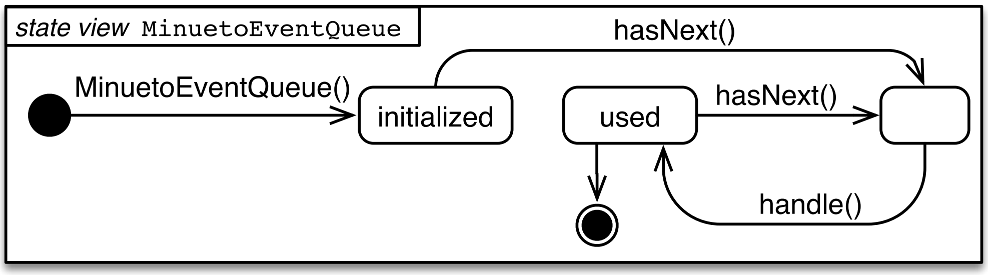


Figure 3.9: The Protocol Model of the Feature *Interactive* for the `MinuetoEventQueue` Class

requiring the window to be visible (not hidden or closed) is called when the window is not visible. This is already covered with the given usage protocol. However, some methods throw a `MinuetoOutOfBoundsException` depending on the actual parameters. For example, the method `setWindowPosition(int, int)` of `MinuetoFrame` throws this exception if either of the two parameters is negative[3]. The protocol model can incorporate this by specifying a guard on the transition for `setWindowPosition` to ensure that `x >= 0 && y >= 0`.

In addition, based on the usage protocol it is also possible to verify that a class is actually being used and not just instantiated. For example, it would be incorrect to create an instance of `MinuetoEventQueue` and never call any of its methods. Figure 3.9 shows the protocol model of the `MinuetoEventQueue` class where this is reflected. However, our feature model of Minueto (see Figure 3.1) requires that at least one of the handlers (sub-features of *Interactive*) is used if the user chooses the *Interactive* feature. This means that it is not enough to just satisfy the protocol of `MinuetoEventQueue` without registering at least one handler. Therefore, the protocol models of the handlers need to specify as a requirement to register the respective handler during the setup of the window. In fact, among the 30 runnable examples that are shipped with Minueto, two of the examples[4] violate this requirement by instantiating and using the `MinuetoEventQueue` (as shown in Listing 3.1) without registering any handler. While the examples still run, the API is used incorrectly by instantiating an event queue but not doing anything interactive. I.e., the loop to handle events, as shown in Listing 3.1 above, never executes the body of the loop. If there had been a protocol model for Minueto, a model checker would have been able to detect this error.

[3]This is not mentioned in the official API reference but determined based on the documentation comments within the source code.

[4]`HelloWorld` and `TextDemo`.

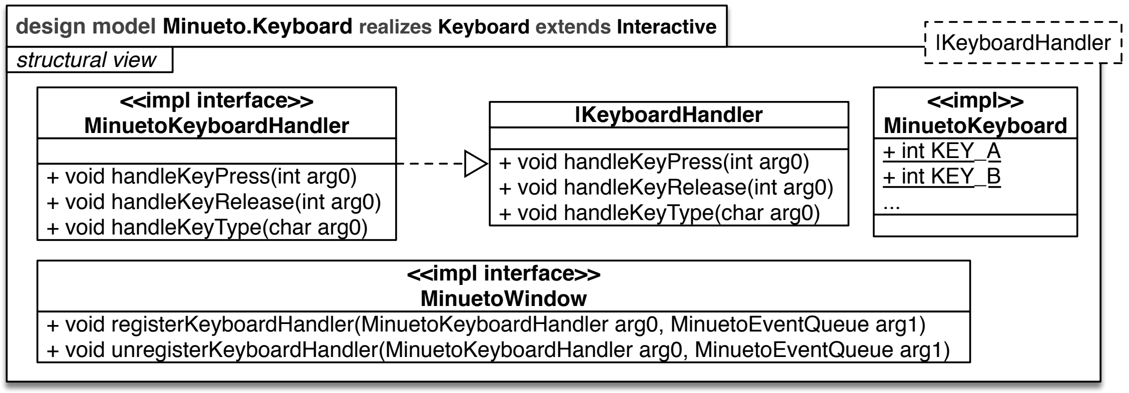

design model Minueto.Keyboard realizes Keyboard extends Interactive
structural view
IKeyboardHandler
<<impl interface>> MinuetoKeyboardHandler
+ void handleKeyPress(int arg0)
+ void handleKeyRelease(int arg0)
+ void handleKeyType(char arg0)
IKeyboardHandler
+ void handleKeyPress(int arg0)
+ void handleKeyRelease(int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

Figure 3.10: The Interface of the Feature *Keyboard* with the Partial `KeyboardHandler` Class

3.5.3 Customization Interface of Minueto

This section describes in detail the customization interface of Minueto. We first discuss how mandatory customization steps by the user can be enforced. We then describe how glue code is integrated into the concern interface.

3.5.3.1 Guaranteeing Correct Reuse

In addition to assigning elements of the API to a feature, realization models can contain additional information. Frameworks sometimes require a user to implement an interface, sub-class an abstract class or customize a method. To ensure that the user knows about this and correctly reuses the framework, CORE allows a developer to mark elements as partial (incomplete). This leads to the element being listed within the customization interface of the concern and enforces that the user provides a customization for the partial element.

For example, the realization model of the feature *Keyboard* (see Figure 3.7) contains the `MinuetoKeyboardHandler` interface. To react to keyboard inputs, the user building an application must implement this interface. To ensure that this is done, a class—which implements the `MinuetoKeyboardHandler` interface—is added to the realization model and marked as partial.

Figure 3.10 shows the realization model with the `|KeyboardHandler` class added. The top-right corner shows the customization interface of this realization model. One added benefit of this is that when implementing an interface, all its methods need to be implemented. It happens often that only a subset of all interface methods is actually used and therefore the unused ones are implemented with an empty body. At the implementation level, the designer can provide default implementations if it is predicted that this situation might occur frequently. Similarly, this situation can be covered in the concern interface by allowing the realization model to provide default (empty) implementations of the interface methods. However, this means that the user might overlook that these methods should be implemented. To enforce the user to provide an implementation, the methods can be marked as partial.

3.5.3.2 Glue Code

Using frameworks often requires the user to write code that “glues” the application code with the framework. Often this code is repetitive because it is the same or very similar for all users and contains no or only some application-specific details. To avoid this, concernification makes it possible to integrate glue code within the realization models themselves, therefore making the glue code be part of the concern interface of the framework. This pre-defined structure and behaviour is then composed together with the application-specific structure and behaviour, relieving the user of this repetitive task. The user can benefit from this and only needs to focus on the logic of the application itself.

Listing 3.1 Main Code to Run an Application with Minueto Including Handling of Events

```java
MinuetoWindow window = // create specific window instance

// Optional: To handle events.
MinuetoEventQueue eventQueue = new MinuetoEventQueue();

window.setVisible(true);

while (true) {
    // Optional: Clear the current canvas.
    window.clear();

    // TODO: Draw something.

    // Optional: Handle events.
    while (eventQueue.hasNext()) {
        eventQueue.handle();
    }

    window.render();
}
```

For example, each application using Minueto needs to have a window. In addition, a *game loop*—an infinite loop—needs to be defined to perform all draw operations and to trigger rendering

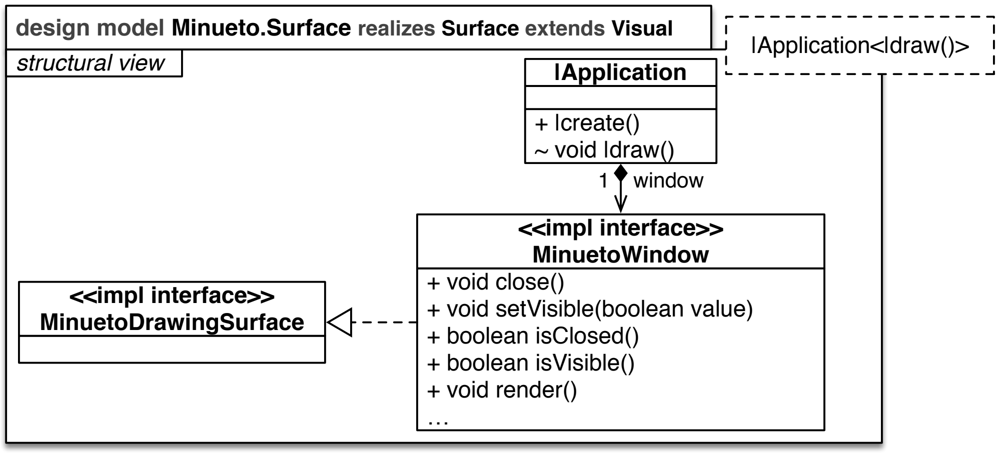


Figure 3.11: The Interface of the Feature Surface with Added Glue Code Structure

on the screen. In the case of interactive applications (making use of the *Interactive* feature), the event queue is used to handle any input events within the *game loop* before rendering. This partial structure and behaviour can be already contained within a realization model in order for the user not to have to deal with this. The required code to realize this is given in Listing 3.1.

For example, the fact that an application always has a window and a game loop can already be contained within the realization model of the *Surface* feature. Figure 3.11 shows the structure of *Surface* with an additional `|Application` class. It represents the main application class responsible of starting the program and contains a `MinuetoWindow`, which the user will have to provide the concrete instance of. In addition, two partial elements are defined. The `|draw()` method corresponds to the method taking care of drawing everything on the screen to be provided by the user. The partial empty constructor is defined in order to integrate the partial behaviour for the game loop.

Figure 3.12 shows the message view definition of `|create()` along with the aspect message view *startGameLoop*, which after any behaviour defined by the user makes the window visible and starts the infinite loop where the user-provided drawing method is called, followed by rendering everything on screen. The realization model of *Interactive* can then provide additional partial structure by providing the event queue and initialize it by *advising* the partial behaviour of the constructor. Furthermore, before rendering everything on screen, the loop to handle all events using the event queue can be inserted.

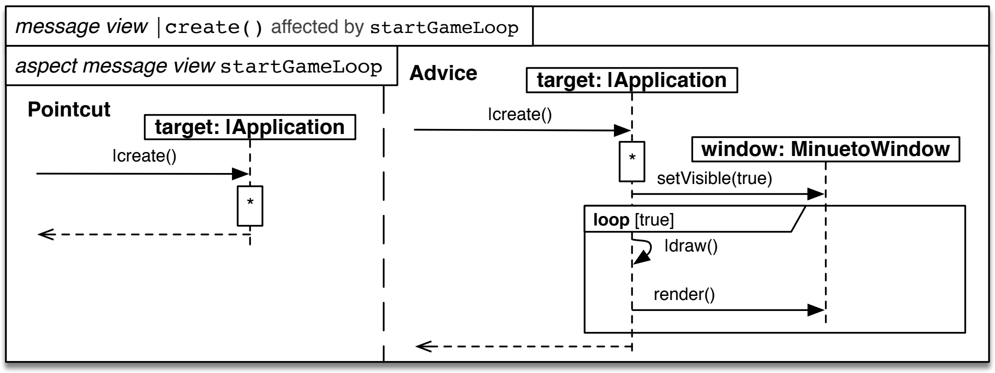


Figure 3.12: The Partial Glue Code Behaviour of the Feature *Surface*

3.6 Reusing a Concernified Framework

A user can reuse the *Minueto* concern by following the three-step process as outlined in Section 2.4.2. While choosing the features the user desires, the impact models are evaluated, which allows the user to do trade-off analysis based on the intended high-level qualities for the system. Once the feature selection is done, the user receives a custom-tailored Minueto API based on the feature selection. It is a subset of the API containing only those elements that are related to the selected features. This provides the user with the *minimal* API for his specific needs, by hiding access to those elements that the user does not need to be concerned with.

The customization interface then tells the user how to adapt the framework. If any partial structure and behaviour is provided, as explained in Sections 3.5.3.1 and 3.5.3.2, the user is forced to provide customizations for partial elements. This ensures that the user provides concrete elements within the application and relieves the user from having to define repetitive glue code. This is based on the result of composing all realization models of the selected features. The usage interface consists of the subset of the complete API and allows the user to use the parts of the framework based on the feature selection. With the help of the usage protocol, it can be ensured that the API is used correctly.

The minimal selection of features for Minueto consists of the root feature *Minueto*, *Visual*, *Surface*, one of the sub-features of *Surface*, *GraphicalElement*, and one of the sub-features of *GraphicalElement*. With those features, it is possible to create an application with a window surface, and to draw some shape on that surface. Taking the classic *hello world* example that is commonly used to introduce someone to a new (programming) language, the selection of sub-features could be, for example, *Windowed* and *Text*. Selecting a non-root feature means that also all its ancestors are selected, therefore, the selection of *Windowed* and *Text* would also result in the selection of features

Table 3.1: Comparison of API Size Based on Feature Selection

Feature Selection[5]	**Classes**	**Interfaces**	**Methods**	**Constants**	**Percentage of Complete API**
Windowed, Text	5	2	56	11	26.2
Fullscreen, Rectangle	4	2	47	6	20.9
Windowed, Fullscreen, Text, Rectangle, Keyboard, WindowFocus, Mouse	10	4	72	89	62
All	32	8	153	89	100

Surface, *GraphicalElement, Visual*, and *Minueto*. Table 3.1 shows a comparison of some feature selections based on the number of different elements and which percentage of the total number of elements of the complete API is retrieved. While the feature selection of *Windowed* and *Text* gives access to 26.2% of API elements, to accomplish the *hello world* example only 7 methods and 1 constant are required. This makes it only 5% of the complete API and 20% of the partial API of the *Windowed* and *Text* feature selection. With a finer granularity of features, the API could be divided further into more sub-features. The reduced API significantly reduces the cognitive effort for a newcomer to write a hello world application.

3.7 Summary

This chapter proposed concernification of frameworks, a process to produce a concern interface for an existing framework that is reusable at the implementation level. The higher level of abstraction is exploited to formalize information of the framework that is otherwise informally described. We build on the three interfaces for reuse provided by CORE. The variation interface documents the user-perceivable features and organizes them in a feature model. This provides the user with an organized, high-level overview of the functionality a framework provides. The functionalities are grouped according to features that are based on typical usage scenarios, and the relationships between the features are clearly expressed. The second part of the variation interface provides impacts that explain to the user how the different features of the framework impact non-functional properties. The usage interface constitutes the public classes and methods of the API. Each feature is linked to the part of the API of the framework required to use that feature. Secondly, the usage

[5]Ancestors of those features were omitted for brevity as they will be selected automatically.

interface also expresses the usage protocol of the API to document how to use the part of the API corresponding to each feature. Based on the selected features by the user the subset of the API relevant to only those features can be presented to the user. Furthermore, the customization interface allows the designer to include within the concern interface those classes and methods that need to be implemented by the user, thereby enforcing that this is done correctly.

The benefits of concernification are two-fold. First, the existing reusable code artefact can now be reused at the modelling level. This allows modellers to access the large amount of functionality offered by existing reusable frameworks. Second, the concern interface clearly documents the framework which supports the user at the modelling level. In addition, it could also support users reusing the framework at the implementation level in understanding what functionality the framework provides, and which API relates to specific functionality.

Open source communities, such as GitHub, are popular and allow users to report issues, suggest new features and contribute additional code. Such a collaborative community could also be used to allow users to suggest changes to the concern interface of a framework. For example, a user could suggest new or different impacts for high-level goals based on experience with the framework.

In the remainder of this thesis, we focus on the steps to determine the user-relevant features and organize them in a feature model (steps 1a and 1b for the variation interface, see Section 3.4), and to determine the corresponding API elements for each of the features and the elements that need to be part of the customization interface (steps 2a, 2b, and 2c for the usage and customization interface).

To illustrate concern interfaces, we manually concernified an existing framework called *Minueto* in this chapter. We first familiarized ourselves with the framework making use of the various information made available by the developers. We then followed the steps outlined in Section 3.4 to create the three interfaces. This was a non-trivial and time-consuming process and required to continuously refine the concern interface. In order to help a developer of a framework in creating a concern interface, we intend to automate the steps mentioned above, with the aim of generating an initial concern interface that the developer can use and refine. In [102] we proposed to exploit key information available from a reusable framework and its API for automating this process. Before moving on to describe the automated concernification, we need to validate the manually created feature model of Minueto to ensure it accurately describes the user-perceivable features and their relationships. Furthermore, this will validate whether our intuitive use of key information we manually extracted from the Minueto code to build the concern interface was sound. In the following chapter, we hence describe the study we conducted with the developers of Minueto to validate our feature model.

4 Validating the Minueto Feature Model

Chapter 3 introduced the idea and process of *Concernification*. We used the reusable framework *Minueto* to showcase concernification. As a result, we derived a feature model with the corresponding API for each feature. This concern interface is based on our understanding of Minueto after familiarizing ourselves with the framework. In order to validate the Minueto feature model, we conducted a user study with the two developers of Minueto. The participants were asked to create their own feature models of Minueto, and individual interviews were conducted to discuss theirs and our feature model.

This chapter begins with an overview of the study and its goals. Section 4.2 then introduces the participants of the study and outlines their level of involvement in the framework's development. It is followed by a description of the study set-up in Section 4.3. The gathered data is analyzed in Section 4.4. First, by discussing the information the participants used to create their own feature models of Minueto. Second, the different versions of feature models are presented and the reasoning behind the participants discussed. The evolution of feature models is shown, and the feature models of the different actors compared. Following the analysis, we discuss interesting insight we gained from the interviews in Section 4.5. Section 4.6 discusses the limitations of the study in Section 4.6 and this chapter concludes with a summary in Section 4.7.

4.1 Overview

The concern interface of Minueto (discussed in the previous chapter, see Section 3.5) we created is based on our own knowledge of Minueto. We familiarized ourselves with the framework by reading all supplied documentation (such as the webpage, How To, FAQ, and API reference), executing the examples and viewing their source code, and taking a closer look at the framework's source code. As a result, we derived a Minueto feature model (see Figure 3.1) with the corresponding API for each feature. This feature model is based on our understanding of what a feature of Minueto constitutes and what features can be used in combination.

However, there exists no unique correct feature model for a given framework for multiple reasons. First, what should or should not be considered a feature of a framework is not clearly defined. In particular, it is very often possible to increase the level of detail and decompose a feature further, adding additional sub-features. Second, even for a fixed set of features, there exist many equivalent feature models that specify the same set of possible configurations with a different set of optional, OR, XOR, requires and excludes constraints between the features [25].

We conducted a qualitative study in which we performed semi-structured interviews with the two main developers of Minueto. The main goal of the study was to validate the feature model of our concern interface of Minueto. To do this, we asked the study participants to create their own feature model of Minueto. This allowed us to compare their feature models with ours to confirm its validity.

In addition, we intended to confirm the key information we proposed in [102] that is available from a reusable framework and from its API and hence can be used to concernify a framework. The information initially considered were the Object-Oriented hierarchy, usage of the API in runnable examples, and the package structure of the API. In addition, we also considered so called *cross-references* as valuable, i.e., references in method signatures, e.g., in return parameters and parameter types, that point to other classes.

As creating a concern interface for an existing framework is a time-consuming task, the key information can be used in automating the process to provide an initial concern interface. The validated feature model of this study can also be used during the design of this automated concernification process.

4.2 Study Participants

The Minueto framework was mainly developed by *Alexandre Denault* (D1) as part of his Master's thesis in 2005 at the School of Computer Science at McGill University. The framework was mainly geared for students to make it easier for them when developing games for their course projects. During its development, Minueto was used in software engineering project courses and further improved based on the student feedback. The last official release is version `2.0.1`, which was released in early 2010. Since its inception, it has been used extensively by many students, mostly for the software engineering project course.

The second developer, *Michael A. Hawker* (D2), contributed some functionality and runnable examples. Additionally, he has volunteered with and advised high-school students who used Minueto for their projects a few years ago. Furthermore, he added support for arcs to the framework in 2015[1].

[1] See the repository on GitHub at https://github.com/Mikeware/minueto.

For brevity, we refer to the participants by their data-coded alphanumeric identifier (ID). In summary, D1 is the core developer and architect of Minueto, whereas D2 can be more seen as a contributing developer and maintainer, having insight into the framework's inner workings.

4.3 Study Set-Up

One of the things we were interested in was to understand how a framework developer sees their own framework in terms of the user-perceived features it provides and how they relate to each other. We therefore first asked the participants to create their own feature model of Minueto independently. This ensured that the participants had an unbiased view of their framework's features. It also allowed them to re-familiarize themselves with Minueto. Along with the request we provided a brief explanation of feature models and a small example.

We then conducted semi-structured interviews [104] with each of the participants in which we asked open-ended questions, such as "How did you come up with the feature model of [framework]?", which could lead to follow-up questions. Furthermore, we also asked clarification questions about the feature model the participants provided.

The interview was split into three main parts, as follows:

- Discussion about the participant's feature model, eliciting:
  - What information was used to create it.
  - Which parts of the API correspond/belong to which feature.
- Discussion about the runnable examples provided with Minueto.
- Discussion about our feature model.

The complete interview guide with the main questions is provided in Appendix B. The interviews were scheduled to last an hour and recorded for in-depth offline analysis.

4.4 Data Analysis and Results

The interviews we conducted were transcribed and used as the base for data analysis. In addition, the notes taken during the interviews were considered. We analyzed the data using a combination of quantitative and qualitative methods [104]. In the following subsections we first discuss the information the developers used to create their own feature model. Then, we discuss in detail the variants of feature models and provide a comparison between them.

4.4.1 Information used for Feature Model Creation

Only D1 looked at the Minueto website in order to "*[...] remember what Minueto does*"$_{\mathrm{D1}}$[2].

API and OO hierarchy Both D1 and D2 looked at the API reference in detail, which D2 referred to as taking a "*more analytical approach*": "*[I] looked at how the API was kinda laid out [...] it was already pretty well structured I think, since it's meant to be approachable for university students just learning about all these sorts of things.*"$_{\mathrm{D2}}$. Considering the API layout means looking at package structure and OO hierarchy, D2 then went on to break down the framework and categorized its main functionalities into groups. Each group was then further broken down with specific features assigned to the group. Some features were broken down based on their OO hierarchy: *"once I did that top level I was like 'you know, there's a little bit more depth here', and that's when I [...] started thinking about the image hierarchy, of all the different types of* `MinuetoImage` *and all the different operations and [...] used the API as a guide at that point to drill in and [...] fill out some of those things to a bit more detail.*"$_{\mathrm{D2}}$. It became apparent during the interview that D1 did not use the API to come up with the initial feature model, as he explained D1 tried "*to separate the feature model from the class diagram*"$_{\mathrm{D1}}$. Contrary to that, D2 was trying to map off from the API while building the initial feature model.

API Usage in Examples D1 looked at some of the provided examples to "*[...] just reminisce about all the different features that were available*"$_{\mathrm{D1}}$. As the creator of most of these examples, he remembered what their purpose was: "*I remember the idea behind why I built the tutorial and how I built them. [...] it was to introduce one [...] aspect of the framework at a time. And so basically, it's to teach them one [new thing] in each [tutorial] and then [...] you could almost map it to one feature of the framework in the sense that, every tutorial is supposed to teach them one new thing about the framework.*"$_{\mathrm{D1}}$. While D2 did not use them to create the feature model, he knows what their purpose is. With the exception of one example—called *FireInTheSky*—which shows a full game using almost all features, "*the other ones are a little bit more [...] trying to focus on specific pieces of the API*"$_{\mathrm{D2}}$. However, both developers mentioned that to see which part of the API is used in which example they would use code analysis tools, e.g., static code analysis, to help with this task.

Cross-references D1 specifically mentioned to analyze parameters in method signatures during the interview because "*the fact of a certain parameter might be an indication of a certain feature that's in there*"$_{\mathrm{D1}}$. For example, when asked about the methods to register and unregister handlers in `MinuetoWindow`, D1 responded "*in the sense we could say[...] the keyboard feature and the*

[2]The quotes of the participants are associated with their identifier for traceability and to distinguish between the participants.

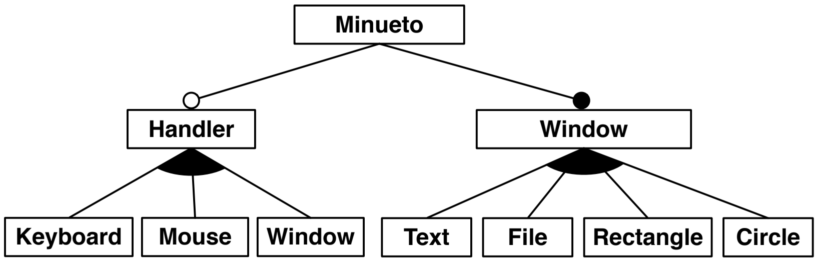


Figure 4.1: The initial feature model created by D1 ($D1_1$)

mouse features are both in that interface”$_{\mathrm{D1}}$.

Summary To summarize, both participants confirmed our theory of using different information available from a reusable framework and its API. Besides the API, this is the inheritance hierarchy, the runnable examples and their usage of the API, and the presence of references within parameters in method signatures. Packages provide a form of grouping classes according to some classification. While none of the participants specifically mentioned packages, the API reference shows on its index page an overview of all packages, and the packages are always visible in the top-left area. This list allows a developer to browse the classes by package. In addition, the runnable examples provided with Minueto are also structured in packages similar to the API itself.

4.4.2 Feature Model Variations

As explained in Section 4.1, the understanding of what a feature should be and how to organize them can differ. Therefore, it is not surprising that the feature models of D1, D2 and our own feature model were not identical. This subsection analyses and comments on the commonalities and differences.

4.4.2.1 Feature Models of D1

The initial feature model of D1 ($D1_1$) is shown in Figure 4.1. It is based on the two core functionalities that Minueto was built for: “*drawing stuff very fast on the screen, and picking up input*”$_{\mathrm{D1}}$. The *Handler* feature group provides different input types, whereas the *Window* feature provides the different things that can be drawn on the window. D1 struggled with knowing the class diagram and deriving a feature model for it. As the main person who implemented Minueto and remembering all the details in the implementation, he struggled with deriving a feature model from the class diagram, even though most of the time “*there is a very close mapping and so it was this challenge of trying to find a way to break down Minueto in a different way which represented more the features than the class diagram*”$_{\mathrm{D1}}$. A frequent discussion point in the interview was centred around the level of granularity of features. For example, Minueto makes use of double-buffering

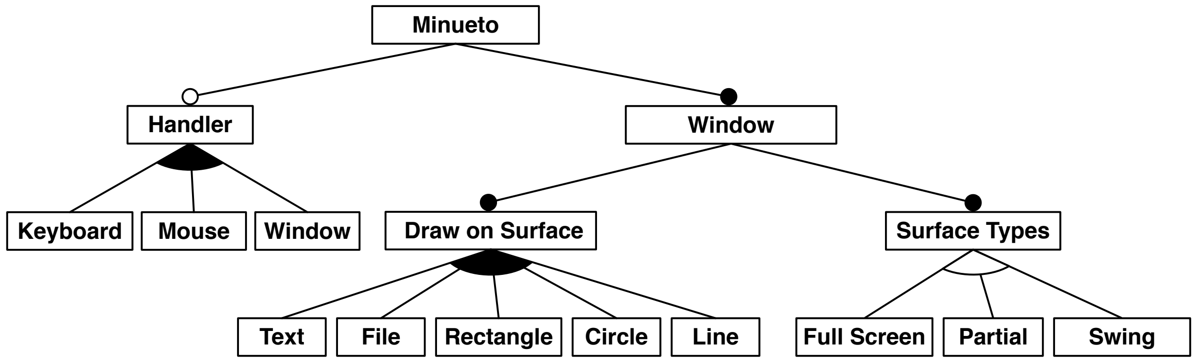


Figure 4.2: The updated feature model of D1 elaborated during the interview ($D1_{2a}$), increasing the level of granularity

to only draw on the screen once everything is finished drawing on the off-screen. However, this is always used and is not something that the user can enable or disable. When asked which things in Minueto could be used or not used, D1 mentioned hardware acceleration. D1 also confirmed that *Line* (i.e., the ability to draw a line) is considered a feature, but admitted that "*it's possible that my model was biased by what I think is pertinent vs. what I think is there*"$_{\mathrm{D1}}$. The following comment further indicated that the feature model was mostly derived from memory: "*If I wanted to do a [...] complete feature list of stuff that I can choose to use, then in that case I would say I would have to go through the source code and try to find every option that's in there*"$_{\mathrm{D1}}$.

Following the questions regarding the feature model $D1_1$ (see Figure 4.1) we asked D1 to go through the list of examples (see Appendix C) and list which features are used by which example. The names of some of the examples reminded D1 of features that he missed in his first feature model. For instance, this led to the addition of the *Line* feature as something that can be drawn on the screen. Furthermore, the different types of surfaces (or windows) were explicitly distinguished. The updated feature model $D1_{2a}$ is shown in Figure 4.2. However, when talking about the names of features that group other features, D1 was wondering whether there would "*be a notation to be able to describe what the relationship is. [...] with class diagrams it's very straight forward to describe the relationship. You're always talking about is-a-parent-of-the-things [...]*"$_{\mathrm{D1}}$. Referring to the feature group for the shapes that can be drawn, and the different kinds of windows, he went on to say that "*[...] here the relationship is things I can draw on the surface and this is types of surfaces I could use*"$_{\mathrm{D1}}$. This is in contrast to the *Handler* feature which groups different kinds of handlers. Similarly, the naming of features is a point that was mentioned many times during the interview, for instance when discussing the two groupings: "*[...] it's this comfort at the same time of figuring out what the [name of the feature] would be and as I was drawing it, I wasn't able*

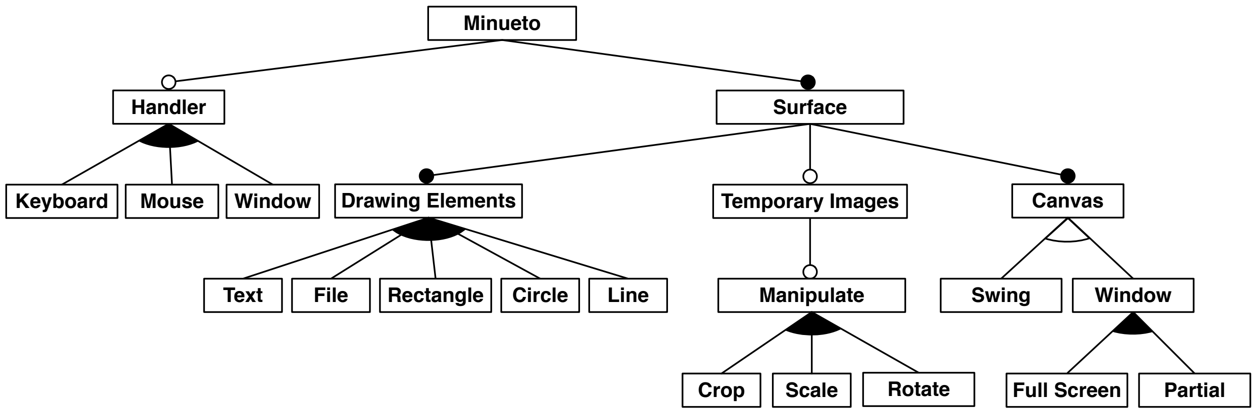


Figure 4.3: The refined updated feature model of D1 elaborated during the interview ($D1_{2b}$)

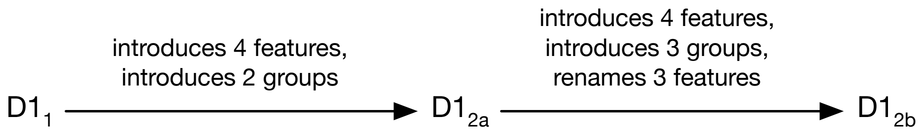

D1₁
introduces 4 features, introduces 2 groups
D1₂a
introduces 4 features, introduces 3 groups, renames 3 features
D1₂b

Figure 4.4: Evolution of feature models by D1

to find a smart way of saying this 'drawing stuff on screen', and then I didn't like the box saying 'things I can draw on the screen'. I found that it was redundant [...]"$_{\text{D1}}$.

When discussing the runnable example `ResolutionChangeDemo`, D1 explained that this example shows how to use and change between a window and a fullscreen window. Since this contradicts the XOR relationship of *Surface Types* in $D1_{2a}$ we asked for clarification from D1: "*Well they are not used at the same time, but they are used in the same program, but they [...] cannot be used at the same time*"$_{\text{D1}}$. This change is reflected in Figure 4.3 which shows the second feature model $D1_{2b}$ elaborated during the interview with D1. The second change relates to the distinction between different types of surfaces. The windows reflect the canvas on which elements can be drawn on, whereas a second type of surface are temporary images. A temporary image allows creating composites and draw them together as well as manipulate them.

Feature Model Evolution Summary The evolution of the feature models of D1 illustrates the difficulty of defining the granularity of a feature. During the interview we established a common understanding.

Figure 4.4 shows the evolution of changes between the three variants of feature models produced by D1. It shows that $D1_{2a}$ is a refinement of $D1_1$ and $D1_{2b}$ is a further refinement of $D1_{2a}$.

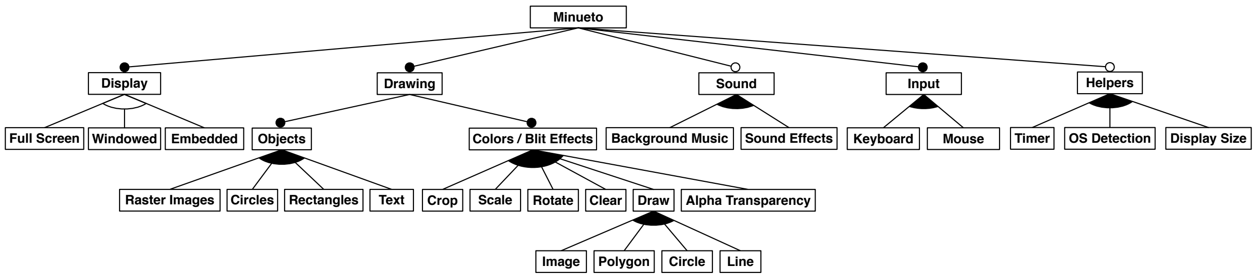


Figure 4.5: The initial feature model created by D2 ($D2_1$)

4.4.2.2 Feature Models of D2

The initial feature model of D2 ($D2_1$), shown in Figure 4.5, was more detailed. The reason could be that—as we outlined earlier in Section 4.4.1—D2 used the API reference to deduce the features and their relationships. Therefore, D2 was looking at all classes and methods provided by the Minueto API. However, the presence of the feature *Sound* indicates that this part was added from memory/knowledge as there are no classes and operations related to sound support in the API reference. The example `FireInTheSky` has sound, but it is not provided by Minueto. As D1 explained, it was given as an optional module "*[...] because at that time when we built it, the virtual machine didn't really do sound very well. So, it was like a ticking time bomb [...]*"$_{\text{D1}}$.

Comparing $D2_1$ to our feature model (see Figure 3.1) shows that it has some similarities. One notable difference is the additional details identified by D2 in the feature group *Drawing*. This again raised the topic of granularity, whether such things are "*useful to call out or not*"$_{\text{D2}}$. For instance, both the *Draw* and *Objects* features provide shapes to draw. However, there is a difference between the two because "*of the way the library is constructed [...] everything is extending from `MinuetoImage`, so even if you make like a rectangle or a circle, they're still effectively just images on the screen. And so I was trying to differentiate the fact that you can have an object that's a circle or rectangle that [is] easy to move around vs. using the draw method to [...] paste things together*"$_{\text{D2}}$. `MinuetoImage` has sub-classes for different shapes as objects, however, it is not an abstract class and by itself represents an empty image. It has *draw* methods to draw different shapes into the image enabling the composition of images. In addition, it has transformation methods to manipulate an image. However, due to the sub-classing, these methods are available for all the object shapes. While D2 was explaining that part of the feature model, he noted that he "*would move the draw one back to the objects side, because it's about [...] what you can put on the screen*"$_{\text{D2}}$.

In terms of possible feature configurations, D2 sees *Input* as a mandatory feature, the reason being that he thought about "*what you would require to use to make something useful vs. what*

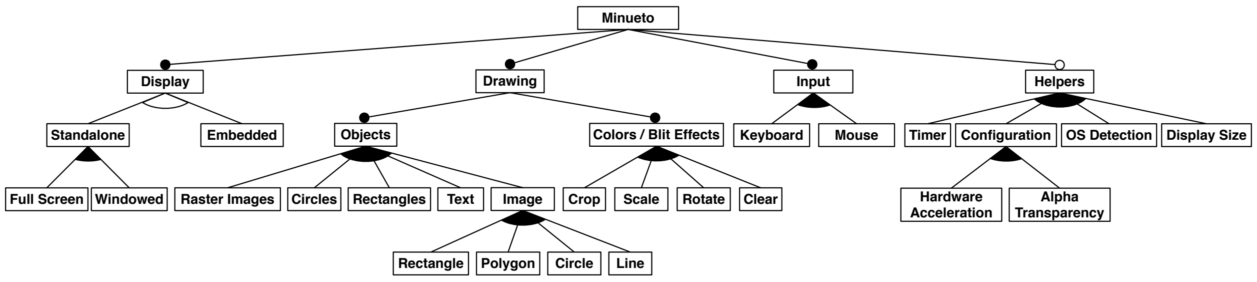


Figure 4.6: The updated feature model of D2 elaborated during the interview ($D2_2$)

isn't"$_{\text{D2}}$. While the examples do not always use handlers, most regular applications (such as games) created with Minueto usually make use of input handling. When asked whether D2 considers handling of window and window focus events as a feature, D2 explained that it could be a feature under *Input*, but that he is expecting that this is always used because one should know about the window state. Discussing hardware acceleration which the API allows a user to enable specifically, D2 responded that it was something one does once to set-up the project but confirmed "*that's optional too, so [...] it's probably worth calling out to have [...] in the helpers [...] something about [...] the different configuration pieces*"$_{\text{D2}}$.

The discussed changes to $D2_1$ are shown as $D2_2$ in Figure 4.6. *Hardware Acceleration* and *Alpha Transparency* are explicitly shown as features within the *Helpers* feature group. Furthermore, D2 also confirmed that a Minueto application could switch between fullscreen and windowed mode, but that the Swing integration (named *Embedded* in $D2_1$ and $D2_2$) cannot be combined with any other window type. However, both participants did not know whether the feature model describes the application at runtime—where only one window can be used at a time—or during development time. Both assumed the former, and hence, they used an XOR relationship at the beginning. Upon clarification by the investigator that the feature model describes what is available to the user of the API at development time, the additional grouping was introduced to allow usage of both window modes. This was again triggered when discussing the runnable example `ResolutionChangeDemo` which shows switching between a windowed and a fullscreen mode.

Feature Model Evolution Summary The initial feature model of D2 is very detailed and covers almost the complete API. As such, the changes between $D2_1$ and $D2_2$ are mainly restructuring of the groups and their constraints (e.g., in the *Display* feature group).

Figure 4.7 shows the evolution between the two feature models. The reason of $D2_1$ containing two additional features is the feature group *Sound*. However, these are not actual features of the Minueto framework, but rather optional code provided in one example. Disregarding these features, $D2_2$ provides a refinement over the initial feature model $D2_1$.

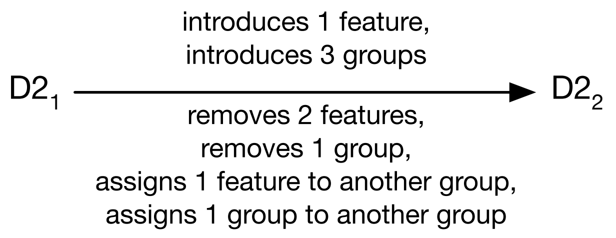


Figure 4.7: Evolution of feature models by D2

Table 4.1: Feature Model Metrics of $D1_{2b}$ and $D2_2$

Feature Model	Features	Groups	Leafs	Mandatory	Optional	OR-groups	XOR-groups	Configurations[3]
$D1_{2b}$	22	8	14	3	3	4	1	4464
$D2_2$	32	10	22	5	1	7	1	3948

Comparison of Feature Models Now that the feature models of both D1 and D2 are established, we can compare them with each other. We use the final versions $D1_{2b}$ and $D2_2$. Table 4.1 shows the metrics of the two feature models. The two feature models have 21 features in common. The numbers show the higher level of granularity chosen by D2. The higher granularity resulted in the *Helpers* feature group and the additional sub-features of *Image* (see Figure 4.6). The only feature contained in $D1_{2b}$ that is not in $D2_2$ is the feature *Window* for handling window events.

To compare the possible feature configurations, we use the 21 common features between $D1_{2b}$ and $D2_2$. The difference in configurations results from the fact that D2 chose input handling to be mandatory, whereas for D1 it is an optional feature. As we discussed above, D2 based these constraints on the fact that when one wants to create a game input handling is always used in order to create something meaningful. However, the examples show usages of Minueto with no event handling at all. Similarly, D2 marked the group feature (named *Colors / Blit Effects*) of the graphical object transformations as mandatory. For D1 it is an optional feature, however, it is grouped under the feature *Temporary Image*.

4.4.2.3 Investigator Feature Models

Our first Minueto feature model which we describe in detail in Section 3.5 is referred to as I_1 (re-shown for convenience in Figure 4.8). This feature model was shown to both participants during their interviews with the intent of gathering feedback on its completeness and its accuracy. Our

[3]Disregarding differing features, i.e., only based on the common features of the feature models.

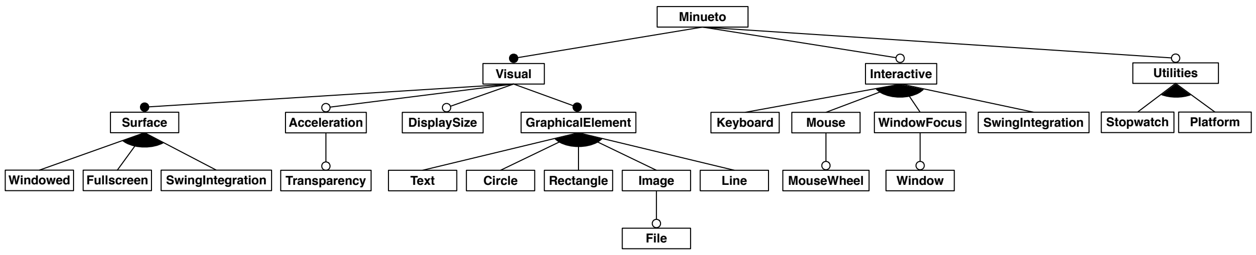


Figure 4.8: The initial feature model I_1 elaborated by the investigator

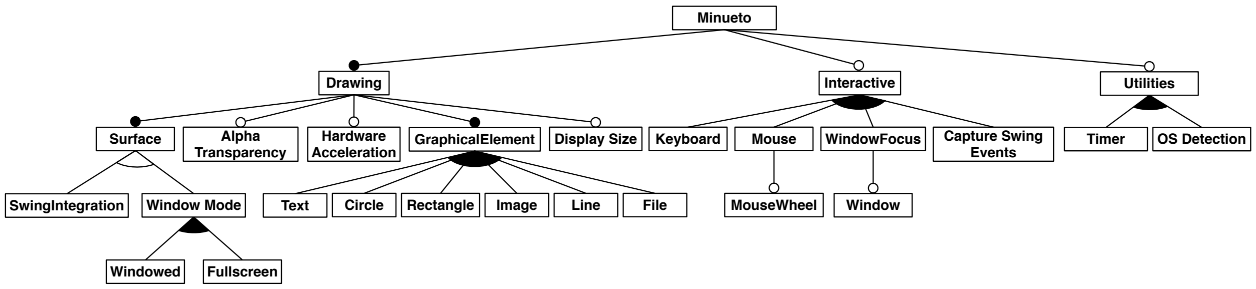


Figure 4.9: The updated feature model I_2 based on feedback from the participants

theory was that there is not only one correct feature model but many different possible variations with varying degrees of granularity.

The corrected feature model I_2 is shown in Figure 4.9. Most of the features were agreed on as correct by both participants. For example, on the fact that *WindowFocus* is a feature (under *Interactive*), D2 mentioned: "*I* d*o like the WindowFocus being there as part of the Interactive thing, [...] that kind of makes sense and I see [...] why they're all there, because they're all [...] the different handlers you can attach to the window*"$_{\text{D2}}$.

The biggest structural change relates to the exclusion of *SwingIntegration* from the other two window modes, as described earlier. Furthermore, our understanding of hardware acceleration being required when using transparency was incorrect. While the documentation of the `enableAlpha` method states that "*this will dramaticly slow down the application unless the JVM uses some kind of acceleration*", it is nevertheless possible to use transparency without acceleration. From the documentation it looks like they should always be used together because in addition to the above comment it is always used in combination with hardware acceleration in the examples. However, D1 admitted that he "*didn't want the students to do something that was not smart*"$_{\text{D1}}$.

D1's main critique about our feature model was the naming of some features: "*[...] they are one words, and then again, I come back to feature models and it's things you can do and [...] we're* *describing in one word [...] things you can do*"$_{\mathrm{D1}}$. He also rejected the name *Drawing* as an alternative to *Visual:* "*I tell the guys in the team in their documentation [...] that if you're describing things with one or two words, then what the person gets out of it is gonna be his interpretation of what the one or two words mean*"$_{\mathrm{D1}}$. Similarly, D2 rejected the feature *SwingIntegration*—sub-feature of *Interactive*—in part because of the duplicated name (with the sub-feature of *Surface*). While feature models do allow different features to have the same name, it can be confusing because it could appear as the same feature appearing twice in a tree. The other reason D2 rejected this feature was that the documentation does not mention any API element related to handling Swing events. Upon further investigation, this is due to the relevant classes being excluded within the Ant build script to generate the API reference (Javadoc) for Minueto. The reason we detected this as a feature was that we also looked at the source code in addition to the API reference and examples. In addition, based on what we learned about Minueto during the interviews, we moved the feature *File* (to load an image from a file) to be a child under *GraphicalElement* because it can be used independently of *Image* (which provides a blank canvas).

Finally, as a result of the feedback, several feature names were improved to better reflect what functionality the features provide, i.e., *Alpha Transparency* (*Transparency* beforehand), *Hardware Acceleration* (*Acceleration*), *Capture Swing Events* (*SwingIntegration*), and *Timer* (*Stopwatch*).

Feature Model Evolution Summary Since no features of I_1 were rejected by any of the participants, and no missing features were detected, there are no feature differences between I_1 and I_2. All changes relate to either a restructuring or renaming. However, the fact of the higher granularity of graphical elements was brought up by D2. If we were to provide a higher granularity for graphical elements and specifically list the different transformation operations as features, this would constitute a refinement over I_1.

4.4.2.4 Feature Model Comparison

The metrics of all feature models are shown in Table 4.2. The rows in bold highlight the final feature model version of the participants and the investigator. The number of feature configurations is based on the common features among the three final feature model versions. In total there are 17 common features. The lower number of configurations for $D2_2$ is mostly due to the fact that D2 considered the *Input* feature mandatory.

In summary, the differences mainly relate to the different levels of granularity. For instance, both D1 and D2 have the object manipulation methods as separate features, whereas I_2 does not. Conversely, our feature model has a finer granularity for event handling. Finally, D2 and the investigator both identified auxiliary features (grouped under *Helpers* in $D2_2$ and *Utilities* in I_2) which

Table 4.2: Feature model metrics for all feature models (rows in bold highlight the final feature model version of each person)

Feature Model	Features	Groups	Leafs	Mandatory	Optional	OR-groups	XOR-groups	Configurations[4]
$D1_1$	10	3	7	1	1	2	0	–
$D1_{2a}$	16	5	11	3	1	2	1	–
$\mathbf{D1_{2b}}$	**22**	**8**	**14**	**3**	**3**	**4**	**1**	**992**
$D2_1$	32	9	23	5	2	6	1	–
$\mathbf{D2_2}$	**32**	**10**	**22**	**5**	**1**	**7**	**1**	**564**
I_1	26	10	16	3	7	4	0	–
$\mathbf{I_2}$	**27**	**9**	**18**	**3**	**7**	**4**	**1**	**1008**

are not present in $D1_{2b}$.

4.5 Discussion

This section summarizes the interesting insights gained from the discussions with the participants. We discuss these in the following subsections.

4.5.1 Configurations provided by Feature Model

It is important that the feature model does not restrict the possible configurations of the framework. This should even be the case when there are some configurations that are not good choices in most of the cases. The configurations should only provide what is needed. The information about how the decision of feature choices affect certain high-level goals can be encoded in an impact model.

One such example is the discussion relating to *Alpha Transparency* and *Hardware Acceleration* which we had initially described as the former requiring the latter. However, as both participants pointed out this is not the case. The fact that the performance is negatively impacted when using only Alpha Transparency could be reflected in an impact model describing the goal *Increase Performance*.

4.5.2 Naming of Features

Accurately describing functionality in one or a few words is very difficult. D1 elaborated that "*the most senior Java programmer we have at the company says his most difficult challenge he*

[4]Disregarding differing features, i.e., only based on the common features of the feature models $D1_2b$, $D2_2$, and I_2.

has at work is naming stuff. He says, finding a proper name for a feature or functionality or something [...] is often more complicated than actually implementing it”$_{\text{D1}}$. As D1 explained, describing something with one or two words requires the reader to interpret what the author meant by it. Interestingly, naming was not a topic that was discussed with D2. The names of features were mainly deduced from the API element names. As a result, D2’s and our feature model have a lot of the same feature names.

D1 reasoned that a feature does not describe a state or an object and suggested to use verbs in the name. This is an interesting suggestion and feature models generally support longer feature names. In certain cases, features describe what can be accomplished with the functionality. For example, the feature *SwingIntegration* as a sub-feature of *Interactive* allows capturing Swing events within Minueto. Based on this name, D2 did not understand what functionality it provides. As a result, we renamed it to “*Capture Swing Events*”. An example where one word can sufficiently describe functionality is the class `MinuetoStopwatch`. In our first feature model we named this sub-feature of *Utilities* accordingly *Stopwatch*. Interestingly, both participants pointed out that the functionality that it provides is a timer (for instance, to calculate the frame rate), and suggested *Timer* as the name for this feature.

4.5.3 Feature Model Granularity

The decision on what constitutes a feature and which granularity to use is not clear. This is reflected in the choice and granularity of features of the two participants. For instance, D1 chose a high-level view of Minueto focussing on the main functionality that Minueto provides. D2, however, considered most of the functionality provided by the API, and specifically chose the drawing and manipulations of graphical elements as the area where more detail is necessary. Contrary to our own feature model, D2 also split the image transformation methods into single features, i.e., each feature providing a single transformation method.

The finest granularity that is possible is bound by the API. For example, if different parameters relate to different features, the number of parameters confine the maximum level of granularity. The granularity most likely is also dependent on the size of a framework in order to limit the number of features and to maintain concise feature models that do not overload the user. It remains to be investigated further which granularity level is helpful to the user.

4.5.4 Mapping from Feature to API

The question of how features are mapped to the API, i.e., which parts of the API belong to which feature, is dependent on the granularity of the feature model. Both participants mentioned that a feature sometimes maps to a class, and sometimes to one or more functions. For D1, the correct way is to “*[map] more to function then to class, because when we’re talking about features, for* *me, I'm thinking more in terms of things you can do, more than things you can have*"$_{\text{D1}}$, unless the feature contains all methods of the class. In addition, D1 confirmed that sometimes methods in an interface indicate the presence of a certain feature and as such belong to a different feature instead of the feature the class is defined in.

4.5.5 Applicability of Concernification

It is important to verify whether the proposed concern interface for frameworks is practical and could potentially help users in understanding the framework. As D2 pointed out during the interview, "*if a developer was asking quickly 'ok, what can I do with Minueto?' I think this would give them a good overall representation of the capabilities of the system*"$_{\text{D2}}$. D1 saw the use of concernification heavily on the training aspect of developers that need to familiarize themselves with a framework they do not know yet. From his own industrial experience, he remembered that in "*a lot of jobs I've come into the company and they've had some very elaborate framework. And the expectation is within a week or so I'm functional in that framework*"$_{\text{D1}}$. He mentioned that documentation and training is an issue in his company. Because a lot of frameworks used in industry are large and complex, understanding them is difficult. As such, he does not see it as helpful, however, more in terms of learning a framework: "*one of the things I want to bring is like, how can this be used in industry. I think in that term there's a very powerful use case. In large applications, in being able to build long term and stuff that has like 300 classes and 5000 features and so on, I'm not as sure, but as a tool for learning a framework or something that's in place and so on, this is beautiful.*"$_{\text{D1}}$.

4.6 Study Limitations and Threats to Validity

According to [104] one of the threats to validity in qualitative studies is *representativeness.* One perspective of the *representativeness* is the choice of the participants of our study on Minueto that we interviewed. The two participants are the two developers of Minueto. We looked at the commit history of the Minueto Subversion repository and found that the two participants are the only committers to the Minueto framework and its samples. Another perspective of *representativeness* is the choice of Minueto. Our study only focussed on one framework. While this framework is mostly used by university students for course projects, its conciseness allowed us to study it in more detail. To mitigate and confirm whether the findings of this study also apply for other frameworks, in Chapter 6, we examine a large framework that is widely used in industry in the evaluation of our algorithm to automatically concernify a framework.

We had no influence on how thorough the participants created the initial feature model. They were provided with a short explanation of feature models along with a small car example feature model and a link for further reading. Coming up with the features and organizing them within the feature model was completely up to the participants. They were not shown any feature model of Minueto beforehand to ensure that they had an unbiased view on their framework.

We performed interviews to mitigate the questions of completeness and feature model understanding. During the interviews it became apparent that certain feature model concepts were unclear, which resulted in different interpretations of what the feature model describes. One participant was unsure whether the feature model allows different sub-trees to create new groupings. The other participant was unsure how to describe a feature in one group that could apply to all features in another group (e.g., handling of window focus events can be done in all kinds of windows).

One problem was that of what a feature is. Therefore, we proposed to the participants to regard a feature as something that the user can choose to use or not use. This revealed some features that the participants had not considered as such before. Another problem revolved around whether the feature model describes what can be used of the framework at run-time (at the same time), or whether what can be used in the same application. This affected only the relationships, but not the features.

The interviews allowed us to ask clarification questions on the feature models and address the issues in understanding the feature model notation and its possibilities. Therefore, we elaborated modified versions of the participants feature models.

4.7 Summary

This chapter presented the user study we conducted with the two developers of Minueto. The developers were asked to each create their own feature model of Minueto first. We then conducted semi-structured interviews with them to find out which information they used when elaborating the feature model. Furthermore, we validated the accuracy of our manually created feature model of Minueto by comparing it to the ones elaborated by the developers. The results confirm our theory of the use of different information that is available from a reusable framework and its API. This includes the inheritance hierarchy, the usage of the API in runnable examples, and cross-references.

The understanding of what a feature is and the granularity of features heavily influences the feature model. While one developer had a more high-level feature model, the other developer had a finer granularity. The results show that the granularity can depend on the specific feature in question. For some features, a coarse-grained granularity is enough, whereas for other features fine-grained granularity makes more sense. For example, Minueto allows manipulations of graphical objects (such as rotating, scaling, etc.). While we did not identify these as specific features, the second developer identified them as features in the initial feature model, and the first developer identified them during our interview. In addition, both developers in large agreed with our identified features and their organization in a feature model and provided minor corrections. Interestingly, our feature model and that of the second developer share similarities in their structure and the features.

To help a framework designer in creating an initial concern interface for a framework, the next chapter describes an automated concernification algorithm that makes use of the information from the framework's API and usage of the API in runnable examples.

5 Automated Concernification

The previous chapters showed that concernification is not trivial to do, even for developers of a framework. In order to support developers in concernifying their framework we discuss in this chapter an automated concernification algorithm that produces an initial concern interface. The result can then be further fine-tuned and adjusted by the developer who as the domain expert has the knowledge of the framework's features and constraints.

We begin by providing a brief overview. We then establish guidelines (information) that the algorithm makes use of (Section 5.2) and define the specific information that is available of the framework (Section 5.3). We then establish hypotheses in Section 5.4 that our algorithm is based on. There are two kinds of hypotheses, those that we assume (require) to always hold, and those that depend on the quality of the examples provided with a framework. Section 5.5 explains in detail the algorithm and shows a small example that helps the reader to follow along the intermediary results during the process. Section 5.6 describes the implementation of the algorithm. We conclude this chapter with a summary in Section 5.7.

5.1 Overview

Ideally, a concern interface is developed together with a framework. However, existing frameworks do not come packaged with a concern, and hence we need to minimize the effort that is required for the framework developers to create a concern interface for the framework. Also, expert framework users that wish to use a framework at the modelling level might want to concernify a framework on their own. Therefore, our goal is to provide support for creating an initial concern interface for a framework using an automated approach. This approach needs to automatically find user-relevant features (step 1a in Section 3.4) for the framework, organize them in a feature model (step 1b), and decompose the framework's API (customization and usage interface) across the features (steps 2a and 2b).

Because there is no single correct feature model—the same set of feature relationships can be expressed with many different feature models—the automatically generated feature model is presented to the developer or expert user, i.e., the domain expert, who is then able to make adjustments.

5.2 Guidelines

Our approach makes use of runnable code examples that often come packaged with the framework to showcase how the framework is used. The intuition behind this is that code examples exemplify how to use a specific feature or a subset of the features of the framework. Research has identified examples as an important artefact that developers frequently use to learn how to use an API [71,91, 92]. This intuition was confirmed by the developers of the Minueto framework (see Section 4.4.1) which we used as a concernification example (see Section 3.5). Ideally, each code example uses the entire usage interface of a feature (and potentially other features) in order to be able to detect that they are part of the same feature and belong together.

The combination of the framework API and its runnable code examples allows us to use the following information:

- **Object-Oriented Hierarchy of the Framework:** Subclasses and classes of the framework that implement an interface are potential candidates for sub-features, because they add to the superclass or provide a specialization.

- **Cross-references in the Framework API:** A cross-reference occurs when a type (class) is being used as a parameter or return type in a public operation of another class. A cross-reference from class *A* to *B* indicates that when using class *A*, class *B* must also be used.

- **Code examples exemplifying Framework Use:** Code examples directly showcase the use of the framework API. Mostly, this involves small, typically minimal examples that show how to use a particular feature, or a set of features together. Code examples showcase the customization of a framework by extending or implementing classes or interfaces, as well as usage of the usage interface (API) by instantiating classes and calling operations.

- **Framework Modularization:** Packages are generally used to structure classes with common or related functionality. Classes within the same package could indicate different variations of the same functionality, or a grouping of classes to accomplish one functionality/feature. Similarly, some programming languages (such as Java) provide the ability to add inner classes to a class. We consider this a form of structuring as well.

5.3 Definitions

This section precisely defines the information that is available to the concernification algorithm. Assuming that there exist runnable code examples that are provided along with the framework, the basic information building blocks are:

- **Framework code element** e: A framework consists of different elements, such as a class (including exception), interface, operation, attribute or package. We consider all of those elements to be code elements regardless of their visibility.
- **Runnable code example** c: Runnable code examples that showcase how to use the framework in different ways are often provided along with the framework.
- **Set of code elements** API: The user interface of the framework consists of a set of code elements e that are visible, i.e., they can be used by the user.
- **Set of relationship types** T: There can be different types of relationships between code elements. These types are:
  - *is subclass of*,
  - *implements interface of*,
  - *is operation of*,
  - *is attribute of*,
  - *has crossreference to*,
  - *is inner class of*,
  - *is in package,*
  - *is sub package of,* and
  - *throws exception*.

The goal of our algorithm is to determine the features of a framework, and associate with each feature the corresponding parts of the API. We assume that frameworks consist of a set of features with more than one user-perceivable feature. Some of these features are mandatory, whereas other features can be used or not depending on the user's need. We therefore define a feature f as a user-perceivable feature of a framework. The usage interface for a given feature f is defined as the relation $api\colon f \mapsto \{e\}$ on the usage interface of the complete framework API. $api(f)$ provides the set of code elements whose elements $e \in API$ and satisfies that an e only belongs to one feature, ensured by the following properties,

- $api(f) \subseteq API$
- $\bigcup_{f} api(f) = API$
- $\forall f_1, f_2 \ (api(f_1) \bigcap api(f_2) = \varnothing)$.

The last property also implies that each feature adds something meaningful, i.e., something new. Therefore, API elements required by multiple features should be put into a shared parent feature or a shared parent realization model. Due to the composition in CORE (see Section 2.4.3), these API elements will be available to the user whenever any of the child features is selected.

Using these basic elements we establish the $uses$ relationship which describes that one or more elements e of $api(f)$ are used by a code example. In addition to these elements we define the relation $usedIn$: $e \mapsto \{c\}$ which provides the set of code examples C a given code element e is used in.

5.4 Hypotheses

In this section we establish hypotheses about the intent of the framework designer, as well as the quality of the framework's API and the framework's code examples. The first 5 hypotheses are common principles of good API design that we always expect to be true. They are:

1. All API code elements are intended to be used by the user.
2. Each code example provided with the framework shows how to use a specific feature (or a subset of the features) of the framework.
3. A code example might cover more than one feature.
4. All code examples make *correct* use of the framework, i.e.,
   $\forall c(c\, uses\, f \Rightarrow c$ achieves something meaningful with $f)$
5. If the framework defines abstract classes/interfaces, and provides
   (a) also two or more implementations of those classes/interfaces, then the implementation classes represent different features/behaviour. The user should instantiate the desired one(s) to configure the framework. This represents closed variability and should be included in the variation interface.
   (b) no concrete implementations, then the user is supposed to implement the subclass/interface and pass the implementation to the framework. This represents open variability and should be included in the customization interface.

(c) only one implementation of an interface, then the user can either use it or provide his own implementation. The "own implementation" variant should be represented in the variation interface as an optional feature with a corresponding customization interface.

Ideally, if a framework consists of *n* independent features, then there should be *n* code examples, i.e., there would be a 1-1 correspondence between code examples and features. If each one of the code examples would use *all* code elements of the API of the feature it demonstrates, then it would be easy to determine the subset of the framework's API related to the feature. Unfortunately, features are rarely independent. It is also unrealistic to assume that "real-world" frameworks come with the perfect examples that showcase all features individually, provide examples of all possible feature combinations, and make use of all the API elements related to every feature.

To account for the imperfectness of the real world, we define an additional set of hypotheses that quantify how well the code examples cover the features and the API provided by the framework, and how focussed the code examples are. The hypotheses are then used in the proposed concernification algorithm to enable/disable different processing steps:

H1. Every user feature is used in at least one code example:
$\forall f \; (\exists c \mid c \; uses \; api(f))$

(a) There is a code example that demonstrates every possible combination of features. This allows the algorithm to detect XOR relationships between child features.

(b) For every feature, there exists a code example using the feature that uses the minimal number of additional features. This allows the algorithm to detect OR relationships between child features.

H2. If f_1 and f_2 are two distinct user features, then there exists at least one code example in which they are not used together:
$f_1 \neq f_2 \implies \exists c \mid c \; uses \; api(f_1) \wedge \neg((c \, uses \, api(f_2))$

H3. All code elements of the API related to a user feature are used in at least one code example.
$\forall f \, (\forall e \in api(f) \mid usedIn(e) \neq \emptyset)$

H4. Two code elements e_1 and e_2 belong to the same feature, if both code elements are used in the exact same set of code examples.
$(\exists e_1, e_2 \mid usedIn(e_1) = usedIn(e_2)) \implies \exists! f \mid e_1, e_2 \in api(f)$. As such, every element belonging to the API of a given feature is used in the same set of code examples.
$\forall f \, (\forall e_1, e_2 \in api(f) \mid usedIn(e_1) = usedIn(e_2))$

5.5 Algorithm

This section describes the algorithm in detail. We begin by describing the input to the algorithm. Then, we provide a general overview of the algorithm and describe each step of the algorithm in detail after. We use a small example based on the Minueto framework to exemplify the algorithm by showing the intermediate results during the process.

5.5.1 Input

As a preparation, the set of code elements CE of the framework along with the set of relationships R between them need to be determined. Furthermore, the set of code examples C must be determined. In addition, the usage information of which example makes use of which code elements is required. Once all this information has been gathered, the input to the concernification algorithm is comprised of:

- $CE = \{ce\}$, the set of code elements, regardless of their visibility,
- $R = \{(ce_s, ce_t, t)\}$, the set of relationships between code elements given as a tuple with the source code element ce_s, target code element ce_t and the type of the relationship $t \in T$,
- $C = \{c\}$, the set of code examples,
- $usedIn : ce \mapsto \{c\}$, the relation from a code element to the set of code examples it is used in,
- $options$, the options describing which of the additional hypotheses are assumed to be holding for the examples.

In the following sections describing the algorithm, it is assumed that a class contains its elements (i.e., attributes and operations) already, and the hierarchy information is provided within the classes.

5.5.2 Overview

To determine the feature model of a framework, the concernification algorithm works with a directed acyclic graph (DAG) where the nodes represent potential features and the edges relationships between them. The DAG used in our algorithm consists of a set of nodes V and a set of edges E. The graph contains a property $roots$ referring to the root nodes in the DAG. Nodes represent the potential features and are processed in the algorithm based on the available information of the framework and examples. Each node stores the following information:

- **classes**: The set of classes that belong to this potential feature. Each class contains the elements that belong to this class within the potential feature.
- **examples**: The mapping $e \mapsto C$ from code elements to the set of examples the element is used in. The union of examples for all code elements represents all examples the potential feature is used in.
- **name**: A name for the node. This is mostly important for grouping and package nodes without any classes. The names of nodes with classes can always be derived from the class names.

Each edge represents a multiset (bag) of relationship types T representing the type of dependencies this edge represents. The possible types of relationships are:

- **inheritance**: Indicates an inheritance relationship between two nodes. This represents both the *is subclass of* and *implements interface of* relationships.
- **containment**: Indicates that the elements in the source node belong to the same class as the target node. This represents both the *is operation of* and *is attribute of* relationships.
- **crossref**: Indicates that an element from the source node references an element from the target node. This represents the *has crossreference to* relationship as well as the *throws exception* relationship.
- **structural_grouping**: Indicates that the source node belongs to the target node, the latter being a node representing a package or outer class. This represents the *is inner class of*, *is in package* and *is sub-package of* relationships. For brevity, from now on we call it *grouping*.

In the first phase, the DAG is created and populated with individual trees for all hierarchies. In the second phase, the trees are connected within the DAG to form one connected graph and then populated with all the information available. In the third phase, the DAG is simplified by removing edges in order to reduce it to a tree that is used as the final feature model structure. In essence, the code-centric DAG from the first phase and second phase needs to be converted into a user-centric tree that represents the feature model from the user's perspective.

Initially, the DAG is populated with separate directed rooted trees, one for each inheritance hierarchy that is found in the framework code. Each directed rooted tree has an orientation towards the root (also *called anti-arborescence* or *in-tree*), i.e., edges are pointing in the inverse direction. This allows us to describe dependencies between two elements (potential features), such as, *X is a sub-class of Y*. A node is created for each class. The OO hierarchy is reflected using directed edges from the sub-class to the superclass, as well as for classes implementing an interface.

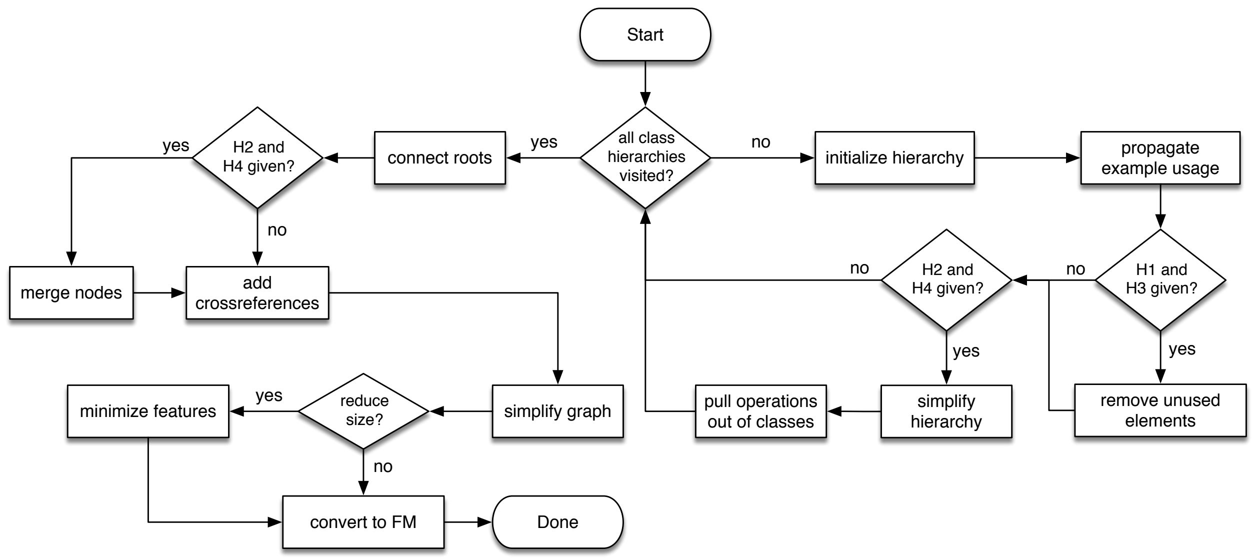


Figure 5.1: Overview of the Concernification Algorithm

At this point, the DAG consists of a disjoint union (forest) of individual directed rooted trees. The trees need to be connected in the second phase. Packages and outer classes can potentially provide a structural grouping of common or related functionality. Therefore, the separate trees are then connected to a single DAG based on the grouping relationships of their root element(s). Based on the usage information from the code examples, framework code elements with the same usage are combined into the same node. Operations that are part of the same node but with differing usage are separated. For example, operations of a class that are used in a subset of examples compared to the class itself indicate that the operation could potentially belong to a different feature. At this point, only the relationship type *crossref* has not been introduced into the graph yet. Cross-references between API elements introduce additional dependencies. If using an operation requires the use of another type, e.g., as a parameter to be passed when calling the operation, an edge of type *crossref* is required to indicate this dependency.

At the end of phase two, the DAG is populated with all available information. It is possible at this point that a node has more than one outgoing edge. In order for a feature model to be retrieved, the DAG needs to be converted to a directed tree. The goal of the third phase is to simplify the edges while maintaining the dependencies among API elements.

An overview of the steps of the concernification algorithm is shown as a flow chart in Figure 5.1. The first phase is shown on the right-hand side of the flow chart, the second phase on the top of the left-hand side, and the third phase is shown on the bottom of left-hand side. The figure also shows the hypotheses that must hold in order for a step to be executed.

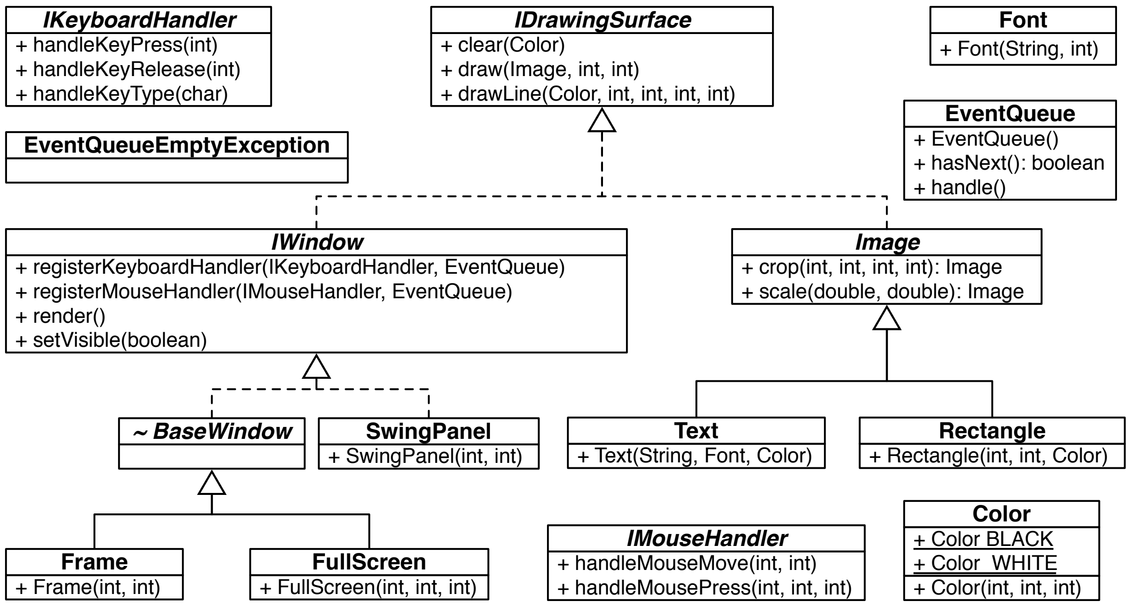


Figure 5.2: Class Diagram of Running Example

To aid the reader in understanding intermediate results between processing steps of the algorithm, we use a running example based on the Minueto framework (see Section 3.5 for more detail about Minueto). The example class diagram is shown in Figure 5.2. It contains a subset of API elements from the Minueto framework and shortened class names for brevity. Names of interfaces are prefixed with "*I*", and names of abstract classes are written in italic font. For the sake of this example, we assume that all hypotheses outlined in Section 5.4 hold.

The complete process overview of concernification is given in Algorithm 5.1 which shows the `concernify` function. Some of the steps are only enabled if the corresponding hypotheses hold. We first explain several auxiliary functions that are used by different steps in the following section. Following that, we describe each step in more detail with the reasoning on why this is done.

5.5.3 Auxiliary Functions

Within several of the algorithm steps, recurring functionality is used to merge and remove nodes. These auxiliary functions are defined as follows and will be used in the following sections that explain the individual steps of the algorithm in detail.

Algorithm 5.2 shows merging of two nodes. While merging is inherently symmetric, for convenience, the `toMerge` node is merged into the `base` node. Therefore, the properties of the nodes are merged. For classes this means that if the same class exists in both nodes, its contents are merged.

Algorithm 5.1 Concernification Algorithm

```
function CONCERNIFY($classes$, $R$, $C$, $usedIn$, $options$)
    $g \leftarrow$ create new DAG
    for all hierarchy $h \in classes$ do
        $root \leftarrow$ INITIALIZEHIERARCHY($g$, $h$, $R$, $usedIn$)
        PROPAGATEEXAMPLEUSAGE($g$, $root$)
        if $options.H1$ and $options.H3$ then
            REMOVEUNUSEDELEMENTS($g$, $root$)
        end if
        if $options.H2$ and $options.H4$ then
            SIMPLIFYHIERARCHY($g$, $root$)
            PULLOPERATIONSOUT($g$, $root$)
        end if
        $g.roots \leftarrow g.roots \cup root$
    end for
    CONNECTROOTS($g$, $R$)
    if $options.H2$ and $options.H4$ then
        MERGESIMILARNODES($g$)
    end if
    ADDCROSSREFERENCES($g$)
    SIMPLIFYGRAPH($g$, $C$)
    if $options.reduce\_size$ then
        REDUCESIZE($g$)
    end if
    return CONVERTTOCONCERN($g$)
end function
```

Algorithm 5.2 Merging Two Nodes

```
 1: function MERGENODES($g$, $base, toMerge$)
 2:     $base.examples \leftarrow base.examples \cup toMerge.examples$
 3:     $base.classes \leftarrow base.classes \cup toMerge.classes$
 4:     for all outgoing edge $e \in$ outgoing edges of $toMerge$ do
 5:         $t \leftarrow$ target of $e$
 6:         REPLACEEDGE($g$, $e$, $base$, $t$)
 7:     end for
 8:     for all incoming edge $e \in$ incoming edges of $toMerge$ do
 9:         $s \leftarrow$ source of $e$
10:         REPLACEEDGE($g$, $e$, $s$, $base$)
11:     end for
12:     $g.V \leftarrow g.V \backslash toMerge$
13:     if $g$ has cycle then                    ▷ handle any resulting cycle immediately
14:         $nodes \leftarrow$ find nodes involved in cycle
15:         $baseNode \leftarrow$ first node $\in nodes$
16:         for all node $n \in nodes \backslash baseNode$ do
17:             MERGENODES($g$, $baseNode$, $n$)
18:         end for
19:     end if
20: end function
```

Algorithm 5.3 Replacing an Existing Edge in the Graph

```
 1: function REPLACEEDGE($g$, $e$, $newSource$, $newTarget$)
 2:     if $\exists$ edge $x \mid (newSource, newTarget) \in g.E$ then
 3:         $x.T \leftarrow x.T \cup e.T$
 4:     else
 5:         $x \leftarrow$ new edge $(newSource, newTarget)$
 6:         $x.T \leftarrow e.T$
 7:         $g.E \leftarrow g.E \cup x$
 8:     end if
 9:     $g.E \leftarrow g.E \backslash e$
10: end function
```

Algorithm 5.4 Moving Node Properties to Another Node

```
1: function MOVENODEPROPERTIES($base, toMove$)
2:     $base.examples \leftarrow base.examples \cup toMerge.examples$
3:     $base.classes \leftarrow base.classes \cup toMerge.classes$
4:     $toMove.classes = \varnothing$
5: end function
```

All incoming and outgoing edges of the `toMerge` node are changed such that they are the source, and target respectively, of the `base` node. If an edge already exists, the relationship types are merged and the existing edge is removed from the DAG. At the end, the `toMerge` node can be safely removed from the DAG. The corresponding `replaceEdge` function is shown in Algorithm 5.3.

Whenever two nodes are merged, due to their edges being consolidated, it is possible that a cycle within the graph exists as a result. Any cycle needs to be removed as it violates the properties of a DAG. Since the nodes forming the cycle have a strong dependency between each other, we therefore merge those nodes, thereby breaking the cycle.

In certain cases, it is required to keep a node, but its properties are determined to belong to another node. This is the case when a node needs to be kept for grouping reasons to group the child nodes. Algorithm 5.4 defines the `moveNodeProperties` function.

Lastly, some features that are not used can safely be removed from the graph because they are not deemed relevant (e.g., they were not used in any example). Algorithm 5.5 defines the auxiliary function `removeNode`. In this case, for all nodes for which the removed node is an intermediary, their edges are connected to the removed node's parent(s). If the removed node is the root, any source node that does not have any other parent will become a root.

5.5.4 Initializing Hierarchies

The following subsections describe the individual steps of the concernification algorithm in detail. As a first step, the framework classes provided as an input are processed to create a DAG consisting of several disjoint directed rooted trees, where each directed rooted tree represents an inheritance hierarchy. As a reminder to the reader, the orientation of the trees are towards the root, i.e., the edges are directed towards the designated root node. The algorithm is defined in Algorithm 5.6. For each class in the code of the framework, a corresponding node is created. The OO hierarchy is replicated between nodes using edges with type *inheritance* (which represents both *subclasses* and *implements* relationships).

However, due to the fact that—depending on the programming language—multiple inheritance is possible, a node can be part of more than one hierarchy. For example, while Java allows inheri-

Algorithm 5.5 Removing a Node from the Graph

```
1: function REMOVENODE($g$, $node$)
2:     if ∃ outgoing edge of $node$ then
3:         for all outgoing edge $o \in$ outgoing edges of $node$ do
4:             $p \leftarrow$ target of $o$
5:             for all incoming edge $e \in$ incoming edges of $node$ do
6:                 $s \leftarrow$ source of $e$
7:                 REPLACEEDGE($g$, $e$, $s$, $p$)        ▷ replaceEdge removes $e$ from the graph
8:             end for
9:             $g.E \leftarrow g.E \backslash o$
10:        end for
11:    else
12:        for all incoming edge $e \in$ incoming edges of $node$ do
13:            $s \leftarrow$ source of $e$
14:            $g.roots \leftarrow g.roots \cup s$
15:            $g.E \leftarrow g.E \backslash e$
16:        end for
17:    end if
18:    $g.V \leftarrow g.V \backslash node$
19: end function
```

Algorithm 5.6 Creating the Initial Hierarchy for all Classes

```
1: function INITIALIZEHIERARCHY($g$, hierarchy $h$, $R$, $usedIn$)
2:     repeat
3:         $c \leftarrow$ current visited class of $h$
4:         $n \leftarrow$ new node for $c$
5:         $n.classes \leftarrow \{c\}$
6:         $n.examples \leftarrow usedIn(c)$
7:         $g.V \leftarrow g.V \cup n$
8:         for all super-class or interface $ce_t \in R \mid (ce_s = c$
9:                 and $\{is\ subclass\ of,\ implements\ interface\ of\} \ni t)$ do
10:            $s \leftarrow$ node for $ce_t$
11:            $e \leftarrow$ new edge $(c, s)$
12:            $e.T \leftarrow [inheritance]$
13:            $g.E \leftarrow g.E \cup e$
14:        end for
15:    until all classes in $h$ visited
16: end function
```

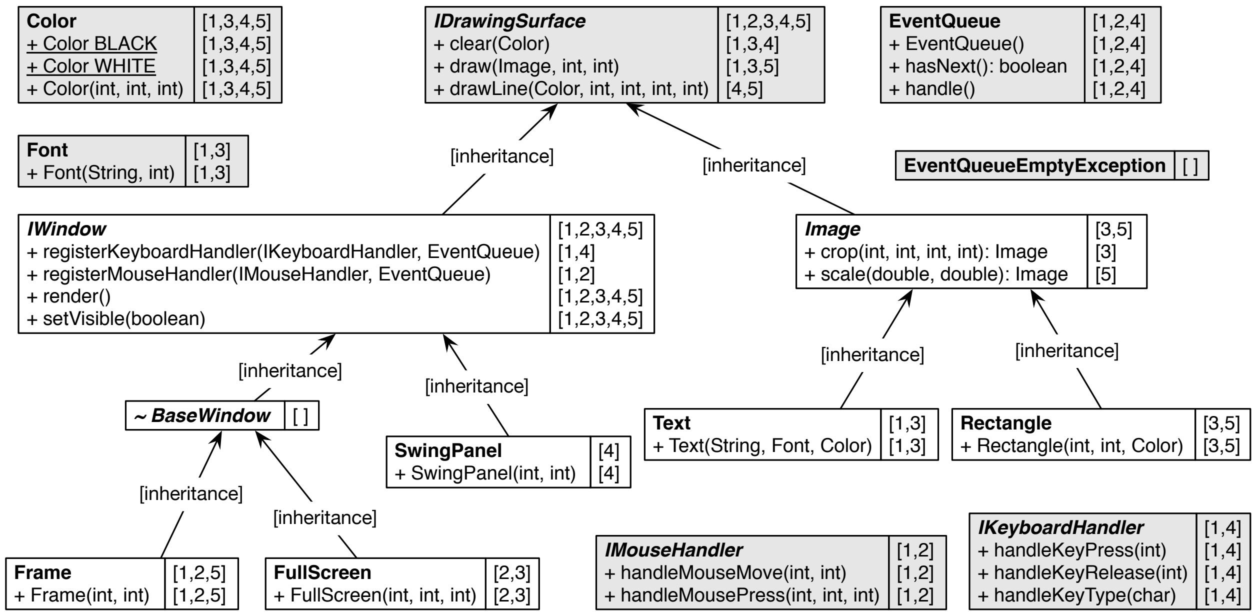

Color | [1,3,4,5]
+ Color BLACK | [1,3,4,5]
+ Color WHITE | [1,3,4,5]
+ Color(int, int, int) | [1,3,4,5]
Font | [1,3]
+ Font(String, int) | [1,3]
IDrawingSurface | [1,2,3,4,5]
+ clear(Color) | [1,3,4]
+ draw(Image, int, int) | [1,3,5]
+ drawLine(Color, int, int, int, int) | [4,5]
EventQueue | [1,2,4]
+ EventQueue() | [1,2,4]
+ hasNext(): boolean | [1,2,4]
+ handle() | [1,2,4]
EventQueueEmptyException | [ ]
[inheritance]
IWindow | [1,2,3,4,5]
+ registerKeyboardHandler(IKeyboardHandler, EventQueue) | [1,4]
+ registerMouseHandler(IMouseHandler, EventQueue) | [1,2]
+ render() | [1,2,3,4,5]
+ setVisible(boolean) | [1,2,3,4,5]
Image | [3,5]
+ crop(int, int, int, int): Image | [3]
+ scale(double, double): Image | [5]
~ BaseWindow | [ ]
SwingPanel | [4]
+ SwingPanel(int, int) | [4]
Text | [1,3]
+ Text(String, Font, Color) | [1,3]
Rectangle | [3,5]
+ Rectangle(int, int, Color) | [3,5]
Frame | [1,2,5]
+ Frame(int, int) | [1,2,5]
FullScreen | [2,3]
+ FullScreen(int, int, int) | [2,3]
IMouseHandler | [1,2]
+ handleMouseMove(int, int) | [1,2]
+ handleMousePress(int, int, int) | [1,2]
IKeyboardHandler | [1,4]
+ handleKeyPress(int) | [1,4]
+ handleKeyRelease(int) | [1,4]
+ handleKeyType(char) | [1,4]

Figure 5.3: Intermediate result after initializing hierarchies (roots of hierarchies shown in grey)

tance only from one class, multiple interfaces can be implemented. This means that a class that is a sub-class and also implements an interface will have two outgoing edges.

Furthermore, based on our hypotheses, we initialize a custom sub-class node for any class of the framework that was sub-classed within the examples. However, interface methods always need to be implemented in the classes implementing them. Hence, we only take those methods into account once where they are initially declared and attribute all example usage to this method. We apply the same for overriding methods in sub-classes.

Figure 5.3 shows the intermediate result after initializing all hierarchies for the running example. The root of each directed tree, representing a hierarchy, is highlighted in grey. The example usage is depicted in the second column of each node. In our example we considered five runnable examples in total. At this point, the DAG consists of several trees, one for each hierarchy.

It is possible that sub-classes have a super-class providing a shared implementation that is not visible (i.e., its visibility is not *public*). We consider such classes as potential groupings and maintain a node for them, i.e., they are part of a hierarchy. For example, this is the case for `BaseWindow` which is *package-private* and contains the shared implementation for the frame and fullscreen window options.

Algorithm 5.7 Propagating the Example Usage within a Hierarchy

```
1: function PROPAGATEEXAMPLEUSAGE($g$, $root$)
2:     repeat
3:         for all leaf nodes $l \in$ non-visited nodes of tree for $root$ do
4:             mark $l$ as visited
5:             $p \leftarrow$ parent node of $l$
6:             $p.examples \leftarrow p.examples \cup l.examples$
7:         end for
8:     until all leaf nodes visited
9: end function
```

5.5.5 Propagating Example Usage

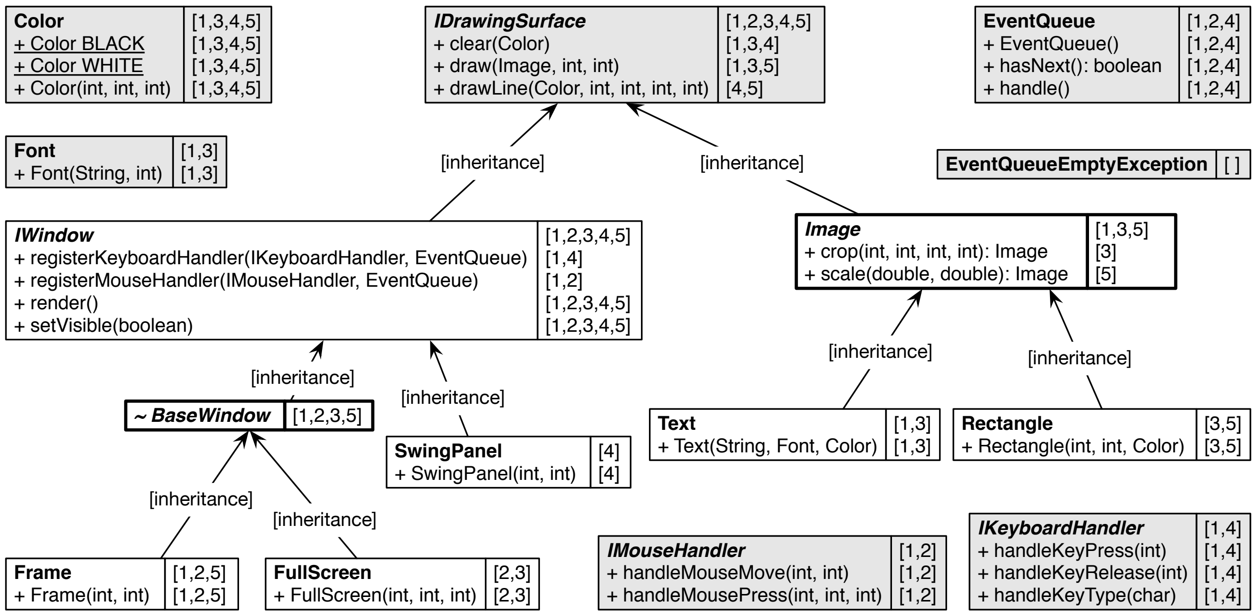
Each node in the DAG has its own example usage based on the usage of the corresponding class in the examples. However, all public methods and fields of a super-class are available in sub-classes. Therefore, we need to take into account that when using an instance of a sub-class also the super-class is used. We do this by propagating the example usage up within the inheritance hierarchy. Algorithm 5.7 shows the definition of the example usage propagation algorithm for a given hierarchy.

Figure 5.4 shows the intermediate result after the propagation. Updated nodes or edges are highlighted with a thick stroke. As a result of the propagation, `BaseWindow` and `Image` now contain the union of their own plus the example usage of their sub-classes.

5.5.6 Removing Unused Nodes and Elements

Algorithm 5.8 defines the step that removes nodes that only contain unused elements as well as unused elements within used nodes. If hypotheses H1 and H3 are assumed to hold for the given code examples, this means that every user feature and all of the code elements corresponding to their APIs are used at least once. Hence, if the DAG contains nodes that have only unused elements, the nodes can be removed since they do not contribute to the API of a feature. The same applies to unused elements in used nodes.

In the example, the class `EventQueueEmptyException` is the only node that is not used in any example. It is an unchecked exception that would only occur if someone incorrectly used the `EventQueue` class by calling `handle()` without ensuring beforehand that there is an event to handle in the queue. This is done using the `hasNext()` method. As a result, this node, and therefore the directed rooted tree of the hierarchy, is removed.



Color [1,3,4,5]
+ Color BLACK [1,3,4,5]
+ Color WHITE [1,3,4,5]
+ Color(int, int, int) [1,3,4,5]
IDrawingSurface [1,2,3,4,5]
+ clear(Color) [1,3,4]
+ draw(Image, int, int) [1,3,5]
+ drawLine(Color, int, int, int, int) [4,5]
EventQueue [1,2,4]
+ EventQueue() [1,2,4]
+ hasNext(): boolean [1,2,4]
+ handle() [1,2,4]
Font [1,3]
+ Font(String, int) [1,3]
[inheritance]
[inheritance]
EventQueueEmptyException [ ]
IWindow [1,2,3,4,5]
+ registerKeyboardHandler(IKeyboardHandler, EventQueue) [1,4]
+ registerMouseHandler(IMouseHandler, EventQueue) [1,2]
+ render() [1,2,3,4,5]
+ setVisible(boolean) [1,2,3,4,5]
Image [1,3,5]
+ crop(int, int, int, int): Image [3]
+ scale(double, double): Image [5]
[inheritance]
[inheritance]
[inheritance]
[inheritance]
~ BaseWindow [1,2,3,5]
SwingPanel [4]
+ SwingPanel(int, int) [4]
Text [1,3]
+ Text(String, Font, Color) [1,3]
Rectangle [3,5]
+ Rectangle(int, int, Color) [3,5]
[inheritance]
[inheritance]
Frame [1,2,5]
+ Frame(int, int) [1,2,5]
FullScreen [2,3]
+ FullScreen(int, int, int) [2,3]
IMouseHandler [1,2]
+ handleMouseMove(int, int) [1,2]
+ handleMousePress(int, int, int) [1,2]
IKeyboardHandler [1,4]
+ handleKeyPress(int) [1,4]
+ handleKeyRelease(int) [1,4]
+ handleKeyType(char) [1,4]

Figure 5.4: Intermediate result after propagating example usage (updated nodes highlighted with thick stroke)

Algorithm 5.8 Removing Unused Nodes and Elements

```
1: function REMOVEUNUSEDNODES($g$, $root$)
2:     repeat
3:         for all leaf nodes $l \in$ non-visited nodes of tree for $root$ do
4:             mark $l$ as visited
5:             if $l.examples$ is empty then
6:                 REMOVENODE($g$, $l$)
7:             else
8:                 for all class $c \in l.classes$ do
9:                     for all element $e \in$ elements of $c$ do
10:                        if $l.examples(e)$ is empty then
11:                            remove $e$ from $c$
12:                        end if
13:                    end for
14:                end for
15:            end if
16:        end for
17:    until all leaf nodes visited
18: end function
```

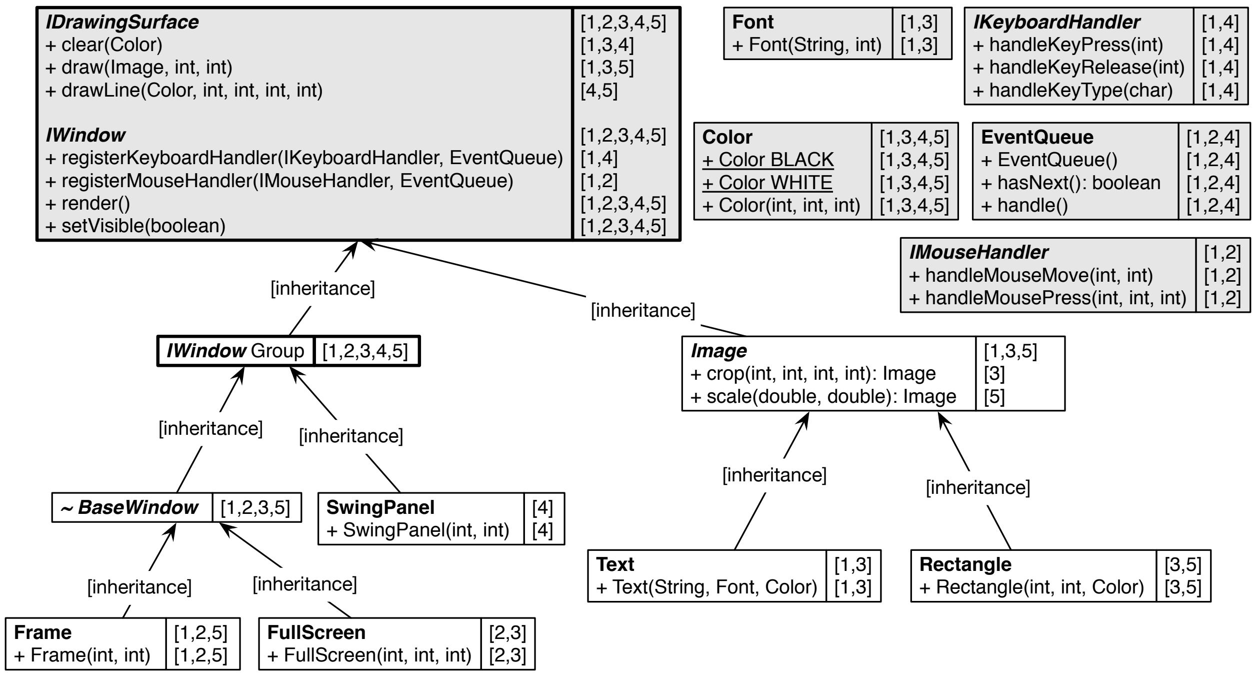
Algorithm 5.9 Simplifying a Hierarchy

```
1: function SIMPLIFYHIERARCHY($g$, $root$)
2:     repeat
3:         for all leaf node $l \in$ non-visited nodes of tree for $root$ do
4:             mark $l$ as visited
5:             $p \leftarrow$ parent node of $l$
6:             if $l.examples = p.examples$ then
7:                 if $l$ is a grouping for its children then
8:                     mark $l$ as potential grouping node
9:                     MOVENODEPROPERTIES($p$, $l$)
10:                else
11:                    MERGENODES($g$, $p$, $l$)
12:                end if
13:            end if
14:        end for
15:    until all leaf nodes visited
16: end function
```

5.5.7 Simplifying Hierarchies

At this point we have several directed rooted trees in the DAG which represent the individual inheritance hierarchies. Any class that is not within a hierarchy will be in a tree that only contains a single node. Hypotheses H2 and H4 state that code elements belong to the same feature if they are used in the same examples. Based on the example usage, we can therefore merge nodes with the same usage set. This requires the comparison of all nodes in the DAG. In order to improve the performance, we use this step to perform this simplification within each directed rooted tree that represents an OO hierarchy first. We only compare each child with their parent. This reduces the overall number of nodes in each tree, and thus reduces the number of comparisons required when merging nodes later. Algorithm 5.9 defines the hierarchy simplification step.

It is possible that a feature provides a logical grouping for its children. In such a case, the node itself needs to be kept. Its contents though can be moved to the parent node. One such case can be seen in Figure 5.5, which shows the state of our example after simplifying all hierarchies. The node representing interface `IWindow` provides a grouping for the different types of windows based on the inheritance hierarchy and example usage. The properties of the `IWindow` node were moved to its parent. Because it is empty, the grouping at least has an OR relationship with its children. At a later stage it can be determined, based on the indication of hypothesis H1a, whether the `IWindow` grouping can actually be used to determine the relationship of the children as XOR.



IDrawingSurface [1,2,3,4,5]
+ clear(Color) [1,3,4]
+ draw(Image, int, int) [1,3,5]
+ drawLine(Color, int, int, int, int) [4,5]
IWindow [1,2,3,4,5]
+ registerKeyboardHandler(IKeyboardHandler, EventQueue) [1,4]
+ registerMouseHandler(IMouseHandler, EventQueue) [1,2]
+ render() [1,2,3,4,5]
+ setVisible(boolean) [1,2,3,4,5]
Font [1,3]
+ Font(String, int) [1,3]
IKeyboardHandler [1,4]
+ handleKeyPress(int) [1,4]
+ handleKeyRelease(int) [1,4]
+ handleKeyType(char) [1,4]
Color [1,3,4,5]
+ Color BLACK [1,3,4,5]
+ Color WHITE [1,3,4,5]
+ Color(int, int, int) [1,3,4,5]
EventQueue [1,2,4]
+ EventQueue() [1,2,4]
+ hasNext(): boolean [1,2,4]
+ handle() [1,2,4]
IMouseHandler [1,2]
+ handleMouseMove(int, int) [1,2]
+ handleMousePress(int, int, int) [1,2]
[inheritance]
[inheritance]
IWindow Group [1,2,3,4,5]
Image [1,3,5]
+ crop(int, int, int, int): Image [3]
+ scale(double, double): Image [5]
[inheritance]
[inheritance]
~ BaseWindow [1,2,3,5]
SwingPanel [4]
+ SwingPanel(int, int) [4]
[inheritance]
[inheritance]
Text [1,3]
+ Text(String, Font, Color) [1,3]
Rectangle [3,5]
+ Rectangle(int, int, Color) [3,5]
[inheritance]
[inheritance]
Frame [1,2,5]
+ Frame(int, int) [1,2,5]
FullScreen [2,3]
+ FullScreen(int, int, int) [2,3]

Figure 5.5: Intermediate result after simplifying hierarchies (updated nodes highlighted with thick stroke)

Algorithm 5.10 Pulling Operations Out of their Class

```
1: function pullOperationsOut($g$, $root$)
2:     repeat
3:         for all leaf nodes $l \in$ non-visited nodes of tree for $root$ do
4:             mark $l$ as visited
5:             for all class $c \in l.classes$ do
6:                 for all operation $o \in$ operations of $c$ do
7:                     if $l.examples(o) \neq \emptyset$ and $l.examples(c) \neq l.examples(o)$ then
8:                         $c_s \leftarrow$ copy of $c$ with $o$ as the only element
9:                         $n \leftarrow$ new node for $o$
10:                        $n.classes \leftarrow \{c_s\}$
11:                        $n.examples \leftarrow l.examples(o)$
12:                        remove example usage of $o$ from $l.examples$
13:                        $g.V \leftarrow g.V \cup n$
14:                        $e \leftarrow$ new edge $(n, l)$
15:                        $e.T \leftarrow [containment]$
16:                        $g.E \leftarrow g.E \cup e$
17:                    end if
18:                end for
19:            end for
20:        end for
21:    until all leaf nodes visited
22: end function
```

5.5.8 Pulling Operations Out

If hypotheses H2 and H4 hold, all code elements belonging to the same feature are used in the same code examples. Therefore, if an operation is used in examples, but not used in the same examples as the class that contains the operation, the operation can potentially belong to a different feature. As a consequence, we move those operations out of the node that contains the class into a new child node.

Algorithm 5.10 shows the definition of the `pullOperationsOut` function. The pulled out operation is placed in a shallow copy of the class. This makes it easier at a later stage to perform a class merge when nodes containing the same class are merged. A potential performance improving step could be to already merge operations of the same class with the same usage. For brevity, this was left out. These nodes will then be merged in a subsequent step.

Figure 5.6 shows the result after pulling out operations of all hierarchies is finalized. All operations with an example usage subset compared to the class are moved out into a new child node.

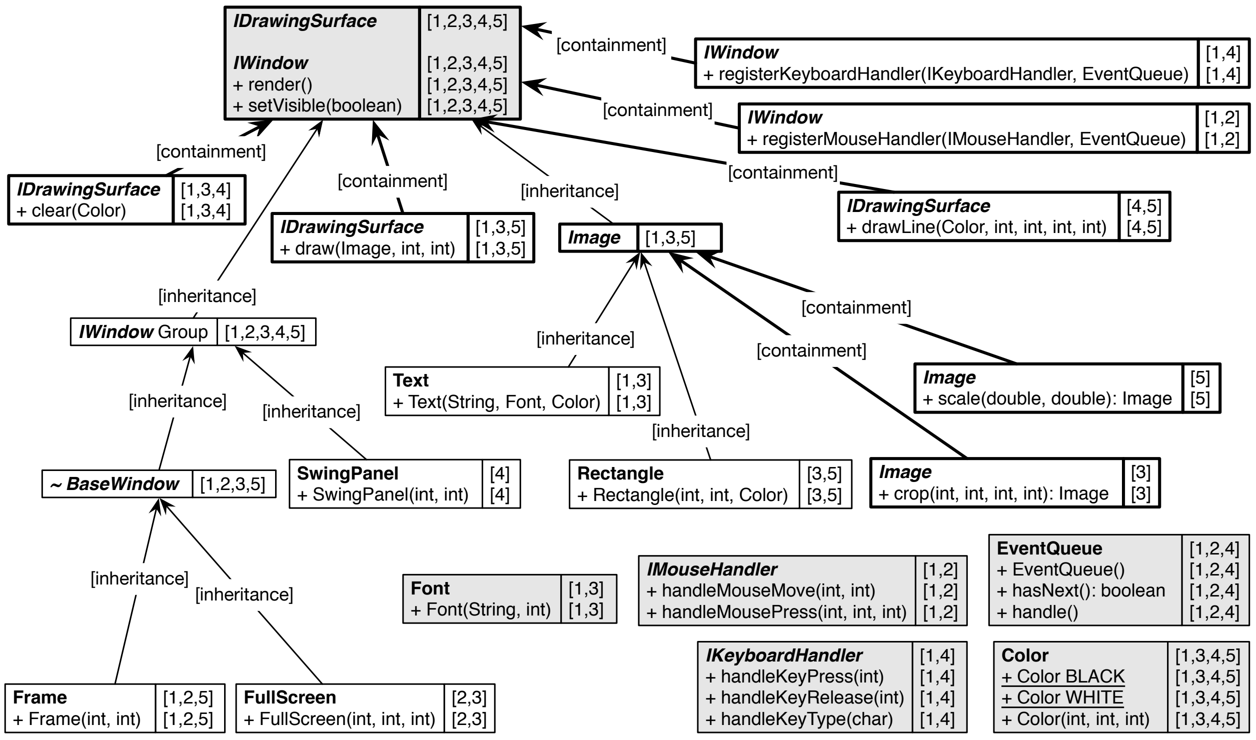

IDrawingSurface [1,2,3,4,5]
IWindow [1,2,3,4,5]
+ render() [1,2,3,4,5]
+ setVisible(boolean) [1,2,3,4,5]
[containment]
IWindow [1,4]
+ registerKeyboardHandler(IKeyboardHandler, EventQueue) [1,4]
[containment]
IWindow [1,2]
+ registerMouseHandler(IMouseHandler, EventQueue) [1,2]
[containment]
IDrawingSurface [4,5]
+ drawLine(Color, int, int, int, int) [4,5]
[containment]
IDrawingSurface [1,3,4]
+ clear(Color) [1,3,4]
[containment]
IDrawingSurface [1,3,5]
+ draw(Image, int, int) [1,3,5]
[inheritance]
Image [1,3,5]
[inheritance]
IWindow Group [1,2,3,4,5]
[inheritance]
[inheritance]
~ BaseWindow [1,2,3,5]
SwingPanel [4]
+ SwingPanel(int, int) [4]
[inheritance]
Text [1,3]
+ Text(String, Font, Color) [1,3]
[inheritance]
Rectangle [3,5]
+ Rectangle(int, int, Color) [3,5]
[containment]
[containment]
Image [5]
+ scale(double, double): Image [5]
Image [3]
+ crop(int, int, int, int): Image [3]
[inheritance]
[inheritance]
Frame [1,2,5]
+ Frame(int, int) [1,2,5]
FullScreen [2,3]
+ FullScreen(int, int, int) [2,3]
Font [1,3]
+ Font(String, int) [1,3]
IMouseHandler [1,2]
+ handleMouseMove(int, int) [1,2]
+ handleMousePress(int, int, int) [1,2]
EventQueue [1,2,4]
+ EventQueue() [1,2,4]
+ hasNext(): boolean [1,2,4]
+ handle() [1,2,4]
IKeyboardHandler [1,4]
+ handleKeyPress(int) [1,4]
+ handleKeyRelease(int) [1,4]
+ handleKeyType(char) [1,4]
Color [1,3,4,5]
+ Color BLACK [1,3,4,5]
+ Color WHITE [1,3,4,5]
+ Color(int, int, int) [1,3,4,5]

Figure 5.6: Intermediate result after pulling operations out of their class (new or updated nodes and edges highlighted with thick stroke)

This makes it possible in subsequent steps to merge the pulled out operations with other nodes, if their example usage matches.

5.5.9 Connecting the Individual Hierarchies

At this point the first phase is concluded. The DAG contains several directed rooted trees that represent the individual hierarchies. The final goal is to deduce a single directed rooted tree representing the feature model. To establish a DAG with a single root we therefore need to connect the individual hierarchies. Packages (and inner classes) provide a structural grouping of classes. We make use of this information to connect the roots of all hierarchies. In order for the single DAG to be created, the roots of all clusters are connected based on the structural grouping of their classes.

Algorithm 5.11 shows the function `connectRoots` that takes care of this. First, nodes for all packages are created and connected with edges between a sub-package and its parent package. The node representing the root package of the framework becomes the root of the DAG. Then, the roots of the directed rooted trees of the individual hierarchies are connected with the nodes representing the structural grouping node of the root's classes. Edges with type *grouping* are added to the graph to establish this structural grouping.

Once this step of the algorithm completes, the DAG has one root node (representing the root package). Figure 5.7 shows the resulting DAG for our example. Depending on the package and hierarchy structure, it is possible at this point that a node has several outgoing edges to different packages. For example, in the Minueto framework, the interfaces `MinuetoDrawingSurface` and `MinuetoWindow` are located in different packages (the former in `org.minueto.image`, the latter in `org.minueto.window`). This is the case even though `MinuetoWindow` extends the interface of `MinuetoDrawingSurface`. Due to the fact that they are within the same node, their node has two outgoing edges. The same is reflected in Figure 5.7 for the node containing `IDrawingSurface` and `IWindow`.

5.5.10 Merging Similar Nodes

The DAG at this point contains many different nodes, either representing packages, or containing classes or their elements. Based on H2 and H4, if two elements are used in the same set of examples, they belong to the same feature. Therefore, across the complete DAG, all nodes with the same example usage need to be merged.

Algorithm 5.12 shows the algorithm for the `mergeSimilarNodes` function. The example usages of all nodes are compared with each other. If two example usages match, the two corresponding nodes are merged into one using the `mergeNodes` auxiliary function.

Figure 5.8 illustrates the intermediate DAG of our example after merging is completed. Most notably, the *register handler* methods as well as the *draw* method, which were previously pulled

Algorithm 5.11 Establishing a DAG with a Single Root

```
function CONNECTROOTS($g$, $R$)
    $packages \leftarrow$ determine package structure of all classes in $g$ through $R$
    for all package $p \in$ hierarchy of $packages$ do
        $n \leftarrow$ new node for $p$
        $n.name \leftarrow$ name of $p$
        $g.V \leftarrow g.V \cup n$
        for parent package $s \in R \mid ce_s = p$ do
            $e \leftarrow$ new edge $(n, p)$
            $e.T \leftarrow [grouping]$
            $g.E \leftarrow g.E \cup e$
        end for
    end for
    for all root $r \in g.roots$ do
        for all class $c \in r.classes$ do
            if $\exists x \in R \mid ce_s = c$ and $t = is\ inner\ class\ of$ then
                $o \leftarrow$ determine node for outer class $x.ce_t$
                $e \leftarrow$ new edge $(r, o)$
                $e.T \leftarrow [grouping]$
                $g.E \leftarrow g.E \cup e$
            else
                $x \leftarrow$ determine parent package $\in R \mid ce_s = c$ and $t = is\ in\ package$
                $p \leftarrow$ determine node for parent package $x.ce_t$
                $e \leftarrow$ new edge $(r, p)$
                $e.T \leftarrow [grouping]$
                $g.E \leftarrow g.E \cup e$
            end if
        end for
    end for
    $g.roots \leftarrow \{$root package$\}$
end function
```

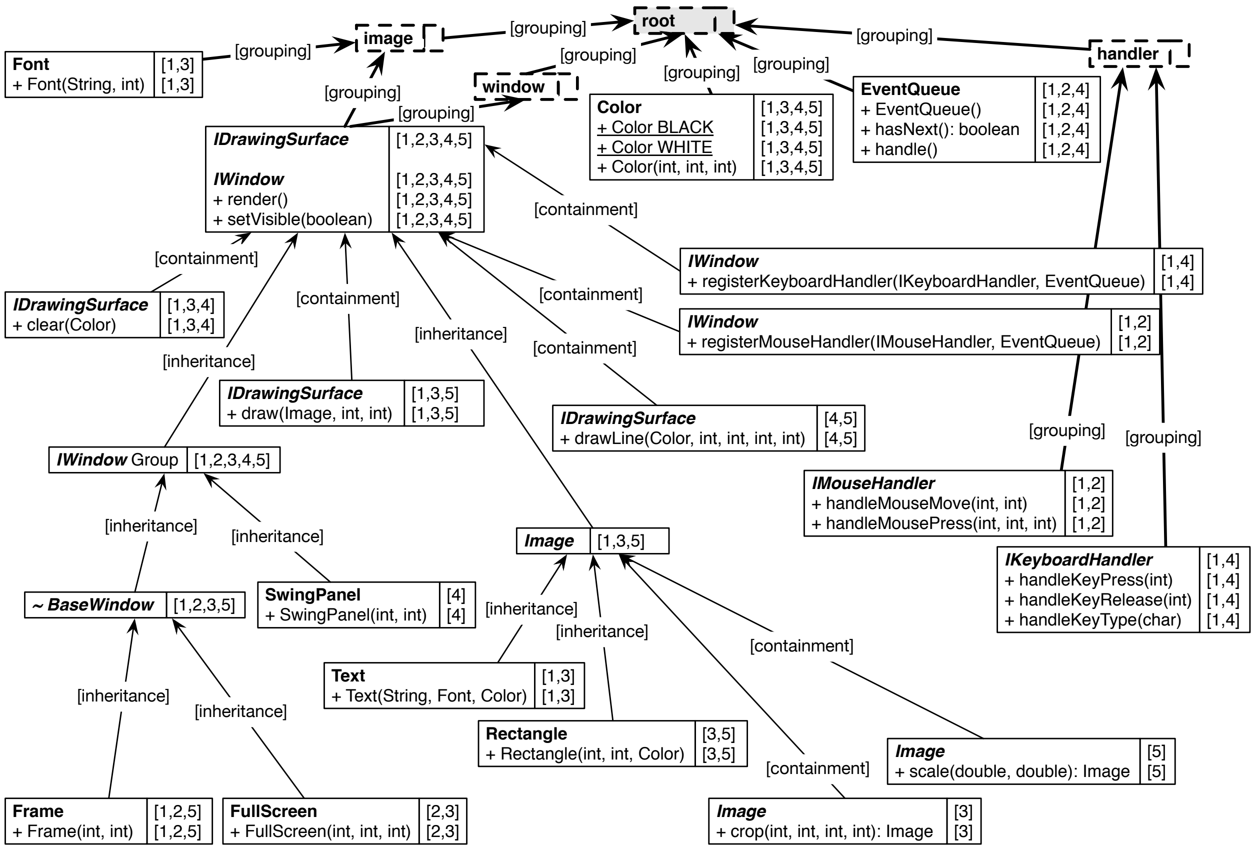

root
image
window
handler
[grouping]
[containment]
[inheritance]
Font [1,3]
+ Font(String, int) [1,3]
IDrawingSurface [1,2,3,4,5]
IWindow [1,2,3,4,5]
+ render() [1,2,3,4,5]
+ setVisible(boolean) [1,2,3,4,5]
Color [1,3,4,5]
+ Color BLACK [1,3,4,5]
+ Color WHITE [1,3,4,5]
+ Color(int, int, int) [1,3,4,5]
EventQueue [1,2,4]
+ EventQueue() [1,2,4]
+ hasNext(): boolean [1,2,4]
+ handle() [1,2,4]
IWindow [1,4]
+ registerKeyboardHandler(IKeyboardHandler, EventQueue) [1,4]
IWindow [1,2]
+ registerMouseHandler(IMouseHandler, EventQueue) [1,2]
IDrawingSurface [1,3,4]
+ clear(Color) [1,3,4]
IDrawingSurface [1,3,5]
+ draw(Image, int, int) [1,3,5]
IDrawingSurface [4,5]
+ drawLine(Color, int, int, int, int) [4,5]
IWindow Group [1,2,3,4,5]
IMouseHandler [1,2]
+ handleMouseMove(int, int) [1,2]
+ handleMousePress(int, int, int) [1,2]
Image [1,3,5]
IKeyboardHandler [1,4]
+ handleKeyPress(int) [1,4]
+ handleKeyRelease(int) [1,4]
+ handleKeyType(char) [1,4]
~ BaseWindow [1,2,3,5]
SwingPanel [4]
+ SwingPanel(int, int) [4]
Text [1,3]
+ Text(String, Font, Color) [1,3]
Rectangle [3,5]
+ Rectangle(int, int, Color) [3,5]
Image [5]
+ scale(double, double): Image [5]
Frame [1,2,5]
+ Frame(int, int) [1,2,5]
FullScreen [2,3]
+ FullScreen(int, int, int) [2,3]
Image [3]
+ crop(int, int, int, int): Image [3]

Figure 5.7: Intermediate result after connecting the roots (new or updated nodes and edges highlighted with thick stroke, package nodes with a dashed border)

Algorithm 5.12 Merging Nodes with the Same Usage

```
 1: function MERGESIMILARNODES($g$)
 2:     for all node $n_1 \in g.V$ do
 3:         for all node $n_2 \in g.V \setminus n_1$ do
 4:             if $n_1.examples = n_2.examples$ then
 5:                 if $n_1$ and $n_2$ are not marked as potential grouping nodes then
 6:                     MERGENODES($g$, $n_1$, $n_2$)
 7:                 end if
 8:             end if
 9:         end for
10:     end for
11: end function
```

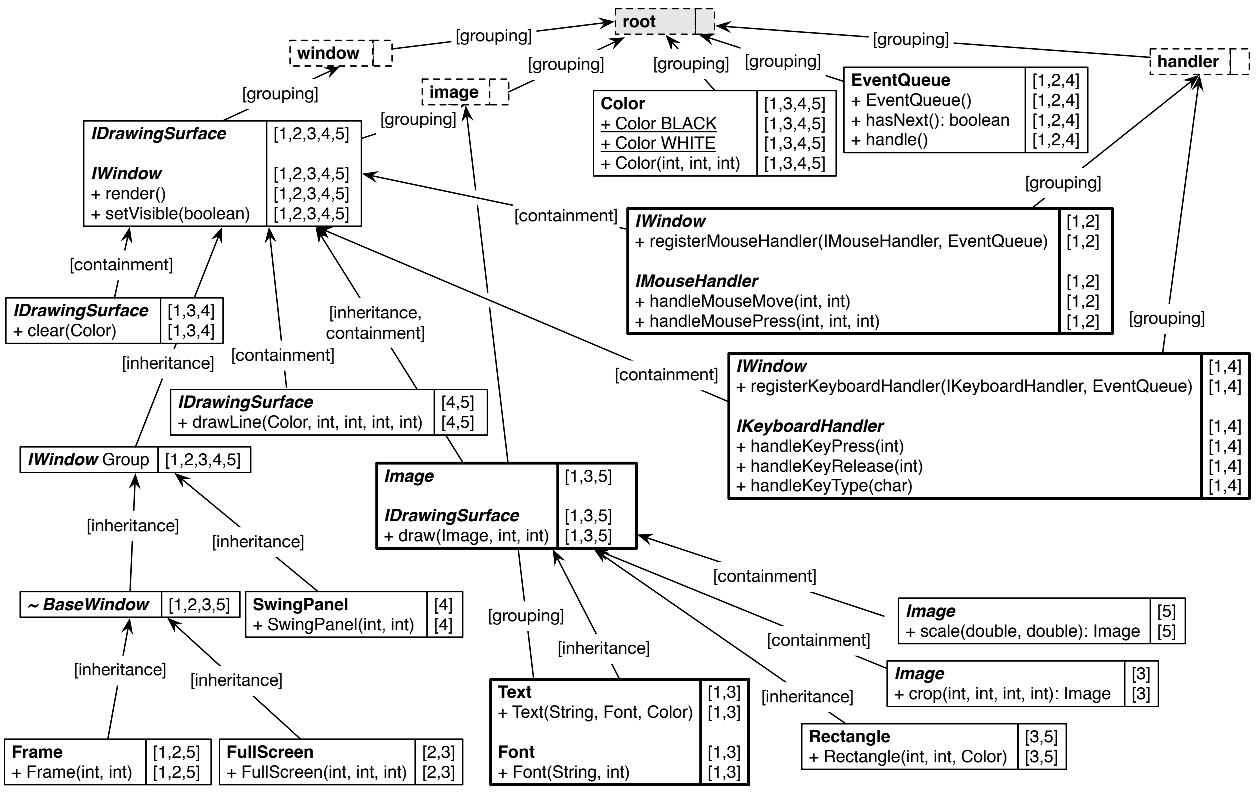

root
window
image
handler
[grouping]
[containment]
[inheritance]
[inheritance, containment]
IDrawingSurface [1,2,3,4,5]
IWindow [1,2,3,4,5]
+ render() [1,2,3,4,5]
+ setVisible(boolean) [1,2,3,4,5]
Color [1,3,4,5]
+ Color BLACK [1,3,4,5]
+ Color WHITE [1,3,4,5]
+ Color(int, int, int) [1,3,4,5]
EventQueue [1,2,4]
+ EventQueue() [1,2,4]
+ hasNext(): boolean [1,2,4]
+ handle() [1,2,4]
IWindow [1,2]
+ registerMouseHandler(IMouseHandler, EventQueue) [1,2]
IMouseHandler [1,2]
+ handleMouseMove(int, int) [1,2]
+ handleMousePress(int, int, int) [1,2]
IWindow [1,4]
+ registerKeyboardHandler(IKeyboardHandler, EventQueue) [1,4]
IKeyboardHandler [1,4]
+ handleKeyPress(int) [1,4]
+ handleKeyRelease(int) [1,4]
+ handleKeyType(char) [1,4]
IDrawingSurface [1,3,4]
+ clear(Color) [1,3,4]
IDrawingSurface [4,5]
+ drawLine(Color, int, int, int, int) [4,5]
IWindow Group [1,2,3,4,5]
Image [1,3,5]
IDrawingSurface [1,3,5]
+ draw(Image, int, int) [1,3,5]
~ BaseWindow [1,2,3,5]
SwingPanel [4]
+ SwingPanel(int, int) [4]
Image [5]
+ scale(double, double): Image [5]
Image [3]
+ crop(int, int, int, int): Image [3]
Frame [1,2,5]
+ Frame(int, int) [1,2,5]
FullScreen [2,3]
+ FullScreen(int, int, int) [2,3]
Text [1,3]
+ Text(String, Font, Color) [1,3]
Font [1,3]
+ Font(String, int) [1,3]
Rectangle [3,5]
+ Rectangle(int, int, Color) [3,5]

Figure 5.8: Intermediate result after merging nodes with the same example usage (merged nodes highlighted with thick border)

Algorithm 5.13 Adding Cross-references to the Graph

```
function ADDCROSSREFERENCES($g$, $R$)
    for all node $n \in g.V$ do
        $crossrefs \leftarrow \forall r \in R \mid (ce_s = any\ element \in n$ and $t = has\ crossreference\ to)$
        for all crossref $c \in crossrefs$ do
            $d \leftarrow$ node containing class of dependency
            if $\neg\exists$ path $n - d$ then
                $e \leftarrow$ new edge $(n, d$
                $e.T \leftarrow [crossref]$
                $g.E \leftarrow g.E \cup e$
            else
                add *requires* constraints from $n$ to $d$
            end if
        end for
    end for
end function
```

out of their class, were merged with other classes. Nodes that were merged are shown with a thick border for convenience.

5.5.11 Adding Cross-references

The last step in populating the DAG is to add cross-reference information. Cross-references form additional dependency information. If a method takes as a parameter (or returns) an instance of another class, the method depends on this other class. Furthermore, checked exceptions form a dependency as well since the caller needs to handle them. Therefore, this information needs to be added to the graph to capture these dependencies.

Algorithm 5.13 provides the algorithm for the `addCrossReferences` function. Before adding a cross-reference, it needs to be ensured that this does not add a cycle to the graph, i.e., there is no path in the opposite direction yet, from the dependee to the source. This can happen especially if hypotheses H1 and H3 do not hold. This means that unused elements are not removed and such operations are also not pulled out because no decision can be made about them. In addition, up to this point the existing path in the inverse direction can only be of type *inheritance* or *containment*. We consider them to be more important than *crossref* edges (we discuss this in more detail later in Subsection 5.5.12.5). Therefore, instead of avoiding a cycle by merging the nodes, we add a *requires* cross-tree constraint to ensure that the required type will still be available for use at the end.

Figure 5.9 shows the result after adding all cross-reference edges to the graph. The additional

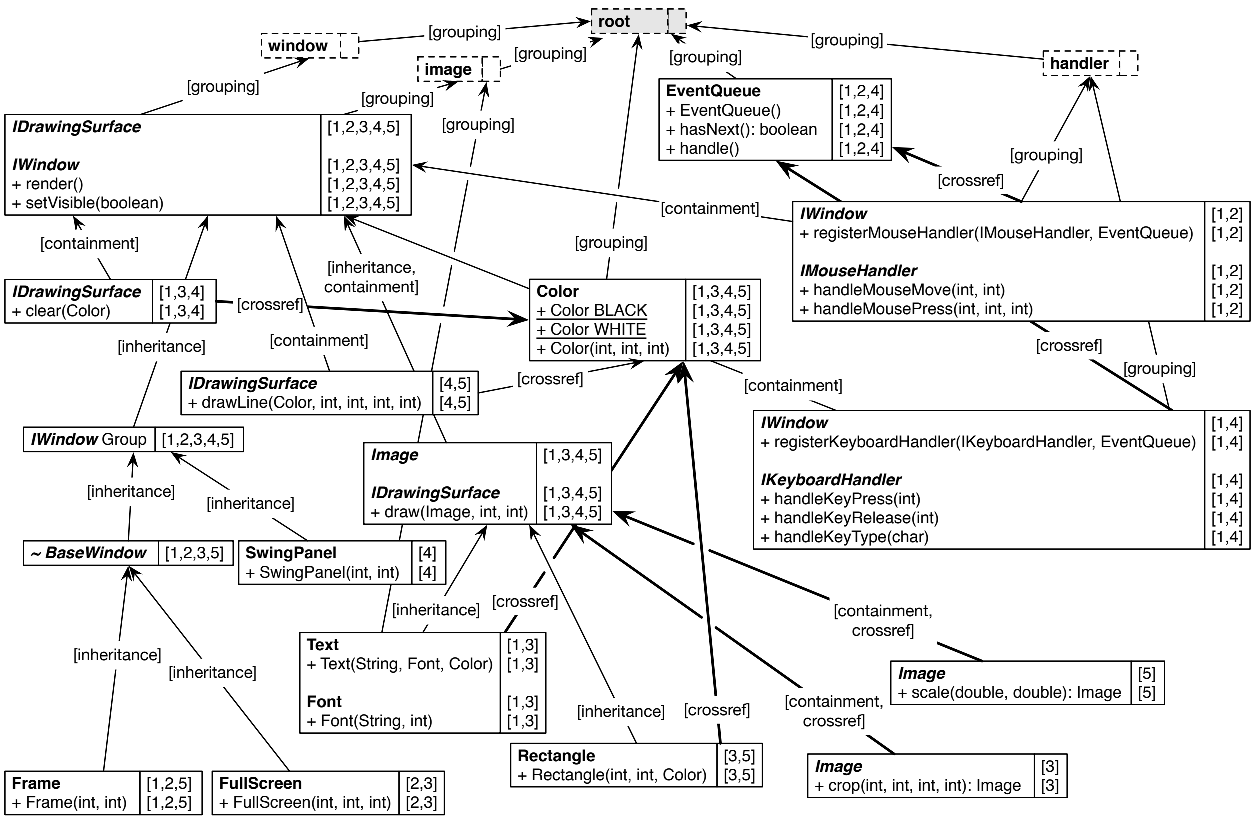

root
window
image
handler
[grouping]
EventQueue [1,2,4]
+ EventQueue() [1,2,4]
+ hasNext(): boolean [1,2,4]
+ handle() [1,2,4]
IDrawingSurface [1,2,3,4,5]
IWindow [1,2,3,4,5]
+ render() [1,2,3,4,5]
+ setVisible(boolean) [1,2,3,4,5]
IWindow [1,2]
+ registerMouseHandler(IMouseHandler, EventQueue) [1,2]
IMouseHandler [1,2]
+ handleMouseMove(int, int) [1,2]
+ handleMousePress(int, int, int) [1,2]
IDrawingSurface [1,3,4]
+ clear(Color) [1,3,4]
Color [1,3,4,5]
+ Color BLACK [1,3,4,5]
+ Color WHITE [1,3,4,5]
+ Color(int, int, int) [1,3,4,5]
IDrawingSurface [4,5]
+ drawLine(Color, int, int, int, int) [4,5]
IWindow Group [1,2,3,4,5]
IWindow [1,4]
+ registerKeyboardHandler(IKeyboardHandler, EventQueue) [1,4]
IKeyboardHandler [1,4]
+ handleKeyPress(int) [1,4]
+ handleKeyRelease(int) [1,4]
+ handleKeyType(char) [1,4]
Image [1,3,4,5]
IDrawingSurface [1,3,4,5]
+ draw(Image, int, int) [1,3,4,5]
~ BaseWindow [1,2,3,5]
SwingPanel [4]
+ SwingPanel(int, int) [4]
Text [1,3]
+ Text(String, Font, Color) [1,3]
Font [1,3]
+ Font(String, int) [1,3]
Image [5]
+ scale(double, double): Image [5]
Rectangle [3,5]
+ Rectangle(int, int, Color) [3,5]
Image [3]
+ crop(int, int, int, int): Image [3]
Frame [1,2,5]
+ Frame(int, int) [1,2,5]
FullScreen [2,3]
+ FullScreen(int, int, int) [2,3]
[containment]
[crossref]
[inheritance]
[inheritance, containment]
[containment, crossref]

Figure 5.9: Intermediate result after adding cross-reference edges (added *crossref* edges are highlighted with thick stroke)

crossref edges are highlighted with a thick stroke.

5.5.12 Graph Simplification

After the cross-references are added all dependency links are present in the graph. While we made sure that there are no dependency cycles in the graph, at this point, some nodes may have more than one outgoing edge. Since the ultimate goal is to create a feature model, the last phase of the algorithm performs a series of simplification steps (i.e., remove nodes or edges) in order to convert the DAG into a tree. However, a node or edge can only be removed if the integrity of the API is maintained, i.e., every correct feature selection on the resulting feature model produces the corresponding subset of the API related to that feature. In the following subsections we discuss the simplifications and the reasons for why they can be performed. And as a last resort, an edge from f_1 to f_2 can be replaced with a *requires* cross-tree constraint for the feature model indicating that when selecting f_1 also f_2 needs to be selected.

5.5.12.1 Merging Mandatory API Elements

Often there are API elements that are used in every usage scenario. Without them, correct use of the framework is not possible. As a result, they are used in every example. Due to the insertion of package nodes when connecting all hierarchies (see Section 5.5.9), these API elements are currently not in the root node. In a feature model, the root feature is always going to be selected. Since the API elements always need to be used, we can ensure the availability of those API elements by merging the node containing those elements with the current root using the `mergeNodes` function. In our example, the node with `IDrawingSurface` and `IWindow` is used in all examples and hence merged with the root node.

5.5.12.2 Merging Equivalent Groups

It is possible that there exist two nodes that group the same children, i.e., both have incoming edges originating from the same nodes. This means that all children have at least two outgoing edges which need to be dealt with. In such a case, we merge both nodes. For example, in our example (see Figure 5.9) the node for the class `EventQueue` and the node for package `handler` group the same children and are therefore merged as a result.

5.5.12.3 Merging Utility Nodes

There can exist classes that are used by several other classes and potentially do not present a user-perceived feature on their own. Instead, they can represent a utility that needs to be used. To detect this, we consider nodes across the complete graph. A requirement that exists is that all nodes with an outgoing edge to such a node need to have at least one other outgoing edge. This means that these edges need to be simplified in order to reach our goal of retrieving a tree at the end.

A possible way to deal with this is to replace them with *requires* constraints. However, this can potentially lead to a large number of constraints. We can however take advantage of the benefit of CORE and its composition. As explained in sections 2.4.3 and 2.5, the composition operator of structural views (class diagram) in the design models will merge classes. Therefore, an alternative is to copy the contents of the node into each referencing node. We therefore relax the property for those elements for $api(f)$ that an API element e only belongs to one feature to allow sharing the same code elements among features.

We define a utility node as such if it satisfies the following properties:

- the node is not within a hierarchy, i.e., it does not have any outgoing edges with type *inheritance*,
- the nodes pointing to the node have more than one outgoing edge,
- the edges of nodes pointing to the node are
  - only of type *crossref*, or
  - only of type *grouping*, and
- the union of usage in examples of the nodes pointing to the node matches the example usage of the node.

If these properties are satisfied, the contents of the node can be copied into each of the nodes pointing to it, and the node be removed at the end. For example, in our example (intermediate result prior to graph simplification shown in Figure 5.9), the node for class `Color` is cross-referenced by various other nodes whose classes contain operations requiring a `Color` instance.

5.5.12.4 Transitive Reduction

Based on feature model configurations, if a feature f is selected, it means that all its ancestors are selected as well. That means that if there is an additional edge from a feature to its indirect ancestor, this edge can be removed by performing *transitive reduction*. As a result, all redundant edges are removed from the graph. The previous simplification steps merged nodes which can change the edges in the DAG. Therefore, transitive reduction is performed after those steps. This also ensures that any edge to the node with mandatory elements (which is the root node) is removed for nodes with multiple outgoing edges, since its elements are always available.

Figure 5.10 shows the intermediate result after these simplification steps are performed. The `IDrawingSurface` and `IWindow` interfaces were merged into the root because they are mandatory. The `handler` package node and node for `EventQueue` were merged as equivalent groups.

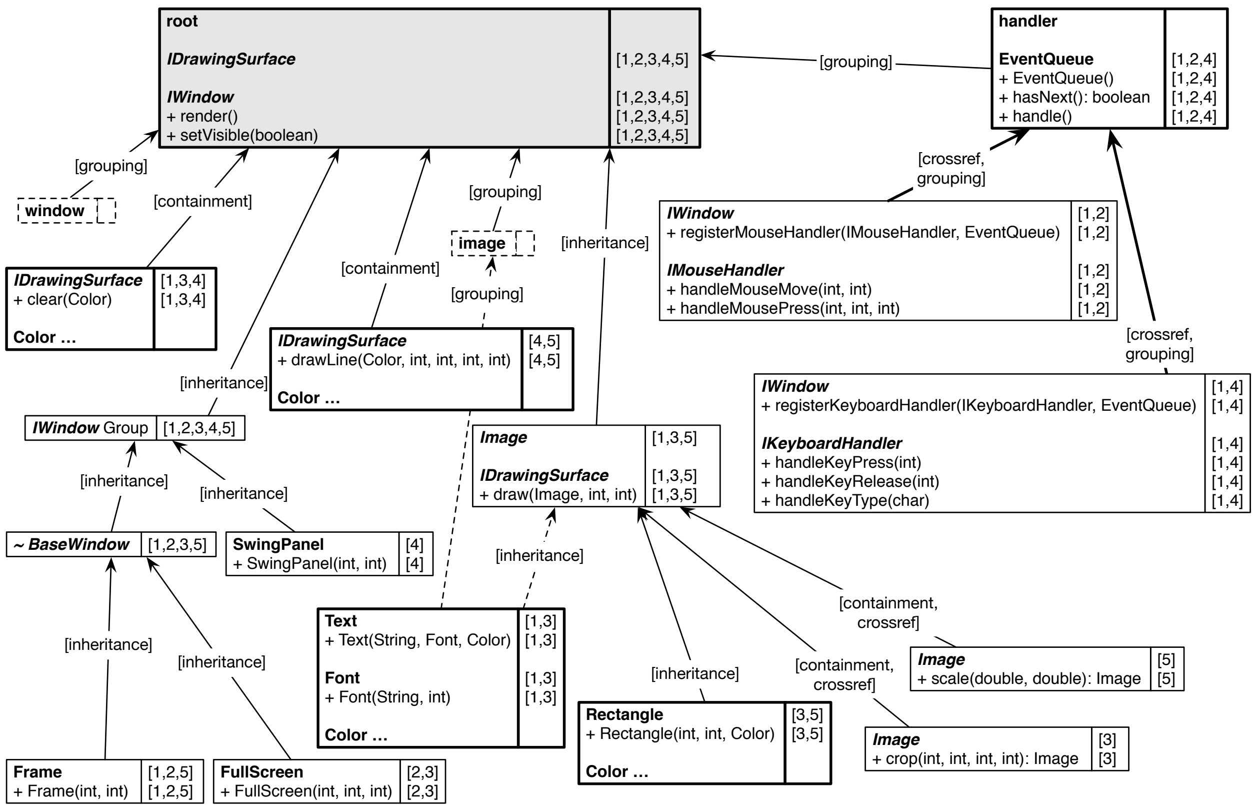

root
IDrawingSurface [1,2,3,4,5]
IWindow [1,2,3,4,5]
+ render() [1,2,3,4,5]
+ setVisible(boolean) [1,2,3,4,5]
[grouping]
handler
EventQueue [1,2,4]
+ EventQueue() [1,2,4]
+ hasNext(): boolean [1,2,4]
+ handle() [1,2,4]
[grouping]
window
[containment]
[grouping]
image
[grouping]
[inheritance]
[containment]
[crossref, grouping]
IWindow [1,2]
+ registerMouseHandler(IMouseHandler, EventQueue) [1,2]
IMouseHandler [1,2]
+ handleMouseMove(int, int) [1,2]
+ handleMousePress(int, int, int) [1,2]
IDrawingSurface [1,3,4]
+ clear(Color) [1,3,4]
Color …
IDrawingSurface [4,5]
+ drawLine(Color, int, int, int, int) [4,5]
Color …
[crossref, grouping]
IWindow [1,4]
+ registerKeyboardHandler(IKeyboardHandler, EventQueue) [1,4]
IKeyboardHandler [1,4]
+ handleKeyPress(int) [1,4]
+ handleKeyRelease(int) [1,4]
+ handleKeyType(char) [1,4]
[inheritance]
IWindow Group [1,2,3,4,5]
[inheritance]
[inheritance]
Image [1,3,5]
IDrawingSurface [1,3,5]
+ draw(Image, int, int) [1,3,5]
~ BaseWindow [1,2,3,5]
SwingPanel [4]
+ SwingPanel(int, int) [4]
[inheritance]
[containment, crossref]
[inheritance]
[inheritance]
Text [1,3]
+ Text(String, Font, Color) [1,3]
Font [1,3]
+ Font(String, int) [1,3]
Color …
[inheritance]
[containment, crossref]
Image [5]
+ scale(double, double): Image [5]
Rectangle [3,5]
+ Rectangle(int, int, Color) [3,5]
Color …
Image [3]
+ crop(int, int, int, int): Image [3]
Frame [1,2,5]
+ Frame(int, int) [1,2,5]
FullScreen [2,3]
+ FullScreen(int, int, int) [2,3]

Figure 5.10: Intermediate result after the first part of graph simplification (updated nodes highlighted with a thick stroke, edges with a dashed line still need to be simplified)

The node for `Color` was identified as a utility node and therefore copied into each referencing node. During transitive reduction the *grouping* edges from the nodes for the two handlers to the root were removed. There is still one node (for `Text` and `Font`) with more than one outgoing edge. Its outgoing edges are highlighted with a dashed line and further simplification is required.

5.5.12.5 Edge Simplification

Until this point several simplification steps have been performed. The node with mandatory API elements was merged into the root node. Nodes grouping the same nodes as well as utility nodes were merged. And finally, transitive reduction removed redundant edges. However, there can still exist nodes with multiple outgoing edges. The last information we can use for simplification is the types of the edges. We can use this information to decide which edge to remove. However, as noted at the beginning of this subsection, we need to ensure that the integrity of the API is maintained. This means that in certain cases, the removal of an edge requires the addition of a corresponding *requires* cross-tree constraint.

Our intuition is that the edge types have different priority. For instance, the edge type *grouping* represents a potential grouping in the case that a node is not associated with any other nodes. As such, if there are edges of other types, the *grouping* edge can be removed. The edge type *inherit*ance represents a relationship between the sub-class and its superclass. Because inheritance is usually used when there is more than one sub-class, in accordance with our common hypothesis 5 (see Section 5.4), the super-class provides a grouping of alternate choices among sub-classes. This should be reflected in the final feature model. Therefore, the *inheritance* relationship has the highest priority. *Containment* and *crossref* share similar characteristics. Both require the depending type to be available. *Containment* reflects an edge due to the fact that an operation of a class was used less than the class itself. However, a *crossref* edge only exists if the type that is depended on is used less than the node itself (i.e., they were not used in the same examples and therefore not merged). Therefore, we regard *containment* as a higher priority.

To summarize, the priority ordering of edge types to help the decision on which edge to remove in descending order of priority is:

1. *inheritance*,
2. *containment*,
3. *crossref*,
4. *grouping*.

In order to deal with the remaining nodes that have multiple outgoing edges, their edge types are compared with each other pair-wise and the edge with lower priority is removed. Removing an

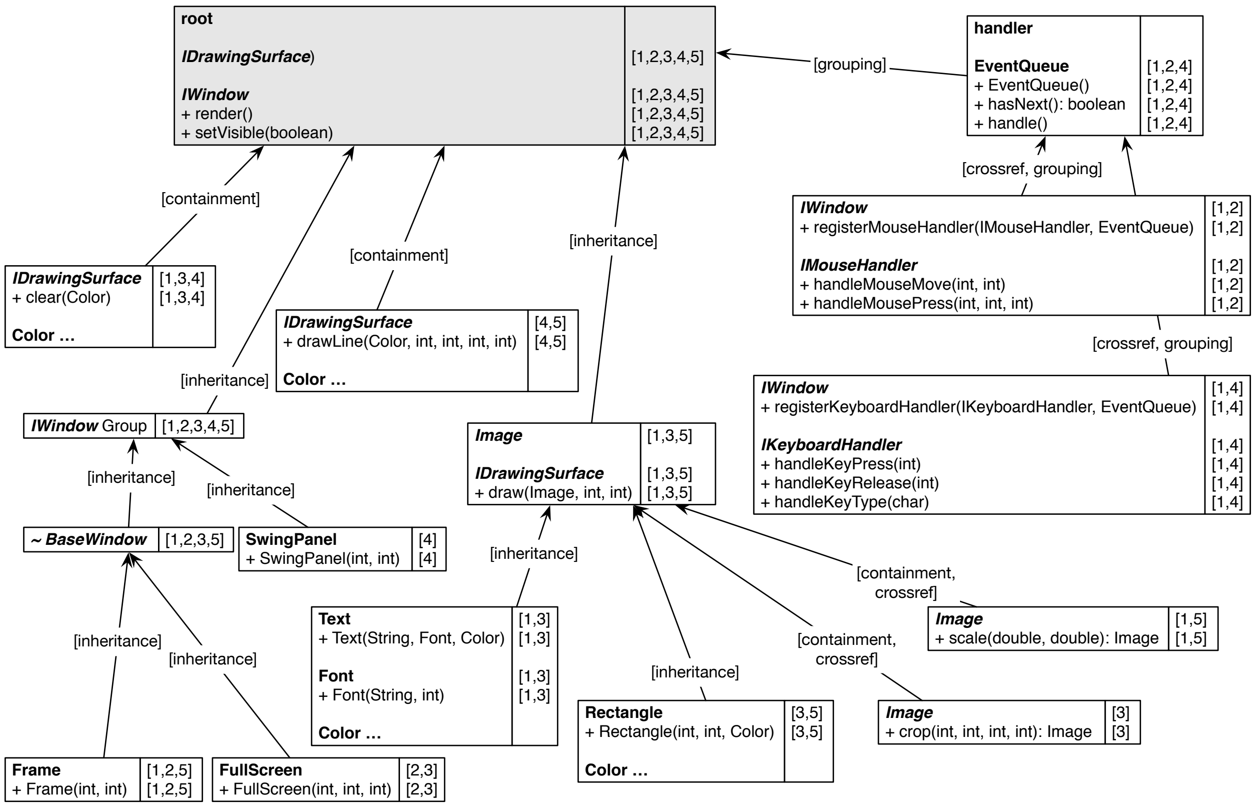

root
IDrawingSurface) [1,2,3,4,5]
IWindow [1,2,3,4,5]
+ render() [1,2,3,4,5]
+ setVisible(boolean) [1,2,3,4,5]
handler
EventQueue [1,2,4]
+ EventQueue() [1,2,4]
+ hasNext(): boolean [1,2,4]
+ handle() [1,2,4]
[grouping]
[crossref, grouping]
IWindow [1,2]
+ registerMouseHandler(IMouseHandler, EventQueue) [1,2]
IMouseHandler [1,2]
+ handleMouseMove(int, int) [1,2]
+ handleMousePress(int, int, int) [1,2]
[crossref, grouping]
IWindow [1,4]
+ registerKeyboardHandler(IKeyboardHandler, EventQueue) [1,4]
IKeyboardHandler [1,4]
+ handleKeyPress(int) [1,4]
+ handleKeyRelease(int) [1,4]
+ handleKeyType(char) [1,4]
[containment]
[containment]
[inheritance]
IDrawingSurface [1,3,4]
+ clear(Color) [1,3,4]
Color ...
IDrawingSurface [4,5]
+ drawLine(Color, int, int, int, int) [4,5]
Color ...
[inheritance]
IWindow Group [1,2,3,4,5]
[inheritance]
[inheritance]
~ BaseWindow [1,2,3,5]
SwingPanel [4]
+ SwingPanel(int, int) [4]
Image [1,3,5]
IDrawingSurface [1,3,5]
+ draw(Image, int, int) [1,3,5]
[inheritance]
[containment, crossref]
Image [1,5]
+ scale(double, double): Image [1,5]
[containment, crossref]
Image [3]
+ crop(int, int, int, int): Image [3]
[inheritance]
[inheritance]
Text [1,3]
+ Text(String, Font, Color) [1,3]
Font [1,3]
+ Font(String, int) [1,3]
Color ...
[inheritance]
Rectangle [3,5]
+ Rectangle(int, int, Color) [3,5]
Color ...
Frame [1,2,5]
+ Frame(int, int) [1,2,5]
FullScreen [2,3]
+ FullScreen(int, int, int) [2,3]

Figure 5.11: Final result after the edge simplification (root shown in grey)

edge with type *grouping* does not remove any dependency information and no cross-tree constraint is required. In all other cases, i.e., when a *containment* or *crossref* edge is removed, a *requires* cross-tree constraint needs to be added for the removed edge to maintain the integrity of the API. In the case that two edges have the same type, the edge with the lower amount of this type is removed. If they have the same amount of this type, one of the edges is chosen randomly to be removed.

Figure 5.11 shows the final result after edge simplification. The *grouping* type edge to the image package node was removed from the *Text* and *Font* node. Because it is a package node, no cross-tree constraint is required to compensate for removing the edge.

It is possible that as a result of the simplification steps the graph contains nodes representing packages that do not group any nodes anymore, or only group one child. Therefore, such a node can be removed using the `removeNode` function. In our example (see Figure 5.10), this is the case with the `window` and `image` package nodes.

5.5.12.6 Reducing the Feature Model size

Lastly, an optional simplification that can be provided to the user is the ability to merge pulled out operations nodes back up to their parent if they weren't merged with any other nodes based on the usage. Especially for frameworks with a large API, it is unrealistic to provide examples that make use of all operations and with all possible combinations. Therefore, the probability is high that at the end there exist several nodes with only operations that were pulled out but never merged. For example, in our example (see Figure 5.11), the operations `scale` and `crop` of `Image` were pulled out, but not merged.

However, the user can also perform this manually after retrieving the determined feature model as this depends on the desired granularity.

5.5.13 Converting to a Feature Model

At this point the DAG takes the form of a directed rooted tree and can be converted—along with the cross-tree constraints—into a feature model. The remaining step is to determine the relationships between the parent features and their children. By default, all children are *optional*. However, based on common hypothesis 5, parent nodes with an abstract super-class and empty nodes need to have at least an OR relationship with its children to allow valid feature selections. Based on the graph simplification, the children represent the sub-classes. For example, this is the case in our example for the *IWindow Group* (as an empty grouping node) and the *BaseWindow* (as an abstract super-class) nodes (see Figure 5.11). If hypothesis H1a holds, it allows the algorithm to determine whether the children are in an XOR-relationship. In our example, the children of the *IWindow Group* are in an XOR-relationship, whereas the children of *BaseWindow* remain in the OR-relationship. In certain cases, e.g., if a parent node containing an abstract super-class has more children than solely the sub-classes, an intermediary node needs to be introduced to group the sub-classes. The intermediary node then becomes *mandatory*, and the other children *optional*. The same applies to the *IWindow Group* node in our example, because it is used in the same examples as its parent.

Based on hypotheses H1a and H1b, for the remaining nodes it can be checked based on the example usage whether the children should be in an OR-relationship and potentially even an XOR-relationship. For instance, the children of the node representing the `handler` package and the `EventQueue` class show that whenever the parent is used, at least one of its children is used as well. Additionally, the example usage shows that both handlers can be used together. As a result, the relationship becomes OR. The resulting feature model for our example is shown in Figure 5.12.

As a reminder to the reader, the automated concernification algorithm provides the identification of user-relevant features of the framework (step 1a in Section 3.4) and organizes them in a

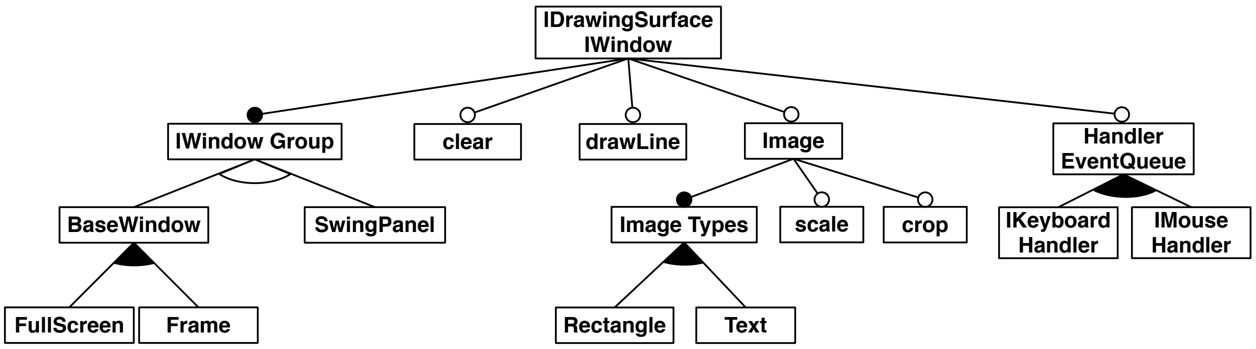


Figure 5.12: The Resulting Feature Model for the Running Example

feature model along with their relationships (step 1b). In addition, we know as a result of performing automated concernification which elements of the framework's API belong to which feature (step 2a). In addition, we need to group those API elements within a design realization model that is associated with the feature (step 2b). For step 2c, determining whether an API element belongs to the customization or usage interface, most API elements belong to the usage interface. Any customized framework class that was found in the examples needs to be converted to a partial class in the design realization model. Furthermore, any methods that are implemented in the customized framework class need to have a corresponding partial operation. The same applies to each interface. In our example, both handler interfaces need to be implemented by the user and as such a partial class implementing the respective interface needs to be added to the design realization model for each feature.

5.6 Implementation

We implemented the algorithm in order to be able to perform automated concernification automatically. This serves as a validation that the algorithm does indeed work in practice, especially with frameworks of larger size. We describe the different components and report on the performance of automated concernification. The algorithm implementation is built by reusing backend components of the TouchCORE tool, i.e., the metamodels and code providing CORE and RAM. To fully support concernification at the tool level, several enhancements had to be made to the TouchCORE tool. We describe these first before outlining the components of the algorithm implementation.

5.6.1 Supporting Concernification in TouchCORE

Prior to the research described in this thesis, TouchCORE initially only supported structural design modelling and structural composition with class diagrams. Behavioural design modelling and behavioural composition using sequence diagrams was added to TouchCORE [2, 97–99, 101]. This allows a modeller to reuse a concernified framework and define structure and behaviour using it at the modelling level. The following subsections describe the enhancements made to the TouchCORE tool in detail.

5.6.1.1 Support for Importing Implementation Classes into a Design Model

The classes and methods of a framework are already defined and implemented in code. These code elements should therefore not be re-defined at the modelling level. Instead, their functionality should nevertheless be accessible from within design models. To this aim, TouchCORE was extended to allow importing existing classes—called *implementation classes*—into a model [99].

To distinguish regular classes and implementation classes, the metamodel of class diagrams in RAM was extended with a new classifier type `ImplementationClass` that has a fully qualified name in addition to its simple name. This also allows abstracting from a specific programming language. Many programming languages use generics. To support genericity, an `ImplementationClass` can have `TypeParameters` that have a name and can have a generic type that restricts the possible types that are used. A `TypeParameter` is itself a `Type` since it might be used as a return or parameter type of an operation. Furthermore, an implementation class can have operations and attributes (i.e., public constants) like a regular class. It can, however, not have any outgoing associations. Associations are restricted to be unidirectional originating from a regular class to an implementation class.

An undergraduate student implemented an importer for Java that allows a designer to load classes from the Java Platform or an external JAR file. Once a class is imported, the user can then import specific operations of the class. By not importing the complete class with all its public attributes and operations, the user is not overloaded. The user only imports and sees those elements that are actually used. To import an implementation class in TouchCORE, the user only needs to specify the JAR file of the framework and can then proceed to import the desired classes and operations he intends to use. To import a class from the Java Platform, no JAR file needs to be specifically loaded. In case the user imports an operation with a return or parameter type that refers to an implementation class that has not been imported yet, TouchCORE automatically imports and adds them to the model.

5.6.1.2 Support for Code Generation

In MDE, models are not used for documentation only, but are meant to be refined and transformed until they can be executed, or code can be generated from them. Following this philosophy, TouchCORE was extended with a code generator which generates Java code from design models [118]. For each class in the design class model, a corresponding Java class is generated along with its structure (attributes and methods). The method's behaviour is generated from the sequence diagrams.

This allows a modeller to use a concernified framework at the modelling level and then generate code for the design. No code is generated for implementation classes. The code already exists and the existing reusable artefact can be included into the build path via traditional means. To this effect, any regular class imports any corresponding implementation class when referencing it by making use of it in its structure or behaviour.

The code generator is implemented using Acceleo [43], an open source code generator environment. It provides a template-based language to perform model-to-text (M2T) transformations.

5.6.1.3 Support for Traceability

Traceability support is crucial for tools that deal with separation of concerns to understand the detailed interactions between concerns. In the context of concernification, it is essential to understand which feature certain design elements relate to in a composed model of a framework's concern interface. We extended TouchCORE with support for traceability [103].

In the CORE metamodel, a new super-class `CORETraceableElement` was introduced that makes it possible to mark elements as traceable. For those models that represent a composed one, `COREModel` now contains a list of woven models. A `COREWovenModel` refers to the original model that was woven into the model. In addition, the woven model contains a mapping of elements that were woven. It maps from the source model element of the original model to the target model element in the composed model. Woven models are hierarchical in order to be able to maintain traceability of models that itself were composed with another model (i.e., by reusing another concern).

The weaver was extended to support the creation of the new data structure. Weaving is done in pairs of two models. At the end of weaving, in the post-processing phase, a new instance of `COREWovenModel` is created and its name initialized to the model that was woven. This is done to obtain the name in case the original model cannot be accessed, which is also set at this point. The mapping of woven elements is established based on the internal weaving information the weaver maintains during the weaving process [101].

In the GUI of TouchCORE, whenever a model contains information about other models that were woven into it, a tracing view is shown that lists the woven models by name. To see which elements were woven from a certain model, the user can select one or more entries and the tool highlights the corresponding woven elements in different colours. This allows the user to understand where certain elements came from.

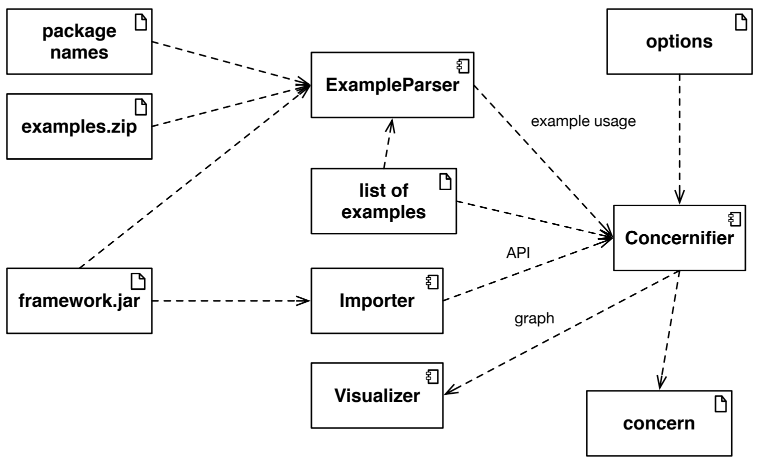


Figure 5.13: Overview of the Main Components of the *Automated Concernification* Implementation and the Flow of Data

5.6.2 Overview

Figure 5.13 shows the overview of the components of the implementation as well as their inputs and outputs. The artefacts required to concernify a framework are the code of the framework and the examples showcasing the use of the framework's API. The framework's code must be provided in form of a packaged JAR file containing the compiled source code. The examples are provided as source code in a ZIP archive.

There are three main components. The *Importer* is responsible for importing the classes and their elements. The *Example Parser* parses the example source code to detect the API usage, and the *Concernifier* performs the graph concernification to produce at the end a concern with the feature model and the realization design models for each feature. In addition, we implemented a graph visualizer which visualizes the DAG data structure that is used in the concernifier. This makes it possible to inspect the manipulated data structure in between steps.

5.6.3 Importer

The *Importer* processes one or more JAR files of a framework to extract the framework's API. To do this, a RAM *StructuralView* is created representing the class diagram with the classes and their attributes and operations as well as the hierarchy between classes. The use of class diagrams in RAM abstracts away from the programming language the framework is written in. Additional importers can therefore be implemented to provide extraction of frameworks written in other object-oriented programming languages.

In order to gather all classes of the framework, the *Reflections* library [94] is used. The *Java*

Reflection API makes it possible to extract information about all the classes of the framework and their properties and create the corresponding classes in RAM. Besides public classes, also non-public classes which have public subclasses are extracted. Methods with protected and public visibility are extracted as well. Overriding methods are ignored, since the overridden method in the superclass is the one that is relevant. The only exception is when an overriding method increases the visibility of the method, allowing instances of the subclass to call a method that is otherwise hidden. For class fields, only those that are public are extracted. Public fields are commonly used for constants. RAM currently does not support the declaration of which exception is thrown by an operation. The exception class itself is imported and added to the structural view, but in addition, the *Importer* stores in an additional data structure the information of which operation throws which exception.

5.6.4 Example Parser

The *Example Parser* parses each source file from the examples ZIP archive and determines the usage of all framework elements within the given example file. To parse the source files, the `ASTParser` provided by the *Eclipse JDT* [42] (the *Java Development Toolkit* used for Java programming in Eclipse) is used. The *Example Parser* determines a mapping of the classes to their used elements. Furthermore, it determines those classes from the framework that were customized by the example as well as their overridden methods, if any. The framework and example package names are used to distinguish classes belonging to the framework and examples.

It is common that an example consists of many (auxiliary) classes that together describe an example. Our algorithm assumes that an example has one main class that represents the example. In the case of a simple example this is the class that defines the `main` method. The list of entry points (examples) is provided to the *Example Parser*. The usage of an example is the union of usages of each class contributing to an example.

5.6.5 Concernifier

The *Concernifier* is responsible for performing the DAG concernification algorithm. It takes as input the results from the *Importer* and *Example Parser* as well as the options containing the settings for the hypotheses. It implements the algorithm definition as outlined in Section 5.5. The DAG data structure is implemented on top of the *JGraphT* library [79]. It allows one to create custom node and edge types. Furthermore, *JGraphT* provides out-of-the-box utilities for graph traversal, transitive reduction and cycle detection which we require for concernification.

At the end, the final DAG is converted into a concern with a feature model based on the DAG as well as realization design models for each non-empty node. Each design model contains the API elements that were determined to belong to the corresponding feature.

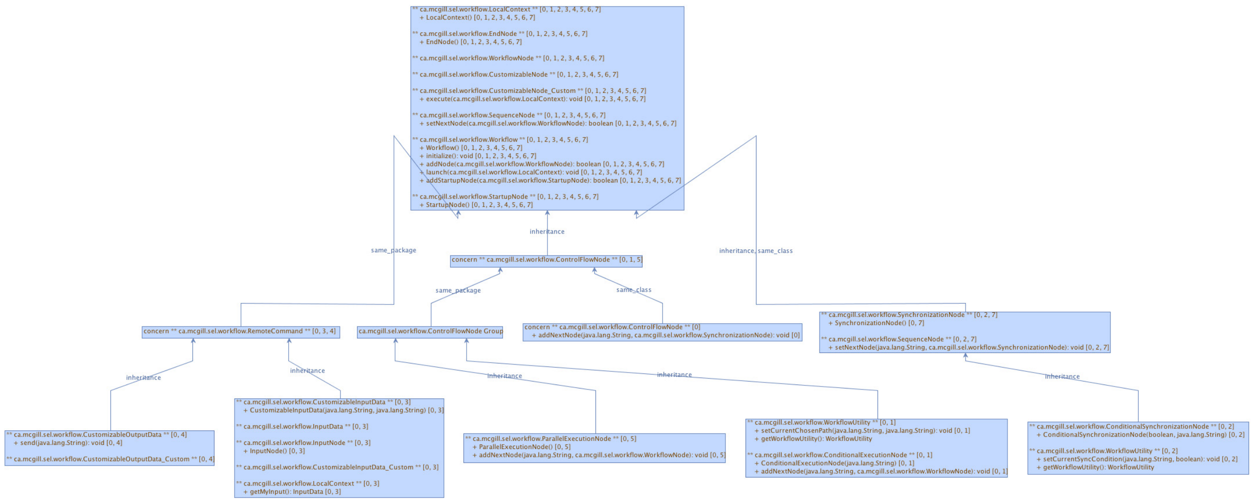

** ca.mcgill.sel.workflow.LocalContext ** [0, 1, 2, 3, 4, 5, 6, 7]
+ LocalContext() [0, 1, 2, 3, 4, 5, 6, 7]
** ca.mcgill.sel.workflow.EndNode ** [0, 1, 2, 3, 4, 5, 6, 7]
+ EndNode() [0, 1, 2, 3, 4, 5, 6, 7]
** ca.mcgill.sel.workflow.WorkflowNode ** [0, 1, 2, 3, 4, 5, 6, 7]
** ca.mcgill.sel.workflow.CustomizableNode ** [0, 1, 2, 3, 4, 5, 6, 7]
** ca.mcgill.sel.workflow.CustomizableNode_Custom ** [0, 1, 2, 3, 4, 5, 6, 7]
+ execute(ca.mcgill.sel.workflow.LocalContext): void [0, 1, 2, 3, 4, 5, 6, 7]
** ca.mcgill.sel.workflow.SequenceNode ** [0, 1, 2, 3, 4, 5, 6, 7]
+ setNextNode(ca.mcgill.sel.workflow.WorkflowNode): boolean [0, 1, 2, 3, 4, 5, 6, 7]
** ca.mcgill.sel.workflow.Workflow ** [0, 1, 2, 3, 4, 5, 6, 7]
+ Workflow() [0, 1, 2, 3, 4, 5, 6, 7]
+ initialize(): void [0, 1, 2, 3, 4, 5, 6, 7]
+ addNode(ca.mcgill.sel.workflow.WorkflowNode): boolean [0, 1, 2, 3, 4, 5, 6, 7]
+ launch(ca.mcgill.sel.workflow.LocalContext): void [0, 1, 2, 3, 4, 5, 6, 7]
+ addStartupNode(ca.mcgill.sel.workflow.StartupNode): boolean [0, 1, 2, 3, 4, 5, 6, 7]
** ca.mcgill.sel.workflow.StartupNode ** [0, 1, 2, 3, 4, 5, 6, 7]
+ StartupNode() [0, 1, 2, 3, 4, 5, 6, 7]
same_package
inheritance
inheritance, same_class
concern ** ca.mcgill.sel.workflow.ControlFlowNode ** [0, 1, 5]
same_package
same_class
concern ** ca.mcgill.sel.workflow.RemoteCommand ** [0, 3, 4]
ca.mcgill.sel.workflow.ControlFlowNode Group
concern ** ca.mcgill.sel.workflow.ControlFlowNode ** [0]
+ addNextNode(java.lang.String, ca.mcgill.sel.workflow.SynchronizationNode): void [0]
** ca.mcgill.sel.workflow.SynchronizationNode ** [0, 2, 7]
+ SynchronizationNode() [0, 7]
** ca.mcgill.sel.workflow.SequenceNode ** [0, 2, 7]
+ setNextNode(java.lang.String, ca.mcgill.sel.workflow.SynchronizationNode): void [0, 2, 7]
inheritance
inheritance
inheritance
inheritance
inheritance
** ca.mcgill.sel.workflow.CustomizableOutputData ** [0, 4]
+ send(java.lang.String): void [0, 4]
** ca.mcgill.sel.workflow.CustomizableOutputData_Custom ** [0, 4]
** ca.mcgill.sel.workflow.CustomizableInputData ** [0, 3]
+ CustomizableInputData(java.lang.String, java.lang.String) [0, 3]
** ca.mcgill.sel.workflow.InputData ** [0, 3]
** ca.mcgill.sel.workflow.InputNode ** [0, 3]
+ InputNode() [0, 3]
** ca.mcgill.sel.workflow.CustomizableInputData_Custom ** [0, 3]
** ca.mcgill.sel.workflow.LocalContext ** [0, 3]
+ getMyInput(): InputData [0, 3]
** ca.mcgill.sel.workflow.ParallelExecutionNode ** [0, 5]
+ ParallelExecutionNode() [0, 5]
+ addNextNode(java.lang.String, ca.mcgill.sel.workflow.WorkflowNode): void [0, 5]
** ca.mcgill.sel.workflow.WorkflowUtility ** [0, 1]
+ setCurrentChosenPath(java.lang.String, java.lang.String): void [0, 1]
+ getWorkflowUtility(): WorkflowUtility
** ca.mcgill.sel.workflow.ConditionalExecutionNode ** [0, 1]
+ ConditionalExecutionNode(java.lang.String) [0, 1]
+ addNextNode(java.lang.String, ca.mcgill.sel.workflow.WorkflowNode): void [0, 1]
** ca.mcgill.sel.workflow.ConditionalSynchronizationNode ** [0, 2]
+ ConditionalSynchronizationNode(boolean, java.lang.String) [0, 2]
** ca.mcgill.sel.workflow.WorkflowUtility ** [0, 2]
+ setCurrentSyncCondition(java.lang.String, boolean): void [0, 2]
+ getWorkflowUtility(): WorkflowUtility

Figure 5.14: Screenshot of the *Visualizer* component visualizing the final graph after concernifying the *Workflow* concern framework

5.6.6 Visualizer

To visualize the final DAG or intermediate results we implemented the *Visualizer* component, which was helpful mostly during the implementation and debugging of the concernification algorithm implementation. *JGraphT* provides an adapter class `JGraphXAdapter` allowing the graph to be visualized by *JGraphX* [55], a graph visualization library for *Swing*.

Due to the fact that our edges show dependency links, the root is the node with no outgoing edges. *JGraphX* provides a hierarchical layout with the option to configure the orientation in which it is layed out, however, due to a bug with our required orientation (south) the graph is drawn off-screen. We fixed this in the implementation of the corresponding `mxHierarchicalLayout` class. Figure 5.14 shows a screenshot of the *Visualizer* displaying the resulting graph after concernifying the *Workflow* concern framework.

5.6.7 Performance

We report on the performance of the automated concernification prototype implementation. Specifically, we report on the performance of the individual components and provide metrics on different properties. We provide the average execution time we measured of running the task 110 times and discarding the first 10 runs to account for code loading and initialization when starting the Java virtual machine. We report on performance of processing the *Minueto* framework which we discussed in Section 3.5. Furthermore, we report on the *Workflow* concern, a workflow engine for reactive systems, and the Android SDK as a bigger framework than the other two. Both Workflow

Table 5.1: Performance Metrics of the *Importer*

Framework	Relevant Classes (total)	Visible Elements (total)	Avg. Elements per Class	Avg. *Importer* time (ms)
Workflow	18 (28)	44 (48)	2.44	10
Minueto	41 (60)	418 (466)	10.19	18
Android	3100 (3577)	47559 (50007)	15.34	1158

Table 5.2: Performance Metrics of the *ExampleParser*

Framework	Examples	Files	LOC	Avg. *ExampleParser* time (ms)
Workflow	8	11	436	228
Minueto	30	59	4388	1267
Android Notifications	12	83	7143	3630

and Android are used for validation and will be described and discussed in Chapter 6 (Workflow in Section 6.1 and Android in Section 6.3).

Table 5.1 provides the metrics of importing the JAR file of a framework and producing the corresponding structural view for its API. The number of visible classes includes those that are not public but have public sub-types since they potentially provide a shared implementation. The number of elements include public fields and public or protected methods.

Table 5.2 shows the metrics of extracting the usage of the framework API in the example source code files. Examples might share implementations of common functionality or utilities. Hence, the number of files is greater than the number of (runnable) examples. The execution times of the *ExampleParser* show that parsing source code takes considerably longer than analyzing the binary code (as is done by the *Importer*).

Table 5.3 lists the performance of the main concernification algorithm. We show in addition the number of nodes and edges in the graph after the individual phases (see Section 5.5.2). Phase 1 concludes after initializing the individual hierarchy trees, phase 2 is finished after populating the DAG with all available dependency information, and phase 3 ends after the DAG simplification. At this point, the DAG is a tree with $n - 1$ edges.

Table 5.3: Performance Metrics of the *Concernifier*

Framework	**Nodes (1)**	**Edges (1)**	**Nodes (2)**	**Edges (2)**	**Nodes (3)**	**Avg. *Concernifier* time (ms)**
Workflow	18	13	12	17	11	3
Minueto	41	13	49	79	21	10
Android (notifications)	2984	1335	48	62	15	73
Android	2984	1335	3123	7665	2795	3522

5.7 Summary

This chapter described in detail the algorithm for automated concernification. The algorithm is designed based on the guidelines we determined and confirmed with the developers of Minueto. The most important inputs to the algorithm are the relationships between API elements and the examples showcasing the use of a framework's API. The usefulness of the result highly depends on the quality of the examples. In the best case, there is at least one example exclusively showcasing one feature. However, this is unrealistic since examples often showcase many features at once. In combination with the API structure and dependency information available from the API we can mitigate this. We established hypotheses about the quality of the examples that can be enabled or disabled that then influence the steps performed in the algorithm.

The algorithm is based on a DAG data structure where a node represents a potential feature. Edges represent dependency information (such as inheritance). The algorithm is divided into three main phases. In the first phase, the DAG is populated with individual trees for all inheritance hierarchies of the framework's API. Operations which are used in less examples than their class are pulled out since they could potentially belong to another feature. The second phase connects the individual hierarchy trees by considering the package structure and merges those nodes with the same example usage. At this point, the DAG has one root. In addition, the DAG is populated with cross-references. The goal of the algorithm is to produce a tree that represents the feature model. Therefore, in the third phase, simplification of the DAG is performed. The simplification steps ensure that the integrity of the framework API is maintained.

The simplification decision might involve removing an edge. The decision which edge to remove is based on a priority ordering of the edge types we use (e.g., inheritance is considered more important than a structural grouping). This decision might not always be the right one depending on the specific case. However, we know that there is no perfect solution for our algorithm, because there exists no unique feature model that needs to be found. Rather, features can be organized in different ways, but in the end the possible configurations remain the same. Furthermore, the way in which features are organized is subject to the view of the designer. As such, the same features might be organized into different feature models by different people. Finally, whether a feature should be decomposed further into sub-features sometimes also depends on the desired level of detail. For example, depending on the size of the API, splitting the methods of a class into separate features could result in large feature models. This increases the cognitive effort a user has to make, which can make it more difficult to make an appropriate feature selection. As such, our algorithm provides the option to minimize the features, i.e., features of methods that were not merged are ultimately moved up into the feature corresponding to the method's class.

In general, the idea is that our algorithm provides an initial concern interface for a framework. The developer as the domain expert can then further fine-tune the feature model and make adjustments as is seen fit. For this, proper tool support is essential to be able to see the features and their corresponding API and be able to quickly rearrange features or move API elements around into different features. Additionally, such a tool needs to ensure that the integrity of the API is maintained and introduce cross-tree constraints if needed. However, cross-tree constraints should be minimized as they are more difficult to understand and, for example, negatively impact the performance of impact model evaluations.

To ensure that the algorithm produces accurate results, the next chapter will validate it in two ways. First, by ensuring that the synthesized concern interface for a known concern produces an accurate result, and its examples can be designed with only the API subset related to the features the example(s) showcases. Similarly, we can evaluate the accuracy of the result for Minueto based on the validation result from Chapter 4. Second, to ensure that automated concernification provides an accurate result for a larger framework that is widely used, the next chapter will also describe a study using the Android SDK.

6 Automated Concernification Validation

The previous chapter defined the algorithm to perform *automated concernification*. Given runnable examples that showcase how to use a framework, the API of a framework can be *concernified*. In order to verify the accuracy of the results the automated concernification produces, we need to know the feature model of a framework to compare the result to. In Chapter 4 we conducted a study with the Minueto developers to validate our own Minueto feature model and determine the developers own feature models. In addition, there is the *Workflow* concern which has been developed as a concern first, i.e., a feature model along with design realization models for each feature already exists.

In this chapter we start by evaluating the *Workflow* concern in Section 6.1. As it was conceived as a concern, we know with 100% certainty what the features are. This allows us to create examples that showcase the use of each feature individually, as well as examples that illustrate the use of the features in combination. In the first section of this chapter, we therefore run the automated concernification algorithm on code generated from the *Workflow* concern to extract a feature model. We then compare the extracted feature model with the known feature model and report on the accuracy of the determined features as well as the API and its integrity, i.e., whether the examples have access to the subset of the API related to the features. In Section 6.2 we evaluate the accuracy of the feature detection on Minueto. We compare the result of automatic concernification with our validated feature model from Chapter 4.

Finally, in Section 6.3 we perform automated concernification on a third framework—the Android Notifications API of the Android SDK—and conduct a user study to gather feedback on the results. We report on the feedback and analysis of the data we gathered. This chapter concludes with a summary in Section 6.4.

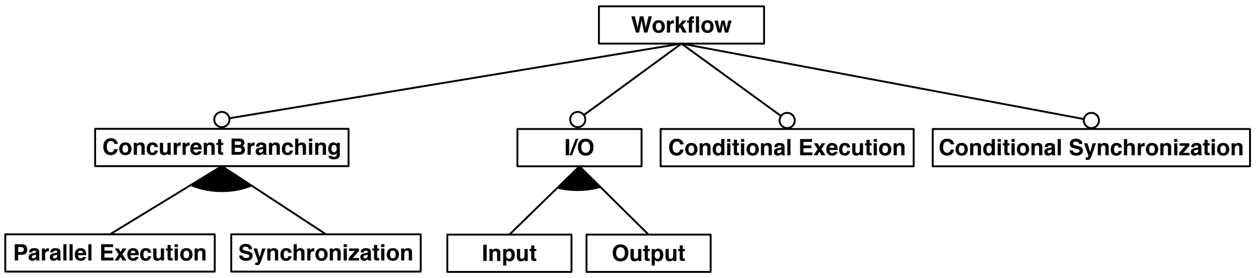


Figure 6.1: Workflow Feature Model

6.1 Evaluation of the Workflow Concern

The *Workflow* concern provides a workflow engine for reactive systems. It defines a workflow executor and different kinds of nodes to describe a workflow. Optional features contribute additional functionality for branching, concurrency, waiting, and input and output handling. The *Workflow* concern models the workflow engine of the system, i.e., it does not deal with the client side interacting with the system.

The *Workflow* feature model is shown in Figure 6.1. It contains the known features and their relationships. There are no mutually exclusive features.

The *Workflow* concern was conceived as a concern from the beginning. Therefore, we know the true features of the *Workflow* concern. In addition, we know the API for each feature. We can therefore use the *Workflow* concern to evaluate the accuracy of the automated concernification algorithm. To be able to perform automated concernification, the complete framework API and examples showcasing the use of the API are required. As such, we selected all features and used the resulting composed model generated by the weaver to generate code. This code represents the API of the entire *Workflow* framework. The composed model has a customization interface. When generating code, any partial class or method is marked as *abstract* to require user implementation[1].

We then created examples showcasing the use of the API and its different features. We created an example for the root feature, each leaf feature, and one showcasing the use of all features in combination. In total, there are 8 examples which are as follows:

1. **SimpleWorkflow**: Showcases a simple workflow using only the API provided by the root feature. The simple workflow consists of a start node, a sequence of responsibilities, and an end node.

2. **ParallelWorkflow**: Showcases a workflow using parallel execution (and fork) provided by the *ParallelExecution* feature.

[1]Constructors in Java can, however, not be abstract.

3. **SynchronizedWorkflow**: Showcases a workflow synchronizing two paths (and join) provided by the *Synchronization* feature.

4. **ConditionalExecutionWorkflow**: Showcases a workflow conditionally forking (or fork) provided by the *ConditionalExecution* feature.

5. **ConditionalSynchronizationWorkflow**: Showcases a workflow conditionally synchronizing a path (waiting place) provided by the *ConditionalSynchronization* feature.

6. **InputWorkflow**: Showcases a workflow handling input provided by the *Input* feature.

7. **OutputWorkflow**: Showcases a workflow sending output provided by the *Output* feature.

8. **FullWorkflow**: Showcases a workflow making use of all features together in combination.

Based on the knowledge of what the examples showcase and the API they make use of, we set the options of the algorithm such that all hypotheses hold. Performing *automated concernification* using the framework and the example code allows us to compare the original feature model with the automatically determined one. In summary, it allows us to evaluate whether:

1. all original features were detected by the algorithm, the API elements were assigned to the same feature than in the original, and

2. the API elements constituting the customization interface in the original were detected.

In this section we evaluate these three points by comparing the original *Workflow* concern (W_{orig}) and the automatically concernified *Workflow* (W_{ac}) to determine whether using examples showcasing the features of a framework *automated concernification* provides an accurate concern interface of the framework. The designer should only need to make minimal "cosmetic" adjustments, if any.

6.1.1 Accuracy of Feature Detection

Figure 6.2 shows the two feature models of *Workflow*, the original feature model (W_{orig}) at the top and the feature model obtained after performing automated concernification (W_{ac}) on the bottom. The names of the features in W_{ac} were manually determined and mostly reflect the names of the classes that are contained in the API of the respective features.

In addition, to better understand the differences between the two feature models, the resulting graph after performing automated concernification including the API elements is also shown in Figure 6.3.

There are three main differences. First, there exist two additional features in W_{ac}. The first feature—named *Control Flow and Synchronization*—contains a single operation from the abstract

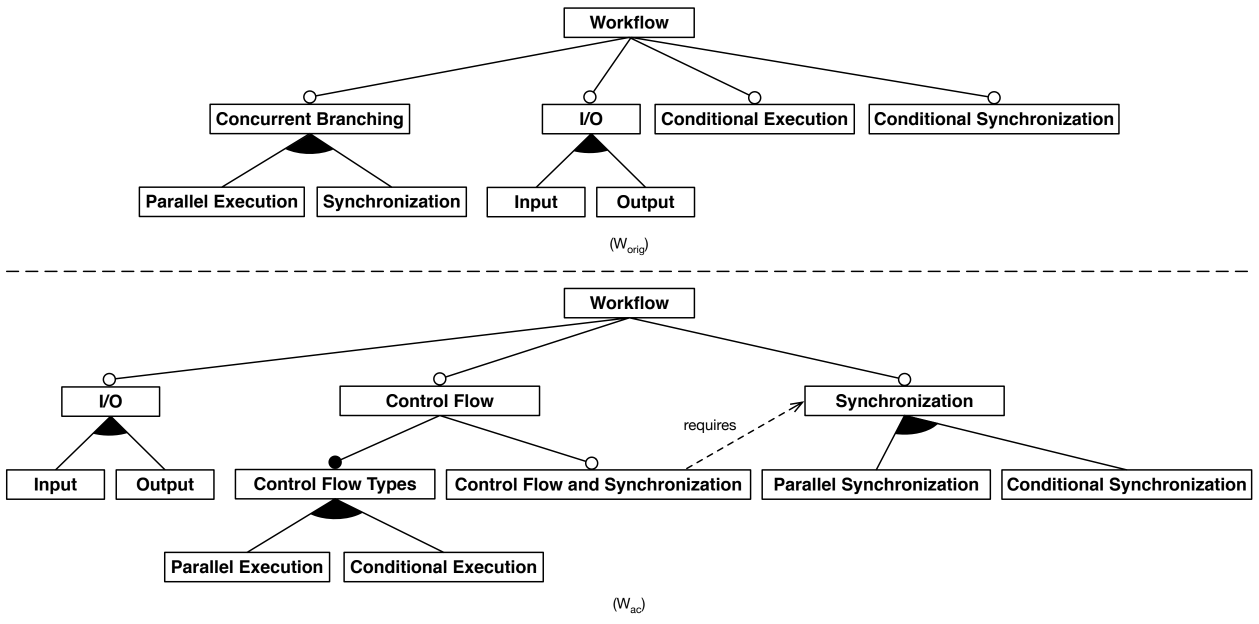


Figure 6.2: Feature Models of W_{orig} (top) and W_{ac} (bottom)

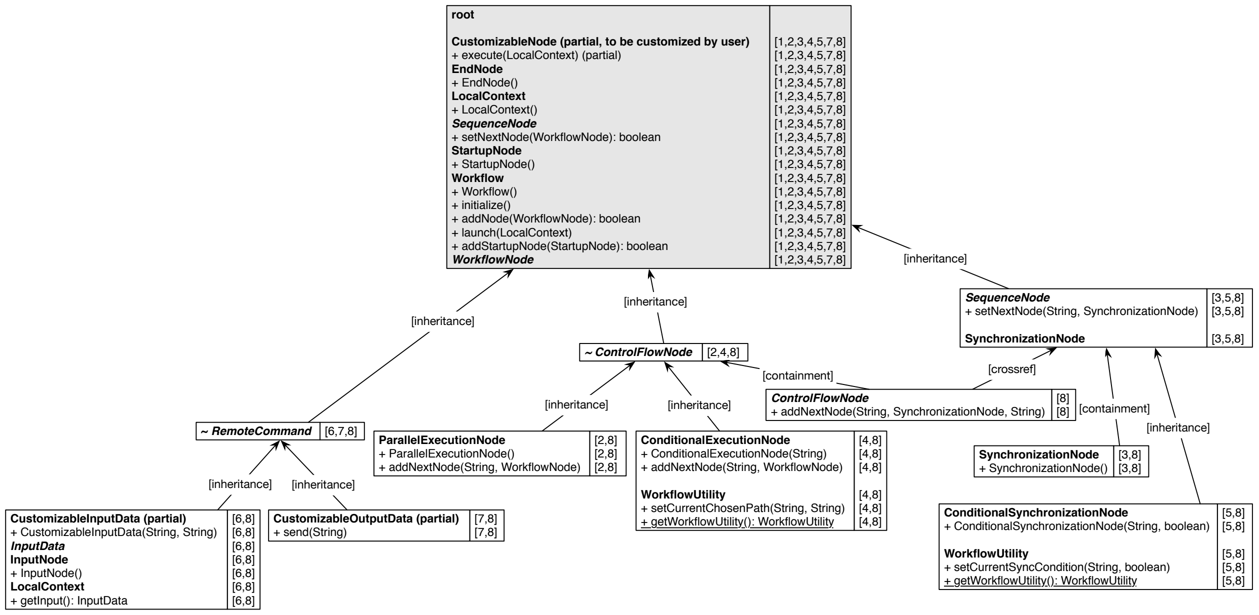

root
CustomizableNode (partial, to be customized by user) [1,2,3,4,5,7,8]
+ execute(LocalContext) (partial) [1,2,3,4,5,7,8]
EndNode [1,2,3,4,5,7,8]
+ EndNode() [1,2,3,4,5,7,8]
LocalContext [1,2,3,4,5,7,8]
+ LocalContext() [1,2,3,4,5,7,8]
SequenceNode [1,2,3,4,5,7,8]
+ setNextNode(WorkflowNode): boolean [1,2,3,4,5,7,8]
StartupNode [1,2,3,4,5,7,8]
+ StartupNode() [1,2,3,4,5,7,8]
Workflow [1,2,3,4,5,7,8]
+ Workflow() [1,2,3,4,5,7,8]
+ initialize() [1,2,3,4,5,7,8]
+ addNode(WorkflowNode): boolean [1,2,3,4,5,7,8]
+ launch(LocalContext) [1,2,3,4,5,7,8]
+ addStartupNode(StartupNode): boolean [1,2,3,4,5,7,8]
WorkflowNode [1,2,3,4,5,7,8]
[inheritance]
SequenceNode [3,5,8]
+ setNextNode(String, SynchronizationNode) [3,5,8]
SynchronizationNode [3,5,8]
[inheritance]
[inheritance]
~ ControlFlowNode [2,4,8]
[containment]
[crossref]
ControlFlowNode [8]
+ addNextNode(String, SynchronizationNode, String) [8]
[containment]
[inheritance]
[inheritance]
[inheritance]
~ RemoteCommand [6,7,8]
ParallelExecutionNode [2,8]
+ ParallelExecutionNode() [2,8]
+ addNextNode(String, WorkflowNode) [2,8]
ConditionalExecutionNode [4,8]
+ ConditionalExecutionNode(String) [4,8]
+ addNextNode(String, WorkflowNode) [4,8]
WorkflowUtility [4,8]
+ setCurrentChosenPath(String, String) [4,8]
+ getWorkflowUtility(): WorkflowUtility [4,8]
SynchronizationNode [3,8]
+ SynchronizationNode() [3,8]
[inheritance]
[inheritance]
CustomizableInputData (partial) [6,8]
+ CustomizableInputData(String, String) [6,8]
InputData [6,8]
InputNode [6,8]
+ InputNode() [6,8]
LocalContext [6,8]
+ getInput(): InputData [6,8]
CustomizableOutputData (partial) [7,8]
+ send(String) [7,8]
ConditionalSynchronizationNode [5,8]
+ ConditionalSynchronizationNode(String, boolean) [5,8]
WorkflowUtility [5,8]
+ setCurrentSyncCondition(String, boolean) [5,8]
+ getWorkflowUtility(): WorkflowUtility [5,8]

Figure 6.3: Resulting Graph of Workflow after *Automated Concernification*

class `ControlFlowNode` which is required to be used when the next node after a control flow node is a `SynchronizationNode`. This means this method is only required if the features *Control Flow* and *Synchronization* are used. In W_{orig} this is modelled as a feature interaction between *Synchronization* and *Parallel Execution*/*Conditional Execution*. The feature interaction realization model that realizes the respective features (see Section 2.4.5) introduce this method. However, our algorithm does not detect this, and it is left to the user to move those API elements into such a realization model. The second feature—named *Parallel Synchronization*—contains the constructor of `SynchronizationNode`. It was pulled out of the class because it is used in less examples than the class itself. `SynchronizationNode` is sub-classed by `Conditional-SynchronizationNode`, therefore, when using the latter the former is also used. This is an improvement over the original feature model where selecting only *Conditional Synchronization* also gives access to the constructor of `SynchronizationNode`.

The third difference relates to the groupings of the features. In W_{orig} the grouping feature *Concurrent Branching* is a logical grouping, i.e., it does not provide any realization model. The modeller of *Workflow* chose to group those features related to concurrent branching of workflows. Our algorithm favours inheritance relationships. The realization model of *Conditional Synchronization* extends the realization model of *Synchronization*. This is in order for the former to extend the `SynchronizationNode` class of the latter. Therefore, due to the inheritance relationship, *Conditional Synchronization* is a sub-feature of *Synchronization*. While the modeller of W_{orig} placed importance of showing concurrent branching features together, the resulting feature model of W_{ac} shows features for branching and synchronization together.

6.1.2 Accuracy of API Element Assignment

To ensure that API elements are actually assigned to the correct feature, we compare the usage interface of each feature of W_{orig} with that of the respective feature of W_{ac}. This answers the question whether the integrity of the API is maintained.

There are two main differences. First, the method `WorkflowUtility.getWorkflow-Utility()` is located in the root feature *Workflow* in W_{orig}. This is because the root feature uses this method internally, whereas it needs to be called by the user when using *Conditional Execution*/*Synchronization*. In W_{ac} this method was identified as a utility node and copied into the features where it is needed (*Conditional Execution* and *Conditional Synchronization*). As a result, this method is in the usage interface in those two features where the user actually needs to use it.

The second difference relates to the fact that the *Workflow* concern reuses another concern (*Command* realizing the command pattern [44]) which requires the `execute()` method of a command to be public because it is a callback method. Both *Input* and *Output* features provide a concrete `RemoteCommand` sub-class which requires the `execute()` method to be imple-

Table 6.1: API Metrics based on Feature Selection of W_{ac}

Example	Feature Selection[2]	Classes	Methods	Percentage of Complete API
`SimpleWorkflow`	Workflow	8	10	38%
`ParallelWorkflow`	Parallel Execution	10	12	47%
`SynchronizedWorkflow`	Parallel Synchronization	9	12	45%
`ConditionalExecution-Workflow`	Conditional Execution	11	14	53%
`Conditional-Synchronization-Workflow`	Conditional Synchronization	11	15	55%
`InputWorkflow`	Input	13	14	57%
`OutputWorkflow`	Output	11	11	47%
`FullWorkflow`	Parallel Execution, Conditional Execution, Parallel Synchronization, Conditional Synchronization, Input, Output	21	25	98%[3]

mented. They are callback methods meaning that the user does not explicitly call them. However, because the user needs to provide application-specific behaviour, they are part of the customization interface. These two methods were removed during *automated concernification* as we specified hypothesis H3 (see Section 5.4) to hold. This shows an additional benefit of concern interfaces: if the concern interface is used instead of the Java API, public methods coming from frameworks that the framework reuses would be excluded in the usage interface, therefore reducing the cognitive effort required by the user.

In summary, of the used API, 98% of API elements were assigned to the correct feature. Ta-

[2]Ancestors of those features were omitted for brevity as they will be selected automatically.
[3]Due to the missing method in `ControlFlowNode`

ble 6.1 lists the number of API elements that are part of the usage interface of the generated API for each feature selection from W_{ac}. Ancestors of those features were omitted for brevity as they will be selected automatically. This shows the reduction in the size of the API, reducing the cognitive effort required by the user.

In addition, we also evaluated whether the code for each example has access to the required API when selecting only the features of W_{ac} that the example showcases. For each example we composed a tailored version of *Workflow* (W_{ac}) based on the feature selection. We then compared for each example whether all API elements used by the example are available in the customized version. In summary, all API elements are available with the exception of one method used in the `FullWorkflow` example. The method in question is `ControlFlowNode.addNext-Node(String, SynchronizationNode, String)` which is located in the additional feature *ControlFlowNode and Synchronization*. W_{orig} does not have this feature and therefore we did not select it to test the `FullWorkflow` example. As discussed above, this method needs to be provided by a feature interaction model and therefore would not be a distinct feature.

6.1.3 Accuracy of Customization Interface

To evaluate whether all API elements were detected that need to be part of the customization interface, we also compared the customization interface of W_{orig} and W_{ac}. W_{orig} provides in total three partial classes. `CustomizableNode` (with a partial method `execute()`) in the root feature *Workflow*, `CustomizableInputData` (with a partial constructor `Customizable-InputData(String, String)`) in *Input*, and `CustomizableOutputData` in *Output*. Our concernification algorithm added all five elements (three classes and two methods) to the customization interface of the respective features. The only difference in W_{ac} consists of an additional abstract class in the inheritance hierarchy of the API. The presence of this class is due to the way the code was generated for W_{orig}. Because there are no partial elements in Java, the partial methods and classes were generated as abstract, therefore requiring the user to provide implementations for those elements.

6.2 Evaluation of Feature Model Accuracy for Minueto

To further evaluate the accuracy of the detected features and their organization within the feature model by *automated concernification*, we also performed an evaluation with Minueto. In Chapter 3 we introduced our own feature model of Minueto which we validated with the developers of Minueto (see Chapter 4). In this section we use our corrected feature model of Minueto (M_I, see Figure 4.9 on page 58) to compare it with the automatically determined one (M_{ac}).

Along with the Minueto framework the developers provide 31 examples that showcase how

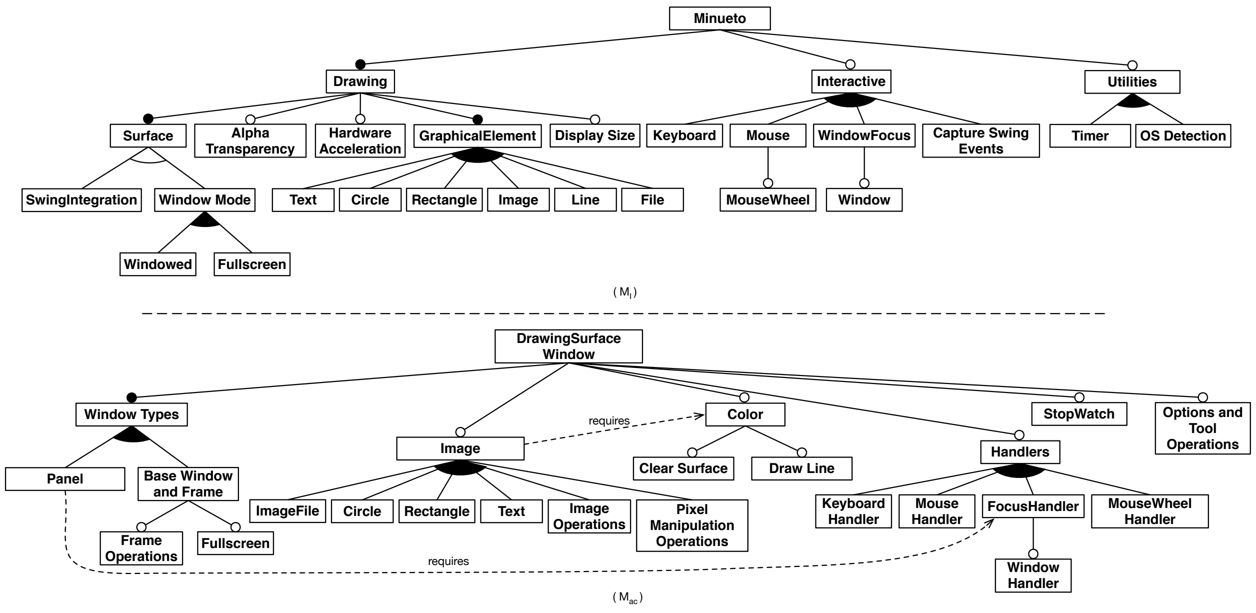


Figure 6.4: Minueto Feature Models of M_I (top) and M_{ac} (bottom)

to use the API. While working with the examples, we identified a problem with them based on the knowledge we gained from familiarizing ourselves with the framework and interviewing the developers (see Chapter 4). Two examples (`HelloWorld` and `TextDemo`) instantiate and make calls to `MinuetoEventQueue` without using this instance in combination with a respective handler (i.e., the `MinuetoEventQueue` instance is never passed to any method to register it for handling specific events). As a result, the call to `hasNext()` always returns `false` and the body of the loop is never executed (thus the `handle()` method is never invoked). In other words, the instantiation of the `MinuetoEventQueue` is useless in those two examples. This violates our hypothesis 4 stating that the examples make correct use of the framework's API. To address this, we removed the use of `MinuetoEventQueue` from the `HelloWorld` and `TextDemo` examples.

While the examples do not use the full API, i.e., every single method of each class, the developers created these examples showcasing the features of Minueto. We therefore enabled all hypotheses to hold with the exception of hypothesis H1a because not all possible combinations are covered by the examples.

The feature model of M_I is re-shown for convenience on the top of Figure 6.4. The feature model obtained after running the automated concernification algorithm contains 45 features. This high number results from several methods being pulled out of their containing class. Reducing the size of the feature model by merging those operation nodes back to the parent (where the class is located) that were not merged with any other node results in 24 features. The minimized feature model is shown on the bottom of Figure 6.4. The names of the features reflect the names of the classes or interfaces (excluding the `Minueto` prefix), or the names of the operations the features provide.

We first discuss in detail the higher-level groupings of the feature model and then focus in on each group to compare them with each other.

Grouping Features While M_I has a grouping feature called *Drawing*, M_{ac} does not have it. However, this represents only a logical grouping for all drawing related features. In M_{ac}, the window types and graphical elements are grouped directly under the root feature. Both feature models contain a grouping feature for handling different events (*Interactive* in M_I and *Handlers* in M_{ac}). Furthermore, M_I groups *Timer* and *OS Detection* under a *Utilities* group. M_{ac} on the other hand has *StopWatch* and *Options and Tool Operations* directly connected to the root feature. The latter feature not only contains the method for OS detection from the `MinuetoTool` class, but also the methods of `MinuetoOptions` to enable alpha transparency and hardware acceleration. These are separate features in M_I.

Lastly, M_{ac} contains a dedicated feature for *Color*. The feature has two sub-features with methods of `MinuetoDrawingSurface` that require the `MinuetoColor` class. Due to transitive reduction their edge to the root feature (containing the class) was removed. Besides these two methods, several methods provided by the *Image* feature and its children (with the exception of *ImageFile*) require the `MinuetoColor` class as well. This shows that it could be a utility class and the designer can make this adjustment at the end.

Types of Window Surfaces Both feature models have a feature group for the types of windows that can be used (*Surface* in M_I and *Window Types* in M_{ac}). While the children of M_{ac} are in an OR-relationship, those of M_I are in an XOR-relationship. Since hypothesis H1a does not hold, this was not detected. If this hypothesis is set to hold, the algorithm determines that the children of *Window Types* are in an XOR-relationship because they are not used together.

The feature configuration further does not allow `MinuetoFullscreen` to be used without `MinuetoFrame` because it is located in the *Base Window and Frame* feature. This is due to the examples never showcasing the use of `MinuetoFullscreen` without also using `MinuetoFrame`.

The feature providing the `MinuetoPanel` class (*Panel*) in M_{ac} has several differences. First, it provides two methods from `MinuetoTool` to determine the display size (separate feature *Display Size* in M_I). Second, it provides the `MinuetoWindow.registerFocusHandler(...)` method which causes the *requires* cross-tree constraint to *FocusHandler*. And third, due to this method, it also provides the `MinuetoEventQueue` class which was identified as a utility node and merged into this feature.

Based on the validation we conducted with the Minueto developers (see Section 4.4), we know that `MinuetoFrame` and `MinuetoFullscreen` can be used separately, retrieving the display size is independent of the *Swing* integration, and that using the focus handler is possible for all window types. As a test, we corrected these inconsistencies in the examples by introducing a new `HelloWorldFullscreen` example (a copy of the `HelloWorld` example) using `MinuetoFullscreen` instead of `MinuetoFrame`. This example also uses the two `MinuetoTool` methods to determine the display size. Additionally, we added the use of the `MinuetoFocusHandler` to the example `HandlerDemo3`. The resulting feature model then contains both frame and fullscreen features as siblings under the *Base Window* grouping in an OR-relationship. Furthermore, the method to register a focus handler and `MinuetoEventQueue` are now provided by the *FocusHandler* feature. And lastly, the display size methods are identified as a separate feature and grouped together with the *Options and Tool Operations*.

Graphical Elements The *Image* feature in M_{ac} is optional. This is due to the method `drawLine(...)` of `MinuetoDrawingSurface` being used solely in two examples to draw lines. In M_I we identified it as a separate feature and placed it under *GraphicalElement*. Our algorithm did pull this method out into a separate node, but as an operation belonging to `MinuetoDrawingSurface` (in the root feature) it has no direct relationship to the other graphical elements. M_{ac} has two additional features with methods to draw within an empty surface (in *Image Operations*) and direct pixel manipulations (in *Pixel Manipulation Operations*). This shows a finer level of granularity, similar to what the level of detail of the feature model elaborated by the second *Minueto* developer (see Figure 4.6 on page 56). Lastly, M_I contains a sub-feature *Image* which provides the ability to create temporary images (i.e., empty surfaces that can be drawn into). In M_{ac} this is provided by the grouping feature *Image*.

Missed Features The one feature that our algorithm missed and that is present in M_I is *Capture Swing Events*. However, based on our study with the *Minueto* developers and the insight we gained, the developers did not regard it as an important feature. Due to that their decision was to hide it from the API documentation and not provide any example showcasing the use of it. Since we set hypothesis H3 to hold, our algorithm removed the corresponding API and did not consider it further. As a reminder, there exists a listener class for each handler, e.g., `MinuetoKeyListener` for relaying *Swing* key events to the corresponding `MinuetoKeyboardHandler`. However, if hypothesis H3 is set to not hold, the listener classes are placed as a child under their respective handler classes, e.g., a feature providing `MinuetoKeyListener` is a child of the feature for `MinuetoKeyboardHandler`. We designed M_I such that there are feature interactions between *Capture Swing Events* and the individual handlers. For example, choosing *Cap-*

ture Swing Events and *Keyboard*, would provide `MinuetoKeyboardHandler` but also the `MinuetoKeyListener` to allow the user to capture key events in *Swing* and handle them in *Minueto*. As we discussed in Section 6.1, our algorithm currently does not detect feature interactions. However, the result produced by our algorithm accurately depicts the dependencies of listeners.

Summary Both feature models have 18 features in common (out of 27 features in M_I and 24 features in M_{ac}). The main features were identified by the algorithm. Some of the features—which provide one operation or group a few operations—were not identified. This is due to the usage of these operations in examples. For instance, *Options and Tool Operations* in M_I provides operations that are separated into three separate features in M_I (*OS Detection*, *Alpha Transparency* and *Hardware Acceleration*). Out of the three important groupings (window types, graphical elements and handlers) all three were detected by the algorithm. As mentioned above, the examples are slightly flawed in the way they make use of the API. This brings to light an additional benefit of automated concernification. A designer can use the result of the automated concernification algorithm to design a coherent and comprehensive set of examples that showcase the use of a framework's API. For example, the designer would realize that there is no example showcasing just the use of a fullscreen window without the use of a frame. As we discussed above, providing such an additional example would correctly result in the two features being detected as siblings.

Due to operations being pulled out when their usage differs from their containing class, the algorithm in general produces a feature model with a very fine level of granularity. However, the number of features might turn out to be unreasonably large. Using the option to reduce the size conforms well with our version of the feature model. In M_I we chose to not provide a fine level of granularity. The Minueto developers chose a finer granularity in certain cases (see Section 4.4). For example, they chose to provide a finer level of granularity for three (out of 5) image manipulation operations (`crop`, `scale`, and `rotate`), whereas in M_{ac} the image manipulation operations were merged back into the parent feature *Image*. In addition, the second *Minueto* developer chose a fine granularity level for drawing inside an image (`drawCircle`, `drawPolygon`, and `drawRectangle` of `MinuetoImage`). This shows that a designer might even want to vary the granularity depending on the feature group. The intention of our automated concernification algorithm is that the designer can make adjustments to the feature model after the algorithm produced the initial version.

6.3 Qualitative Study with Android Notifications

The *Automated Concernification* algorithm was designed incrementally taking into account the design of the *Minueto* framework and the *Workflow* concern. To validate whether the algorithm produces accurate results for other frameworks, we performed a user study using a third framework. For practical relevance, we wanted to perform the study on a widely used framework which is actively and continuously developed and maintained. Furthermore, the framework should contain potentially many features and have examples that showcase the use of them.

The framework we chose for the study is the Android SDK [6] that allows app developers to create mobile apps for *Android* devices (phones, tablets, watches, and car infotainment systems). The Android SDK was first released in October 2009 along with Android 2.0 and has since been updated frequently along with the *Android* platform. We use version 8.0 which was released in August 2017. Within the SDK, *Android* distinguishes by API level to identify different revisions of the API. Version 2.0 corresponds to API level 5, whereas version 8.0 corresponds to API level 26. The core of the Android SDK provides 2235 public classes, 449 interfaces and 21773 public methods. Due to the large size of the Android SDK our study focusses on one specific part of the Android SDK, namely *notifications*. Several examples are provided showcasing different features and uses of notifications. In general, notifications provide an app developer the ability to show a "*message that Android displays outside [an] app's UI to provide the user with reminders, communication from other people, or other timely information from [the] app. Users can tap the notification to open [the] app or take an action directly from the notification*" [11].

To build apps that support many API versions, *Android* provides a support library which supports newer features for older versions by "*gracefully fall[ing] back to equivalent functionality*" [12]. This relieves app developers from handling different API versions on their own. The support library's minimum required API level is 14 (Android 4.0). The support library defines 396 classes, 142 interfaces, and 3896 methods. For example, to support push notifications across all those versions, the support library provides an additional class `NotificationCompat` to be used instead of the `Notification` class from the core SDK. Depending on which API version a method is executed on, it will be handled differently to support features that were added later.

6.3.1 Study Preparation

This section describes the steps taken to prepare the study. First, we discuss which examples were selected. We then describe the focus of the study on backward compatible notifications. Finally, we describe how *automated concernification* was performed and show the resulting feature model.

6.3.1.1 Selection of Examples

Android provides guides for developers on different topics besides examples. The official IDE for Android app development—Android Studio—provides the option to import code samples that showcase the use of the API. The examples can be run in a simulator or on a real device. The samples are maintained in an Android specific repository[4], and mirrored to GitHub[5].

For notifications, the following samples are provided:

- ActiveNotifications,
- BasicNotifications,
- CustomNotifications,
- LNotifications,
- MessagingService,
- NotificationChannels,
- Notifications,
- SynchronizedNotifications,
- WearNotifications, and
- XYZTouristAttractions.

We excluded *SynchronizedNotifications* from our study because it was deprecated in February 2017.

6.3.1.2 Focus on Backward Compatible Notifications

In total, the API related to *notification*s comprises 65 classes and 11 interfaces, which is a significant number. The classes include those for *core notifications* (class `Notification`) supporting the latest features and *backward compatible notifications* (class `NotificationCompat`). In fact, they share the same structure of classes and in large the same API. The main differences are the classes `NotificationChannel` and `NotificationChannelGroup` as well as the respective methods in the `NotificationManager` class for managing them. These classes are only available in the *core notifications* part. However, with API level 26, it became mandatory to provide a channel to which notifications are assigned to.

To avoid overloading the participants with a feature model that is very large and contains duplication, our study focusses on the *backward compatible notifications*. This is supported by the fact

[4]Repository *platform/developers/samples/android* hosted on https://android-review.googlesource.com/

[5]https://github.com/googlesamples/

that most samples make use of compatible notifications. Furthermore, due to the high fragmentation of Android platform versions, it is generally recommended to ensure compatibility across as many versions as possible[6] by making use of `NotificationCompat` and `NotificationManagerCompat` from the support library. This ensures that new features can be added while providing compatibility for older versions. The user does not have to write conditional code since the API handles it. Our decision is further confirmed by the fact that the official developers guide [8] makes use of the `NotificationCompat` class.

However, there are three samples that either just use *core notifications* or mix the use of both. For consistency, we replaced the use of *"Notification."* with "*NotificationCompat.*" in the source code of the samples *LNotifications*, *NotificationChannels* and *Notifications*. Furthermore, `NotificationManager` was replaced with `NotificationManagerCompat` unless, as mentioned above, the methods of `NotificationManager` were used for managing notification channels or groups.

6.3.1.3 Performing Automated Concernification

We performed automated concernification on the Android SDK with the notification samples. The hypotheses that hold are H1, H2 and H4 (see Section 5.4). Due to the large size of the API, not all code elements are used in the examples, only the most important ones, and as such, H3 does not hold. In order to reduce the feature model size for the participants we enabled the hypothesis nevertheless, thus focussing on the important API elements actually used in the examples only. The interface allows a participant to report a feature that is missing providing the information of the feature, such as its name, parent, and relationship. For the same reasoning, we additionally enabled the option to minimize the size of the resulting feature model, which moves operations back to their class in case they were not merged with any other elements.

Determining the names for features is out of the scope of this thesis. However, to make it easier for participants, providing a useful name for features is helpful. Therefore, we manually named the resulting features based on the API they contain. In most cases the name resembles the containing class.

Figure 6.5 shows the automatically determined feature model of *Android Notifications*. There are 15 features and four constraints. The feature *Use Newer Notification Manager Features* is highlighted in grey because it does not have any realization model and is therefore just a grouping feature. The other two groups are abstract super-classes and therefore their children are in an OR-relationship.

[6]See https://developer.android.com/guide/topics/ui/notifiers/notifications.html#compatibility

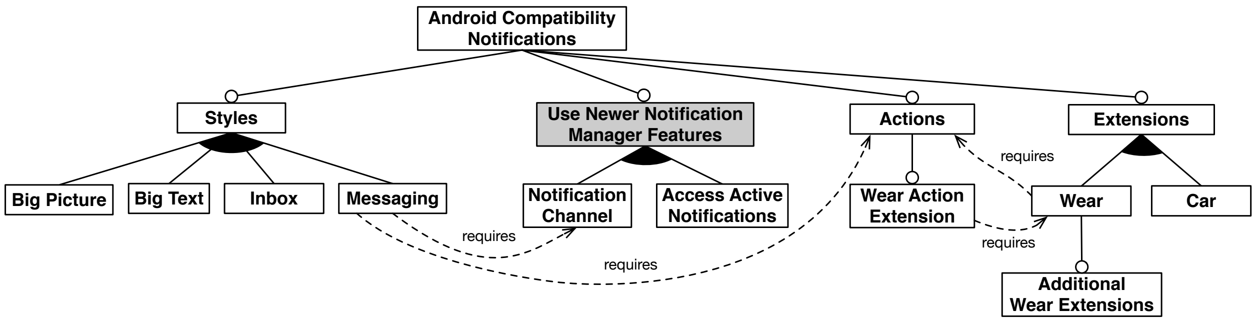


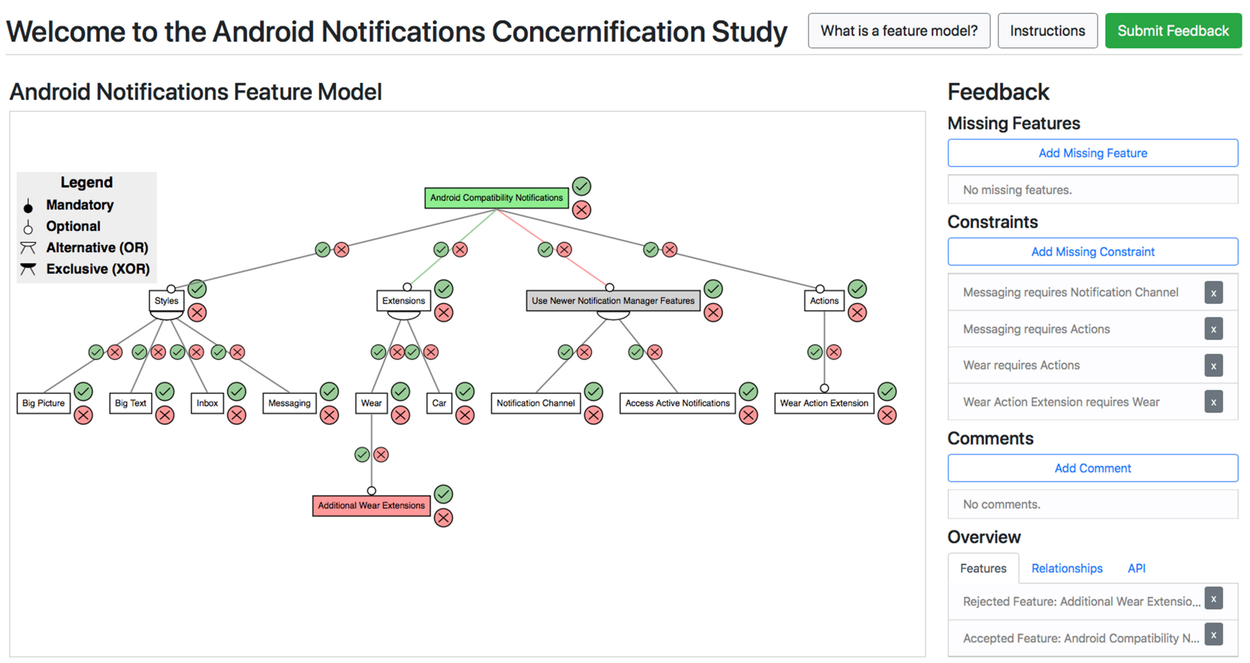
Figure 6.5: *Android Notifications* Feature Model determined by *Automated Concernification*

6.3.2 Study Task

The task of the participants is to provide feedback on the accuracy of this automatically determined concern interface for *Android Notifications*. We designed a convenient to use, interactive website for the participants to input their feedback. The website shows the feature model and allows a participant to view the API for each feature. Additionally, the cross-tree constraints are listed. Each element of the feature model (i.e., features and their relationships) can be accepted—i.e., the participant agrees that it is in fact a user-perceivable feature—or rejected. Rejecting an element requires the participant to provide an explanation which describes the rationale or provides an alternative. For example, a feature could be rejected because the participant disagrees with the name describing the functionality the feature provides. Similarly, for relationships, it is possible that the relationship type or parent is deemed incorrect, and as such, the participant needs to indicate the correct one according to his view.

In addition, missing features can be proposed by a participant using a suggestion that includes the name of the feature, the parent feature, and the proposed relationship type. Furthermore, feedback on the determined cross-tree constraints can be provided. Figure 6.6 shows a screenshot of the study website displaying the feature model of the study subject. The participant feedback is visualized directly within the feature model for features and relationships with green and red colour highlighting (as can be seen in Figure 6.6). In addition, the feedback is also listed in textual form in the sidebar on the right at the bottom.

In terms of the API, due to the large size, if no feedback for an API element is given, it is considered to be accepted. Participants can reject any element of the API and provide a reasoning for it, e.g., for misplaced API elements the participant can specify the feature that he thinks the element belongs to instead. Figure 6.7 shows a screenshot of the study website when viewing the API provided by a feature. In this case, the API for the feature *Notification Channel* is shown.



No missing features.

Messaging requires Notification Channel

Messaging requires Actions

Wear requires Actions

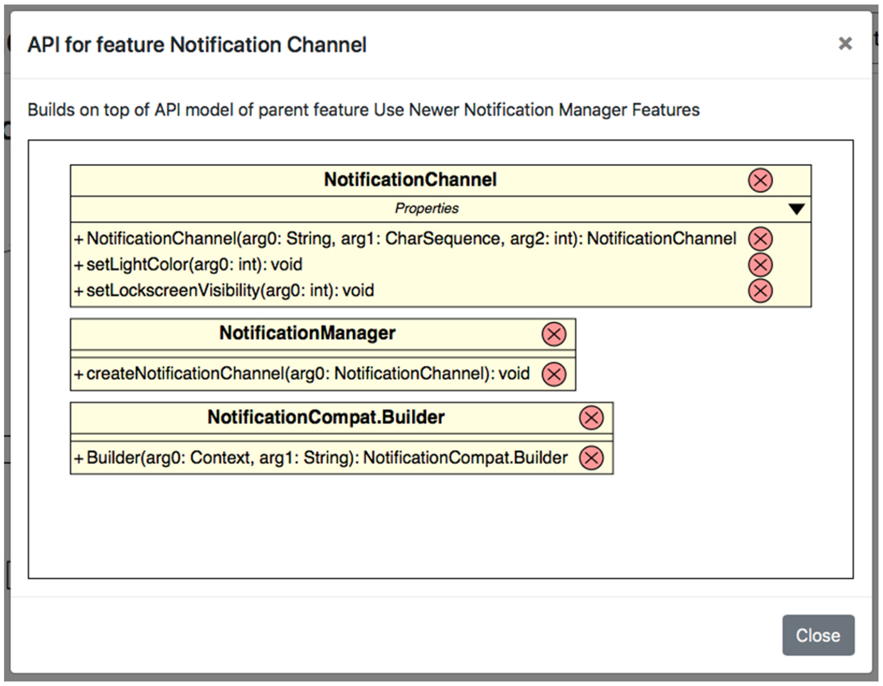
Wear Action Extension requires Wear

No comments.

Rejected Feature: Additional Wear Extensio...

Accepted Feature: Android Compatibility N...

Figure 6.6: Automated Concernification Study Website



Builds on top of API model of parent feature Use Newer Notification Manager Features

NotificationChannel

Properties

+NotificationChannel(arg0: String, arg1: CharSequence, arg2: int): NotificationChannel
+setLightColor(arg0: int): void
+setLockscreenVisibility(arg0: int): void

NotificationManager

+createNotificationChannel(arg0: NotificationChannel): void

NotificationCompat.Builder

+Builder(arg0: Context, arg1: String): NotificationCompat.Builder

Figure 6.7: Viewing the API of a Feature on the Automated Concernification Study Website

6.3.3 Study Participants

An interesting insight from the Minueto study (see Chapter 4) we gained was that it is possible that the internal knowledge of how a framework is designed and implemented can bias the view when looking at it from the user's perspective. Users of an API are usually not aware of the internal details and mainly focus on what can be accomplished with an API and how it can be accomplished. We therefore recruited users (i.e., app developers) of the Android SDK. We required participants to have previously used the *Android Notifications API*. We recruited four participants (P1 to P4). Two were recruited through an online community for Android developers, one through a *Google Developers Community Group* (GDG) for Android Developers, and one through McGill's School of Computer Science student body. All of the participants had at least three years of experience with Android app development, and three of the participants had professional Android app development experience.

6.3.4 Data Analysis and Results

Using the feedback provided by the participants, we perform a qualitative analysis of the data and discuss it in this section. To verify the validity of the feedback, we also consider the official (textual) notifications guide [11] providing an overview on notifications, their different features and instructions on how to create them.

6.3.4.1 Features and their Relationships

Three participants (P1, P2, and P3) agreed with all the features. P4 rejected the feature *Additional Wear Extensions* commenting that "*I'm not sure why the content of Additional Wear Extensions is not part of the parent Wear[eableExtender*]". Consequently, the relationship to the parent was also rejected by P4 with the reasoning that "*I think everything in the Additional Wear Extensions could be part of the parent*". This optional child feature provides four methods of `WearableExtender`. Furthermore, it contains one method of `NotificationManager-Compat` and three methods of `NotificationCompat.Builder`. They were all used together in one example. However, based on their names they do not seem to be related and none of them are mentioned explicitly by the official user guide [9].

P3 commented that "*all notifications need a title and a small icon at the very least*". According to the documentation [11] the only requirement for a notification is a small icon. However, in [8] the documentation provides an example with a small icon, content title and text, and priority. In essence, while it is possible to create a notification with only a small icon it is not very useful to the user. The feature model produced by our algorithm contains the required methods in the root feature, plus additional methods. The designer could add a usage protocol that requires to call the required methods before `build` can be called.

In terms of relationships, two of the participants (P1 and P3) mentioned that the relationships of the features under *Style* should be in an exclusive relationship (XOR) because "*only one style should be applied to any single construction of a notification*". *Style* provides an abstract superclass `Style` and the children provide concrete sub-classes (e.g., `BigPictureStyle`). However, the feature model does not specify run-time constraints but rather provides the API at design-time. A user might use different kinds of notifications in an application that require different styles and then create several instances of notifications with a different kind of style for each notification. This is reflected with the OR-relationship.

P1 also provided a comment relating to the *Extensions* feature group: "*Overall I agree with the grouping of the features, the "Extensions" [group] is one that is rather tricky, since, for example, "Wear Action Extensions" can fall under either "Extensions" > "Wear" or under "Actions" > "Wear"*". *Wear* under *Extensions* provides the main extensions for watch-specific features whereas *Wear Action Extensions* adds the ability to extend actions with watch-specific features. While both are only used for watch apps, there is no indication in the documentation [9] stating that both need to be used together. `NotificationCompat.WearableExtender` can be used to specify special settings for the watch app, and `NotificationCompat.Action.Wearable-Extender` allows a developer to specify watch-specific settings for actions. If they were always used together, it would indicate a feature interaction between *Wear* and *Actions.* P1 ended the comment remarking that "*[i]ntuitively, the approach in this model seems to be the correct choice, but I'm unable to place exactly what makes it so*". The *Extensions* group was formed due to the inheritance relationships between the `NotificationCompat.Extender` interface and the implementing classes `NotificationCompat.WearableExtender` (in the *Wear* feature) and `NotificationCompat.CarExtender` (in the *Car* feature). The *Wear Actions Extensions* feature was placed under *Actions* due to the method `extend(Action.Extender)` being pulled out of its class `NotificationCompat.Action.Builder` (which is provided by *Actions*). The `NotificationCompat.Action.WearableExtender` class was merged with this method due to the same example usage.

6.3.4.2 Missing Features

P2 stated that `RemoteInput` could be added "*as an optional child of Actions*". We excluded this class because even though there are methods that use it, it is not solely related to notifications. `RemoteInput` provides the ability to collect input from the user, for example, to accomplish reply actions within notifications. If we do not disregard this class, the algorithm in fact produces a sub-feature of *Actions* which contains the `RemoteInput` class along with the `addRemoteInput` method of `NotificationCompat.Action.Builder`. In addition, a *requires* constraint to the *RemoteInput* feature is added from the *Car* feature due to a method (called `setReplyAction`) that is provided taking as a parameter a `RemoteInput` instance. A further investigation into the complete API of Android revealed that besides notifications the `RemoteInput` class is only used in code that is hidden from the user[7]. This explains why the designers of the Android SDK did not group the `RemoteInput` class with the notifications API.

P3 remarked that "*sound, vibration, and light don't seem to be covered*". Only three of the related methods were used in individual examples, the examples do not use most of the methods relating to these features and were therefore removed. The participant added that "*[t]he old implementation (before channels existed) should be under Android Compatibility Notifications, whereas the newer sdk's have it under [notification] channel*". Up until API 25, sound, light, vibration, and also priority had to be set per notification as methods on the `NotificationCompat.Builder`. With the introduction of notification channels in API 26 these were added to the channel. This also allows an app user to modify the settings because the notification channel settings are exposed within the system settings. The participant further added that "*using the new features would mean that some of the older features shouldn't be used*". However, maintaining compatibility across different API levels means that for those platforms that do not have notification channels, these settings need to be still set using the methods on the builder instance. The documentation on notification channels explicitly states this fact, for instance, when referring to the importance of notification channels [10]:

> *To support devices running Android 7.1 (API level 25) or lower, you must also call `setPriority()` for each notification, using a priority constant from the `NotificationCompat` class.*

In the *core notifications* API that does not provide compatibility and only supports the newest API, the corresponding methods on the `Notification.Builder` are marked as deprecated and the documentation refers to the corresponding methods in the `NotificationChannel` class. A usage protocol could ensure that both methods are used as required for compatibility if the user selects *Notification Channel* and intends to use any of those methods.

6.3.4.3 Constraints

P1 rejected three constraints. P1 only accepted the constraint *Wear Action Extension* $\dashrightarrow$ *Wear*. The three rejected constraints are caused by structural dependencies where the type that is referenced is not part of the API allocated to one of the ancestor features, but associated with a feature located

[7]Either through the non-public visibility or the Android-specific pseudo-annotation `@hide` to hide public API from the user.

in a different branch of the feature model tree. In all cases one or more methods were pulled out and merged due to matching example usage.

- *Messaging* ⇢ *Notification Channel/Actions*: The *Messaging* feature provides two methods of `NotificationChannel` and one of `NotificationCompat.Action.Builder`. However, the participant stated that they belong to the corresponding feature of the class. In the case of the latter, P1 stated that "*while the `setAllowGeneratedReplies` method inside `NotificationCompat.Action.Builder` is something that would commonly be used with the Messaging style notification, it's not required, whereas it's something that always is used with a notification action*". However, while it is related to actions, in the examples it is always used together with the messaging style. The method allows enabling the automated generation of possible choices for replying based on the context.

- *Wear* ⇢ *Actions*: This constraint is caused by the method `addAction(Action)` of `WearableExtender` in *Wear*. P1 suggested to "*move the `addAction` method to be under Wear Action Extension*". This method allows the developer to add actions that should only appear on a watch app. As such, an alternative could also be to put this method into a feature interaction model for *Wear* and *Actions,* in which case this method is only included in the API when a user selects both features.

P1 only agreed with the cross-tree constraint *Wear Action Extension* ⇢ *Wear* with the reasoning that "*Wear actions should only be used if the type of notification is a Wear notification*". As we discussed above, the two features can be used independently. Alternatively, it would be possible to restructure the feature model such that *Wear* is a logical grouping with two children in an OR-relationship providing `NotificationCompat.WearableExtender` and `NotificationCompat.Action.WearableExtender`. P4 removed all cross-tree constraints but did not provide an explanation as to why. This confirms most of the feedback of P1 since the only way the integrity of the API can be maintained with the removal of the constraints is to move the corresponding methods to the target of the constraint.

6.3.4.4 API

P3 rejected the method `addPerson(String)` of the `NotificationCompat.Builder` class provided by the root feature. The participant stated that the method is deprecated and provided a reference to the API reference, however, explicitly referring to the *core notifications*, i.e., `Notification.Builder`. In API level 28, the method was replaced with `addPerson(Person)`. As we described above, we excluded the *core notifications*. In the *backwards compatible notifications*, this method is not deprecated.

Further to the rejection of the `addPerson` method, P3 suggested that the method would not necessarily fit into the root feature because it is only specific to "*apps relating to actual contacts (which can be whitelisted from silent mode)*". Our algorithm did move it out of its class, but due to the size minimization it was moved back. Depending on the desired granularity of the developer, this could be shown as a sub-feature.

This signifies the divergence between the *core* and *backward compatible notification* APIs. In order to maintain compatibility to older API levels, the existing method cannot be replaced. However, the designers of the API chose—for reasons unknown—not to add the new method to `NotificationCompat.Builder` and providing a fallback for older versions. On the contrary, when introducing notification channels, the constructor `Builder(Context)` of class `NotificationCompat.Builder` was marked deprecated and replaced with a new constructor taking a notification channel ID as a second argument (`Builder(Context, String)`). As a fallback for older devices, the channel ID is ignored.

6.3.4.5 Analysis Summary

The feedback of the participants shows that they in large agree with the produced concern interface of the *Android Compatibility Notifications* API. In one case we disregarded a class (`Remote-Input`) that seemed relevant to the Android SDK as a whole although it is only used for notifications from the user's perspective (app developers).

The quality of examples plays an important role in our algorithm. Some parts of the algorithm are fragile to the quality of the examples. One of them are the cross-tree constraints. The cross-tree constraints that were rejected by some participants showcase this. However, our algorithm ensures that the integrity of the API is maintained. As we discussed earlier, this can serve as feedback to the developer to improve the examples. Additionally, our algorithm provides an initial concern interface which the developer can fine-tune after.

If hypothesis H4 (see Section 5.4) is set to not hold, the methods causing the cross-tree constraints would remain with their class (as explicitly suggested by one participant). However, then other cross-reference dependencies would cause constraints that would otherwise be merged into the same feature because they always need to be used together.

We consulted the official documentation for notifications [8, 10, 11]—including the guides for watch [9] and car [7] apps with respect to notifications—to crosscheck whether there are any potential features that participants did not explicitly mention as missing features. We identified three features. For example, there are two additional styles that can be used, one for media controls and track information (`MediaStyle`) and one for providing a custom layout for notifications (`DecoratedCustomViewStyle`). When unused API elements are not discarded by the algorithm (by specifying hypothesis H3 to not hold), the two classes are detected as separate features and grouped as siblings to the other styles under the *Style* parent feature. Another feature is `NotificationChannelGroup` which allows grouping notification channels, e.g., for apps supporting multiple accounts that provide the same notification channels per account. In this case, due to methods in `NotificationManager` cross-referencing `NotificationChannelGroup` that are never used, the resulting concern interface provides it as a feature. However, `NotificationChannel` (along with `NotificationManager`) is placed as a child under `NotificationChannelGroup` due to the *crossref* edge from the former to the latter. This also reveals a flaw of our implementation (see Section 5.6) where we didn't consider generic types for return or parameter types of methods. Addressing this results in an additional constraint between the parent (*Notification Channel Group*) and the child (*Notification Channel*). Based on the provided API it makes sense, since `NotificationChannelGroup` alone can not be registered without the appropriate method from `NotificationManager`.

However, these features are not covered in the examples we used as an input to the notification examples. The developer can in the end add those features when fine-tuning the concern interface. As well, the developer is reminded that the examples need to be improved because they do not showcase these features.

6.3.5 Study Limitations and Threats to Validity

The *Automated Concernification* algorithm does not provide proper names for the features that it determines in the sense that the name concisely reflects what it provides. In order to be able to show the participants a feature model, we determined names for the features manually based on the API they provide. This can influence the participants. To minimize the *researcher bias*, where possible we named the features based on the classes they contain. The participants were required to be familiar with and have used the notifications API of Android before. All participants had at least three years of experience developing Android apps. However, the specific knowledge of the individual participants can influence results. We triangulated the responses of each participant with the official documentation (user guides and API reference) to verify. In some cases, we manually ran sample code in the simulator to verify effects of API calls. For example, the fact that the participants did not miss specific styles of notifications could be because they have never used them. Using the official documentation mitigated this *participant bias*.

In terms of *transferability,* in addition to the two small frameworks we concernified, we chose a large framework that is widely used in industry. Android has been continuously developed for several years and it evolves at a fast pace. With this fast pace comes the challenge of maintaining compatibility across older devices that smartphone users still use. While we restricted the study to notifications, this part of the API has a decent size, consists of different features and is fairly complex when going beyond a simple notification. This API can be seen as representative of the framework as a whole. However, it is possible that performing automated concernification on a fourth framework could expose something we missed. For example, the *Spring* framework makes heavy use of annotations or XML configurations to configure the framework. Our algorithm currently does not take this into account.

6.4 Summary

This chapter evaluated the accuracy of the automated concernification algorithm. We evaluated three frameworks. The *Workflow* concern is a concern that was originally conceived as a concern from which code was generated. We therefore know the features and their API in advance, which enabled us to evaluate the accuracy in detail. The second framework is Minueto where we validated our feature model in a qualitative study with the developers in Chapter 4. This allowed us to compare the validated feature model with the automatically concernified one. To ensure that automated concernification works for large frameworks used widely in industry, we performed automated concernification on *Android Notifications*. We conducted a user study with app developers to validate the produced concern interface.

Overall, the results show that a high number of features are discovered by the concernification algorithm. In the case of *Workflow*, all features were found. For *Minueto* and *Android Notifications*, even though the examples have some flaws, the main features were detected. In terms of relationships, the use of structural dependencies and simplification of the graph based on the relationship types provide very good results.

The validation in this chapter confirmed one insight from the study with Minueto developers (see Chapter 4). The desired granularity of the feature model depends on the intention of the developer and can vary among different feature groups. For example, the image transformation operations in Minueto were identified as individual features by both Minueto developers. Our algorithm identified them but moved them back into the parent to minimize the overall number of features. The validation revealed an additional insight. In the case of *Workflow* we implemented the examples such that all the API is used and there is an example showcasing each feature individually. This produced a concern interface that is almost 100% accurate. In the case of "real world" frameworks like *Minueto* and *Android Notifications*, the examples are not flawless. The performance of our algorithm is affected by the usage of the API in the examples. For example, our algorithm has trouble modularizing API elements that are never used in any examples. In *Minueto*, one feature (capturing *Swing* events) was missed, whereas in *Android Notifications* three were missed. However, all four cases seem to pertain to features that are rarely used by users of the frameworks.

In any case, the developer can in the end make adjustments to the concern interface result. A developer can do this with minimal effort provided she has access to a tool that allows her to make adjustments to the feature model or move API elements between features in a quick and intuitive way.

7 Related Work

7.1 Concernification

To the best of our knowledge, there exists no other approach that addresses all the benefits of concernification as described in Chapter 3. The most similar related work are the *design fragments* approach proposed by Fairbanks *et al.* [40] and framework-specific modeling languages (FSML) by Antkiewicz [13].

7.1.1 Design Fragments

A design fragment [40] encodes a design that uses a framework for a specific purpose. A design fragment specifies parts of the framework, i.e., the parts of the framework API that are relevant to achieve the goal of the design fragment. Furthermore, it declares the code elements the user needs to provide. and in what relationship they are with the framework provided elements, i.e., whether they must sub-class a class, implement an interface, or override a method. The described API and the user code elements can consist of classes and interfaces, methods, and fields. A design fragment also includes a behavioural specification that specifies that the user is required to create new instances and invoke methods. In addition, support for XML configurations, which are used by many frameworks, is provided [39]. Furthermore, a design fragment can be accompanied with free-form text for documentation reasons. The approach recommends a common text template to describe more detail, e.g., for each method, how often it is invoked, whether it is a callback method, etc. A catalog of design fragments can provide a list of "*conventional solutions to problems*" for a framework. Within the catalog, design fragments can be categorized into *stable*, *unstable*, and *testing* categories. While design fragments provide a way to mark a design fragment as *deprecated*, there is no way to refer to the design fragment that should be used instead.

The design fragment language is defined using an XML schema definition (XSD). This allows a tool to read a design fragment specification and use it to verify that the user correctly applied the fragment within the client code. To specify that a design fragment is being used, the user needs to add annotations to the source code elements that correspond to the ones prescribed by the user-provided elements of the design fragment. The names of those elements are referred to as roles in the annotation. The authors describe a tool that is provided as an Eclipse plugin that allows a user to view a catalog and the instances of design fragments, and the tool provides feedback on any errors that are found during analysis of the code to ensure the design fragment has been correctly applied. However, there does not seem to be an editor to create design fragments, requiring to write the XML manually according to the schema. Furthermore, the tool is not available for download.

In general, design fragments and concernification provide a partial specification of the subset of the framework API and user-relevant code (glue code). There are however many differences between our concernification approach and design fragments. First, design fragments provide a loose collection of example uses of a framework, i.e., there exist no relationships between them (groupings, constraints, etc.). In contrast, our framework concern interface provides all user-relevant framework features via the variation interface, and the inter-relationships of the features are clearly expressed in the feature model. Furthermore, design fragments do not provide any insight on the impacts that the use of a fragment has on high-level goals. Our customization interface and the mappings that the user needs to establish to his application model elements, though, is similar to the bindings that the design fragments user needs to provide. However, while in our case mappings are defined in one place during the reuse process, with the design fragments approach the bindings that take the form of annotations have to manually be placed on the source code elements that are potentially scattered over multiple modules. Third, while design fragments distinguish between callback methods (with a description of when and how often they are called) and service methods (framework-provided methods), we distinguish between methods that can be called by the user and those that should be customized. Service methods are part of the usage interface, and all callback methods are part of the customization interface, however, the design fragments language definition only provides a way to textually describe those methods that need to be implemented/overridden as documentation information to the user. Additionally, there could be other methods in the customization interface that are neither callback nor service methods, such as methods that are advised with additional behaviour. Finally, in our approach, the structure and behaviour a user always has to use (glue code) can be encapsulated within the concern and applied through mappings to the user's design, whereas with design fragments the user needs to manually add glue code to his program. A design fragment provides a behaviour specification to ensure that a new instance is created or certain methods invoked. Other types of user-provided behaviour can only be textually described. Our concern interface allows the inclusion of partial behaviour using sequence diagrams. Furthermore, the usage protocol of API methods can not be encapsulated in design fragments. With concern interfaces we can specify usage protocols.

7.1.2 Framework-Specific Modeling Languages

Another approach that has very similar goals to ours is proposed by Antkiewicz [13], which introduces framework-specific modeling languages (FSML) as domain-specific modelling languages (DSL) for a specific framework. A FSML is designed for a specific framework to express a framework-based application to help application builders develop an application with the framework. It formalizes the API concepts, their features, and constraints. The concepts are usually classes and the features specific parts of the class that need to be provided. FSML makes use of cardinality-based feature models [24] allowing features to have a multiplicity. Features are very fine-grained and code-centric and encode required user-provided code, such as the name for an instance or extending a certain super-class. In addition, mappings can be specified to define the correspondence between features and structural and behavioural code patterns. This makes it possible to achieve round-trip engineering by providing forward mappings (how to generate code from a feature) and reverse mappings (how to recognize an instance of a feature in the client code). A framework-specific model represents a feature configuration and describes how a framework-based application uses the FSML of a framework. A generic FSML infrastructure built as an Eclipse plugin is provided.

In comparison to our concernification approach, FSMLs require the definition of a metamodel which extends a generic FSML metamodel for each framework. The effort in creating a specific DSL for each framework is very high. The upper level features represent higher level concepts, and the features nested under them represent customization and usage steps as well as constraints. For instance, a higher level concept is usually a class, and the underlying features the required and optional means to use that class in an application. Names of features describe the customization or usage steps. For example, the mandatory feature `extendsApplet` refers to the fact that the client needs to extend the `Applet` class. Furthermore, features in FSML have code mappings which enable round-trip engineering and application migration. In contrast to this code-centric view, our approach clearly specifies the functional (user-perceivable) features and thus presents a high-level view of the complete framework. The feature model of an FSML contains the variations as well as customization and usage steps. In our concern interface the details about customization and usage are handled separately in the interface. This separation of concerns allows a user to deal with customization and usage afterwards, i.e., after selecting the desired features. The required glue code can be included within the design model of a feature.

In summary, FSMLs are targeted at the code level which requires a code-centric view, whereas our approach is targeted at the modelling level. However, it needs to be investigated in the future whether and how both approaches could be combined to result in the best of both worlds. For example, the feature model could show the high-level view and instead of having a fine level of granularity in the feature model, individual features could have a realization models that include the round-trip engineering hooks found in FSML. Round-trip engineering makes it possible to ensure that model and code are synchronized. If a user modifies the code after code generation, it is important to update the models, e.g., to verify that no violations were introduced.

7.1.3 Comparison

Both approaches described in the previous sections use *Java Applets* as an example. For these, there are 20 official demos/examples that are provided with the Java Development Kit (JDK). We ran our automated concernification algorithm on Java Applets using the official demos as examples. This allows us to compare whether the automated concernification algorithm detects the design fragments/features manually determined in the two approaches.

A Java applet is a small Java program that is intended to be embedded inside another application [84]. It provides an interface between the graphical user interface of Java (AWT/Swing) and the other environment, such as a browser. In essence, it provides one class `Applet` that needs to be sub-classed. Additionally, Swing provides a sub-class of `Applet` called `JApplet` for applets that use Swing components. The official examples however do not make use of this class. Since Java 9, applets are deprecated[1] as they require a browser plugin which browser vendors are phasing out. It is debatable whether applets constitute a framework as it can instead be seen as one feature of the graphical interface part of Java. Nevertheless, it provides some variability. For instance, it has lifecycle methods that can be overridden, and applet parameters that act like command-line arguments.

Due to the nature of the `Applet` class, features from AWT/Swing can also be used with applets. However, in this comparison we focus solely on applets. Therefore, features outside this scope that are specified in the two approaches are left out, such as code snippets that illustrate how to define various listeners and how to run background tasks in separate threads. After excluding these, only two relevant design fragments remain: *Parameterized Applet* and *Manual Applet*. The framework-specific modeling language for applets however is very detailed. It has features for extending the `Applet` class, overriding various lifecycle methods, showing a status, and parameters.

The root feature that automated concernification finds contains the `Applet` class with a customized sub-class as well as the method `getAppletInfo()` which should be overridden by the user to provide information about the applet. Interestingly, both approaches do not mention this method. For the lifecycle methods (`init`, `start`, `stop`, and `destroy`) our algorithm detects three features overriding these methods: *init*, *start* and *stop*, and *destroy*. Furthermore, a feature providing the `showStatus` method to show the status of the applet is detected. Another feature

[1]https://openjdk.java.net/jeps/289

provides the `getParameter` method as well as the required `getParameterInfo` method to override in the custom sub-class. This relates to the *Parameterized Applet* design fragment. Lastly, two features are detected relating to manual invocation of an applet, one for `init`, and one for `start`. The reason they are separated (the examples mentioned by Fairbanks relating to this design fragment all call both of them) is because another example explicitly calls `start` when handling an event. Our algorithm found more features providing other methods of the `Applet` class, such as retrieving an image or playing an audio clip.

In summary, this shows that our automated concernification algorithm can automatically determine features based on the example usage that were manually determined from the examples for the two approaches.

7.2 Automated Concernification

Automated concernification is broadly related to several fields, namely feature mining, feature location, software maintenance and evolution, and program comprehension. The source code for a particular feature needs to be located. Dit *et al.* [31] provide a comprehensive overview of feature location approaches until 2011. Feature location is a common task performed by software developers during software maintenance, e.g., when adding a new feature, improving existing functionality or fixing a bug. All surveyed papers provide solutions for identifying source code for a particular feature that a developer is interested in. The approaches use three different location techniques. Dynamic analysis approaches make use of execution scenarios that are analyzed. For example, Wilde and Scully [126] describe software reconnaissance where one set of test cases does not invoke a feature and another set of test cases does. Eisenberg and Volder [37] propose an approach that uses ranking heuristics to determine the relevance of a code element for a feature.

Static analysis feature location techniques make use of the dependency graphs of the source code based on a program element or elements provided by the developer (e.g., [23,90]). This is often done in an interactive way where during the investigation the developer narrows or widens the scope. In this context, Robillard and Murphy [93] propose to use concern graphs to describe program elements and their relationships. Within the graph, the vertices represent program elements (classes, fields, and methods) and the edges between them describe six types of relationships. The relationships are more implementation-specific than ours, however, there is some commonality. For instance, *superclass* refers to our *inheritance* relationship. The *declares* relationship specifies that a class declares a method or field, which is similar to our *containment* relationship.

The third category of techniques relates to information retrieval (IR) approaches that take as input a query and use textual analysis (e.g., [68]). Some approaches combine the use of different kinds of techniques. For instance, Eisenbarth *et al.* [36] present an approach where execution scenarios invoking the features of interest are created by the domain expert. The scenarios are dynamically analyzed to provide a mapping between scenario and invoked code. Formal Concept Analysis (FCA) is used to help a developer manually identify relationships between scenarios and computational units and essentially a mapping between features and computational units. A developer interested in the code related to a feature performs FCA iteratively using different execution profiles to incrementally build a mapping between relevant code and features of interest. In combination with the dependence graph retrieved via static analysis, program elements that were not executed in the scenario but are relevant can be identified, and unrelated elements to the feature can be removed.

Our approach is more closely related to reengineering an existing system into a Software Product Line (SPL). In this area, several system variants are reengineered into an SPL. This migration can be divided into three phases as proposed by Anwikar *et al.* [14]:

- The *detection phase* deals with locating the features by extracting relevant information from artefacts, such as source code.
- The *analysis phase* uses the previously discovered information to identify commonalities and variabilities, and to organize the functional features into a feature model.
- The *transformation phase* performs transformations on the artefacts to migrate the existing system into an SPL.

Our first and second phase where the DAG is established and populated relate to the detection phase. The third phase relates to the analysis phase where the DAG is simplified to retrieve a tree that forms the feature model. Our algorithm does not transform existing artefacts. However, the API related to each feature is placed within a design realization model. The resulting concern interface allows a user to retrieve a customized interface variant of a framework based on the desired features.

In the context of reengineering existing systems into an SPL, a family of products has been created over time. However, each product is individual. The migration to an SPL is therefore a bottom-up approach and is needed to facilitate systematic reuse and improve the development by consolidating the code. It is often the case that the implementation of one product has been continued and the same features across product variants have differences in their implementation. In contrast, in our case we consider only one variant, the complete framework. Frameworks consist of the combination of all variants that are possible. I.e., it is possible that there are features that exclude each other and it is the responsibility of the user to not use them together.

Assunção *et al.* [15] provide a recent overview of the current research in reengineering applications into SPLs. Approaches from the survey by Dit *et al.* that fall into this category are also present in this study. In addition, more recent work is included. We discuss those approaches that are related to our work here.

Damaševičius *et al.* [26] propose an approach that is similar to ours in its goal, to automatically extract features and the feature model from Java code. Methods are considered as features. A dependency graph is extracted from compiled Java code. The nodes represent methods and are clustered according to the similarity of their dependencies to other methods (based on dependency information from the dependency graph). All code elements, even those that are invisible to the user, are considered. The resulting feature model is verbose and does not maintain the integrity of the API. For example, applying the approach to four buffer classes provided by Java results in a feature model with 73 features. Furthermore, cross-tree constraints and OR/XOR relationships between features are not supported by the approach.

Ziadi *et al.* [130] propose a three-step approach to identify features from the source code of product variants. The approach assumes that features are implemented consistently across the products, i.e., the names of code elements do not differ. First, the source code of a set of product variants is analyzed by abstracting the structural elements of the code, i.e., packages, classes, attributes and methods. The elements are represented as construction primitives and are considered individually, i.e., each element might belong to a different feature. In the following step, feature candidates are automatically identified. A feature candidate is a set of interdependent construction primitives. The authors define interdependence between two construction primitives iff they belong to exactly the same products. In those cases, they belong to the same feature candidate. In our algorithm, we identify code elements as belonging to the same feature candidate if they are used in the same examples. The third step is manual and allows removing non-relevant features or adding any missed features. The organization of the features into a feature model is not covered by this approach.

AL-msie'deen *et al.* [3] consider variability expressed through packages and classes in software variants. This means that the structural elements of a class, attributes and methods, always belong to the feature the class is in. In their experience from case studies, features are implemented at class or package level. They distinguish code elements in source code as those that are common (called *common block*) among all variants, i.e., mandatory, and those that appear only in some variants (called *block of variations*), i.e., optional. Their mining process first identifies the code elements that are common and the different sets of code elements that vary across the product variants. Formal Concept Analysis (FCA) is used to identify those. In a subsequent step, the identified concepts (sets of code elements) are further separated to ensure that they implement only one feature. Using a combination of structural similarity based on dependencies between the code elements and lexical similarity determined using Latent Semantic Indexing (LSI) the similarity of code elements is determined. This is determined in binary form (similar or not) and FCA is used again to accomplish the separation of concepts. Like in Ziadi's work, the feature model is not determined. However, this approach determines potentially several mandatory features and considers the possibility that code elements do not belong to the same feature even though they are present in the same product variants.

Xue *et al.* [129] propose an approach that also makes use of FCA and LSI. They combine it with software differencing to determine code elements that appear in all variants, and those that appear only in subsets of variants. However, the set of features and their textual descriptions as well as the information of which product variant contains which features needs to be provided as input. FCA is used to find sets of features and code elements that are common and different among product variants. LSI is used to determine which set of code element implements a feature.

Maâzoun *et al.* [67] propose a UML profile that augments feature information as stereotypes to a design class diagram. Besides the feature name this includes the relationship types. Optional and mandatory information for classes and their elements, and optional, mandatory and XOR relationship information between classes is also provided. In addition, the UML profile provides the ability to specify OCL constraints to ensure consistency. They describe an approach that aids the extraction of the feature model and class diagram augmented with the stereotypes of the UML profile from product variants with the help of FCA and LSI. Similar to the approaches by Ziadi and AL-msie'deen, commonalities and variations are determined.

Some of the discussed approaches make use of Formal Concept Analysis (FCA). FCA is a mathematical technique for analyzing binary relations [127]. It allows one to derive implicit relationships between objects described through a set of attributes. A formal context is a relation table that describes relations between objects and attributes. Performing FCA produces a concept lattice, i.e., a line diagram, which depicts the natural hierarchical order between concepts of a context (called the *subconcept-superconcept* relation). We could describe the relation between API elements and their use in examples and then perform FCA to determine a concept lattice depicting the relationships between API elements usages. However, we then loose the dependencies between API elements inherent in the code which are essential to maintain API integrity. Without code dependency information it is also difficult to determine the placement of unused API elements.

Martinez *et al.* [69] identify the challenge of practical adoption due to the lack of end-to-end-support for bottom-up SPL creation, as is the goal of the works discussed before. To overcome this challenge, they propose principles for a unifying framework and present a framework for bottom-up SPL adoption. The main idea is to provide a framework that is independent of concrete types of artefacts and algorithms to support all kinds of artefacts and algorithms. One of their goals through this is to allow comparison of different approaches. They present an implementation of such a framework called *Bottom-Up Technologies for Reuse* (BUT4REUSE). It is intended to be generic and extensible.

Further to the SPL migration approaches, another approach is explained in [77] where large C programs, such as the Linux kernel, are mined for constraints between features. The features are already known, because they are part of the build configuration (*Kconfig*). The approach is specific to C programs due to the use of preprocessor statements such as `#ifdef` which contain the feature name. The resulting configuration constraints can then be used to reverse engineer the feature model.

Such an approach is described by She *et al.* [106]. They describe procedures on how to reverse engineer feature models from a set of known features, their descriptions and their dependencies described as logic formulas. They describe a procedure to identify potential parents based on a ranking heuristic as well as the identification of feature groups. The user first has to choose from the parent candidates and in a subsequent step choose which feature groups to keep. This is a semi-automated approach, whereas our approach is completely automated. We could formulate a heuristic based on information in the DAG (e.g., relationship types, names of elements, etc.) to help in deciding which edge to cut in the simplification phase.

The idea of creating feature models that satisfy a set of constraints using logic formulas was originally proposed by Czarnecki and Wasowski [25]. In our approach we could, in theory, map our class hierarchy and parameter dependencies into constraints, generate a logic formula, and use the approach by She *et al.* to determine a corresponding feature model. However, one disadvantage in using logic formulas is that the information about the object-oriented hierarchy and the different relationship types are lost.

Other approaches make use of natural language processing (NLP) on the textual documentation to determine features or tasks. One such approach is TaskNav [119], which extracts potential developer's tasks from the framework documentation. A task is a specific programming action described in the documentation. Code elements mentioned in the textual documentation are related to tasks. Different variations of a task may be found and a code element may be related to several tasks, while in our case a code element is related to one particular feature. In addition, we are interested in user-relevant (high-level) features, whereas tasks might be low-level. Nevertheless, it might be interesting to combine NLP or information retrieval approaches for finding feature names in the future. In addition, it could be helpful in deciding which edge to cut in the simplification step of our algorithm, or for determining groupings of unused elements.

Part III

Signature Extension: Fine-Grained Abstraction

8 The Need for Flexible Operation Signatures

Reuse is central to improving the software development process, increasing software quality and decreasing time-to-market. Hence it is of paramount importance that modelling languages provide features that enable the specification and modularization of reusable artefacts, as well as their subsequent reuse. This chapter begins by introducing interfaces in the context of reuse in Section 8.1 and motivates the need for flexible signatures in Section 8.2. We then outline several difficulties caused by the finality of method signatures that make it hard to specify and use reusable artefacts encapsulating several variants (Section 8.3). The difficulties are illustrated with a running example. To evaluate whether these difficulties can be observed at the programming level, we report on an empirical study conducted on the Java Platform API as well as present workarounds used in various programming languages to deal with the rigid nature of signatures (Section 8.4). Section 8.5 concludes this chapter.

8.1 Introduction

Complex systems are rarely built from scratch. To improve productivity and achieve higher quality during software development, it is common practice to rely on the existence of reusable artefacts. Reuse of artefacts comes in different flavours [65]. *Planned reuse* [47] refers to the situation where:

1) a *recurring development issue* has been identified,
2) *one or several solutions* to this issue have been developed, and
3) the software artefacts (e.g., documentation, models (if any) and code) realizing the solutions are *packaged in a reusable unit* and made available for reuse. At the programming level, reusable frameworks and libraries are in widespread use.

The philosophy of model-driven engineering (MDE) is that during development high-level specification models of a system 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. Reuse in MDE is achieved through (domain-specific) modelling languages, which capture the essential concepts relevant to the development of the software at a given level of abstraction, and through model transformations that assist developers in transitioning from one layer of abstraction to another towards a concrete solution and implementation. To be effective in this framework, models that represent the system at a given level of abstraction need to be generic enough to allow for (ideally many) possible solution-specific refinements of the system at lower levels. This is even more true for models that are meant to be reusable.

Interfaces have been effectively applied at the programming level—but more recently also at the modelling level—to enable reuse within and across abstraction levels during software development [63]. This chapter reflects on the challenges that developers face when defining interfaces for higher levels of abstraction or for units encompassing multiple solution variants. In particular, we concentrate on the problems caused by the *finality of signature declarations*. *Concernification*, which we discussed in-depth in Chapter 3, makes it possible to raise the abstraction level of frameworks and in particular their API. Raising the abstraction of APIs exposes a difficulty when the signature of operations depends on the particular feature the user wants to use. One case where we observed this was in the *Android Notifications API* (see Chapter 6) where a new feature was added to the API that required augmentation of an additional argument to the constructor of the builder for notifications. Another example is the *Association* concern [16] that raises the abstraction level of collections. In the design of the concern, common functionality like adding and removing of elements, could not be designed at a high-level in a common way for all children features.

8.2 On Reuse, Interfaces and Signatures

As described in detail in Section 2.4, in the context of reuse there are at least two clearly distinct software development roles that arise. The *designer* of the reusable unit is an expert of the domain of the development issue that the unit addresses. She has a deep understanding of the nature of the issue, but does not know in what contexts the reusable unit may be used. To facilitate this, the designer strives to make the reusable unit as versatile and generic as possible, so that the solutions can be applied in a wide variety of reuse contexts.

A *user* of a reusable unit on the other hand is an expert of the application he is developing. He is aware of the specific requirements of the system he is working on. The user might desire to solve a specific development issue using an existing reusable unit. The user does not know the implementation details of the reusable unit, and only needs to be able to know which solution of the reusable unit is most appropriate. Furthermore, the user needs to know how to customize and use the chosen solution.

8.2 On Reuse, Interfaces and Signatures

Experience has shown that reuse of artefacts with explicitly defined *interfaces* leads to high levels of reuse maturity [63]. Interfaces specify a contract that bridges the worlds of the designer of a reusable unit and the (hopefully many) users of the reusable unit. Furthermore, applying the information hiding principles [85], interfaces make it possible to hide solution complexity and properties within a reusable unit, and hence significantly reduce the complexity that the users of the reusable unit need to deal with.

A very common way of providing a static interface that allows the users to trigger functionality provided by a reusable unit is an *operation signature* (or *service signature*). A signature is made of the operation *name*, of a *set of parameters*, each one comprised of a formal *name* and *type*, as well as the *type of value* returned by the operation, if any. Finally, some modelling or programming languages also include in an operation signature the set of exception types that might be raised at runtime when the operation is invoked.

Most statically compiled modelling or programming languages require signatures to always be specified in their entirety. This forces the designer of a reusable unit to decide on the exact number[1] and type of every parameter of an operation before she can declare an interface or signature for it. Once declared, the signature is *set in stone*, i.e., the existing parameters are immutable, and no new parameters can be added. It is of course possible to *overload* methods, i.e., declare new methods with the same name and additional parameters, but then the API contains *several* methods.

While this might be sometimes appropriate, the *finality* of signature declarations poses difficulties to the designer and user of a reusable model in certain situations. These situations are summarized here and then elaborated further in Section 8.3:

1. *When a reusable unit encapsulates several solutions, it is* sometimes *difficult for the designer to come up with a common* final *signature* that works for all of the solutions. The situation is similar when a modeller needs to define a signature in a model at a level of abstraction that allows for different solution refinements.

2. With final signatures, it can be *difficult* for the designer *to add a new feature to a reusable unit* in a non-intrusive way. The situation is similar for modellers who want to add a new feature to a model of an existing product line.

3. *When signatures are used to define callback interfaces* that allow a reusable unit to trigger reuse-context-specific functionality, *it can be difficult* for the designer *to define a* final call-

[1]Some statically compiled languages support the declaration of signatures with an arbitrary number of parameters. For instance, in Java with the *varargs* feature [82], or in Go [116] and Python [87] (an interpreted language) with the *variadic function* feature.

back *signature that is ideal for any reuse context*. The situation is similar in reusable models where the reusable behaviour has to trigger reuse context specific behaviour.

4. *When* a programmer or modeller designs a reusable unit x and reuses reusable unit y with different variations, *it is* sometimes *difficult to make a final decision*, i.e., which variant from y to use, since the reuse context of x is unknown. As a result, *it is difficult to define a* final *signature* for the functionality offered by x, and it is difficult to specify behaviour in x that calls operations of y if the variants in y have different signatures.

8.3 Problematic Situations

This section describes in detail four problematic situations that we identified.

8.3.1 Difficulties Defining a Common Interface for Alternative Implementations

When designing a reusable unit with several variations for a common purpose, a designer generally aims at providing a common interface to the user that is independent of the concrete variation being used by the user. This strategy is highly beneficial, because it allows the user to maintain a design that stays at a high level of abstraction without depending on a concrete variation. A common interface makes it possible for the user of a reusable unit encapsulating multiple solution variants to replace a chosen variation with another one exhibiting different qualities without significant effort. In model-driven design a similar situation occurs when the designer uses abstraction to delay deciding on solution-specific details. An interface at a given level of abstraction makes it possible to explore different solution-specific refinements during development. Unfortunately, when signatures are final once they are declared, it is difficult to provide a common interface in situations where different implementations of an operation achieving the same functionality require different parameters to execute.

For example, *collections* are reusable units that are used very frequently, and they come in many variants offering different functionality and exhibiting different non-functional properties. In programming languages, collections are typically grouped together so that they can be treated in a similar way at a high level of abstraction. *Java* and *C#*, for instance, use inheritance to group different kinds of collections and algorithms to process them.

But defining a common interface for all kinds of collections is difficult. For example, the signature for adding an element to a plain collection is typically `add(Element)`, whereas adding an element to a map is provided by an operation with the signature `add(Key, Element)`. This can be problematic if a user wants to treat maps and collections in a uniform way, e.g., to check whether a collection/map contains a certain element.

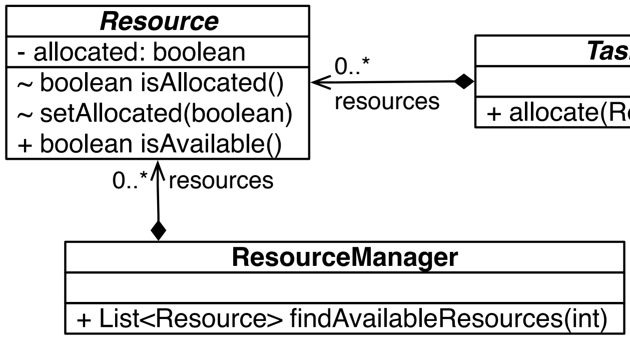


Figure 8.1: Resource Management Base Design

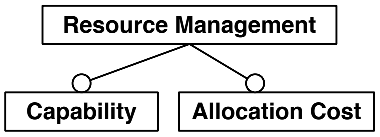


Figure 8.2: Resource Management Feature Model

8.3.2 Difficulties Adding New Functionality

Consider *Resource Management*, which is a recurring functionality required in many applications. At its core there are *Resources* which can be allocated to *Tasks* (signatures `isAvailable()` of class `Resource` and `Task.allocate(Resource)`), and a `ResourceManager` class provides operations to find and allocate a number of resources to a given task (`findAvailable-Resources(int)`). A corresponding class diagram is shown in Figure 8.1.

Some applications might need additional functionality, which can be seen as additional features of the *Resource Management* unit. Figure 8.2 shows a feature model with two optional variants. *Capability* provides the ability to differentiate resources according to their capabilities, and *Allocation Cost* augments the behaviour of resource allocation to consider individual resource allocation cost.

Generally, APIs are *set in stone* once published. The general recommendation is therefore to put a lot of effort into initial API design [19]. However, APIs need to evolve to accommodate changing requirements and integration of alternative solutions. In the case where the additional or optional functionality needs to execute together with existing functionality *and* needs additional information from the user to process, the situation is problematic. Since the signature of the operation providing the existing functionality is final, it is difficult to add additional parameters to it, and doing so would invalidate every place where the operation is being used.

For example, the optional feature *Capability* extends resources with capabilities. Such functionality might be required by a *crisis management system* (CMS) that needs to allocate workers to missions, and differentiate the workers according to their capabilities, such as *driver*, *fire fighter*, *first aid provider*, etc. In this case, a resource has an associated set of capabilities. Additionally, resources are allocated to a task because they fulfill a needed capability. Therefore, in order to look for available resources, additional information is required from the user: the desired capability for which resources are sought for has to be specified. Hence, some of the operation signatures of the reusable unit would need an additional `Capability` parameter: `findAvailableResources(int, Capability)` of `ResourceManager`, `allocate(Resource, Capability)` of `Task`, and `isAvailable(Capability)` of `Resource`.

When signatures are final, though, it is impossible to provide a clean common interface to the user of *Resource Management* that can be used at a high level of abstraction, i.e., with and without the optional *Capability* feature. In programming, optional functionality of a method is often supported by overloading, or triggered by parameters at the end of the signature. The behaviour of the method checks the value of the parameter, and if the user passes in a specific value, e.g., `false` or `null`, the optional functionality is not executed. If the language has support for default values, the designer can specify `false` or `null` as the default value, which relieves the user from needing to do so. However, the user still needs to consider these parameters, understand their intent, make a decision on whether to use them or not, and consistently use the correct ones if there is more than one.

In our example, the designer of *Resource Management* is forced to provide a common `findAvailableResources(int)` method that provides the general ability to find available resources, *as well as* an additional overloaded `findAvailableResources(int, Capability)` method to support the optional feature. Alternatively, the designer can choose to define only one operation `findAvailableResources(int, Capability)`, and specify in the documentation of *Resource Management* that users who do not want to use the capability feature must pass `null` as an argument when invoking the operation at run time.

Neither of the workarounds are ideal, though. The former is error-prone, because users who have made the decision to use capabilities should only call the operations that have the *Capability* parameter. If by mistake they invoke one of the operations that does not handle capabilities, the consistency of resource management is jeopardized. The second workaround is at the least confusing, because users who do not want to use capabilities must upon every operation invocation pass `null` as a value.

8.3.3 Difficulties Providing a Callback Interface that fits all Reuse Contexts

Many frameworks take over the flow of control of an application and use the *callback* technique to execute application code when needed. This requires the designer of the framework to specify *at design time* an interface that defines the operation signature of the callback method that the framework will call when it wants to hand control to the application. However, the designer does not know in which contexts her reusable unit is going to be reused, and therefore has difficulties defining a final signature for the callback that will work in all contexts.

For example, the *Allocation Cost* feature (see Figure 8.2) augments *Resource Management* with the functionality `estimateTotalAllocationCost(Set<Resources>)` that can determine the total cost of allocation given a set of resources. Since the allocation cost of an individual resource typically depends on the state of the resource and since the designer of *Resource Management* does not know anything about the state of the actual resources, the behaviour that calculates the individual allocation cost of a resource needs to be provided by the user of *Resource Management* and invoked through a callback. To this aim, the designer defines a parameterless callback signature `estimateAllocationCost()` that needs to be implemented for the `Resource` class by the user.

Unfortunately, this can be problematic, e.g., when *Resource Management* is reused in the context of the CMS where workers are allocated to missions. Missions typically are performed in a certain region or location. When allocating a specific worker to a mission, the allocation cost depends on the distance between the worker's current location and where the mission is taking place. The current location of the worker—the resource—is part of the state of the worker, and accessible in the callback method. Unfortunately, the location of the mission—the task—is not accessible. Possible steps the designer can take to anticipate this problem is to include a dummy `Object` parameter in the callback. The user then has to create a class that subclasses `Object` with attributes to hold the desired application-specific state. Additionally, the user then has to tell the reusable unit at runtime which concrete instance of this new class to pass to the callback. This does work but is very cumbersome.

8.3.4 Difficulties Delaying Design Decisions

An additional difficult situation arises in the context of software product line development (SPL) and reuse hierarchies. Software product line development is an approach that is beneficial when developing a collection of similar software systems—a family of products—that share some commonalities and differ in a well-defined set of features exposed by the SPL. To increase reuse, functionality that has been identified as common is encapsulated within shared software artefacts that are reused within multiple products. Unless the shared functionality is absolutely identical for all products, it is again difficult for the designer of the shared reusable unit to define an interface that satisfies the needs of each individual product. A specific product or feature might require a slightly different variant of the functionality encapsulated within the reusable unit, which in turn could require additional information to process. This resembles the difficult situations explained in

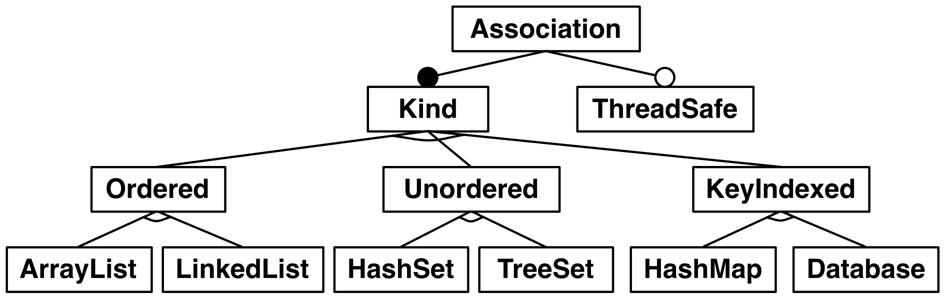


Figure 8.3: Association Feature Model

sections 8.3.1 and 8.3.2. In a way, the design decision of which concrete product to produce in an SPL is delayed until the SPL is configured with the desired set of features. It is only at that time that it is absolutely clear what information is needed to process some shared functionality.

This situation is even more prominent when a designer of a reusable unit wants to reuse (within her own design) some other reusable unit that offers different variations. For example, in [16] the authors describe the design of a reusable unit *Association* encapsulating various association designs based on different collection types with different features: support for unique elements, multi-threading, enforcement of minimum and maximum amounts of elements, etc. A feature model of such a reusable unit is shown in Figure 8.3. Internally, it uses Java collection types to implement the different association features, as suggested by the names of the leaf features. However, it would be possible to use collection types of another object-oriented programming language instead.

The designer of *Resource Management* can use the *Association* reusable unit to keep track of one or more resources that are allocated to a task. The designer of *Resource Management* could reuse any of the variants provided by *Association* to realize this relationship. However, each variant exhibits different properties, in particular with respect to non-functional properties, e.g., performance or memory usage. Since *Resource Management* is a reusable unit itself, it is impossible for the designer of *Resource Management*, who now also is the user of *Association*, to determine the most appropriate alternative to reuse, since this decision depends on the context in which *Resource Management* will be reused in the future. Ideally, the designer would like to delay the decision [64] of which specific variant of *Association* to reuse and complete her design using a common, high-level interface.

8.4 Validating the Need for Flexible Signatures

This section provides proof of the existence of the problematic situations at the implementation level. We describe an empirical study we conducted on the Java Platform API as well as techniques and workarounds used in various programming languages to overcome some of the problematic situations.

8.4.1 Exploring the Java Platform API

This section presents an empirical study that we conducted to demonstrate the potential usefulness of signature extension. Java ships with an extensive runtime library of reusable classes providing different kinds of commonly needed functionality. We examined the Java 10.0.1 runtime and focussed our attention on the *java.base* module, which contains 5746 classes, of which 3245 are in the `java` and `javax` root packages.

The Java programming language has evolved since version 1.0 (released in 1996) and contains evolution information in the source code. We extracted the following information from the source code of the *java.base* module (version 10.0.1) *for each public method of public classes*:

- the method's signature
- whether the method overloads another method
- whether the method is marked as *deprecated*, which is the Java way of saying that a method is outdated and should not be used anymore. If available, we also extracted since which version the method is deprecated[2], and from the Javadoc's `@deprecated` tag the comment explaining the deprecation, and whether the method was replaced with other alternative methods
- in which version the method was introduced (provided using Javadoc's `@since` tag)[3]
- whether the method implements/overrides another (from a superclass/interface)

To accomplish this task, we used the *AST Parser* of *Eclipse JDT* and parsed each source file contained in the corresponding source code file of the *java.base* module. We only considered the main type declaration of the java file and hence ignored any additionally defined inner classes. In total we found 1104 public classes and 11720 public methods. The following tables present the gathered information.

As Table 8.1 shows, close to 35% of all the public methods offered by the *java.base* module are overloaded. This is a considerable number. 41 methods are overloaded in subclasses, which is an indication of the presence of alternate features that need additional or potentially a different set of parameters. 36 overloaded methods are deprecated, which means that potentially due to evolution they had to be replaced by other operations with different parameters.

[2]Since the *since* attribute for `@Deprecated` was only introduced with Java 9, the earliest value retrieved is version 9.

[3]If a method does not have this information, we assumed that it was introduced in the same version as the class, which is provided with the `@since` tag in the Javadoc information of the class.

Table 8.1: Gathered Data of the *java.base* Module

Module	**java.base**
Number of public classes	1104
Number of public methods	11720
Number of overloaded methods	4072
Number of methods overloaded in subclasses	41
Number of deprecated methods	138
Number of overloaded methods that are deprecated	36

We therefore examined the overloaded methods in more detail, looking at each group of overloaded methods. A group consists of all methods of the same name in the same class. Table 8.2 shows the results of our investigation.

Table 8.2: Gathered Data of All Method Groups of the *java.base* Module

Module	**java.base**
Number of method groups	1488
Minimum group size	2
Maximum group size	20
Average group size	2.74
Number of groups with a deprecated method	21
Number of groups where a method was introduced in a later version	403
Number of groups where a deprecated method exists and a method was introduced (potentially replacing the deprecated method)	10

There were a total of 1488 groups of overloaded methods. 403 of them, i.e., 27%, contained methods that were introduced over time in later versions of Java. Again, some of those methods could represent situations in which new features were introduced that required different parameters. 21 of those groups had a deprecated method in them, and for 10 of those new methods were introduced at the same time. These could represent situations in which new features were introduced, and as a result, methods required a new set of parameters to be able to continue to provide their functionality in the presence of the new feature.

We further manually searched the Java base module to find situations other than collections where signature extension could be helpful.

8.4.1.1 Adding Optional Functionality

With Java 1.4, support for different character encodings other than the platform's default encoding was added as a new feature through the additional class `java.nio.charset.Charset`. As a result, several methods across the API were added to take a `Charset` as an argument, or alternatively a *String* argument designating the name of the character set, e.g., *UTF-8*. For example, in addition to the method `java.lang.String.getBytes()`, `java.lang.String.getBytes(Charset)` as well as `java.lang.String.getBytes(String)` were added. Another example affected by this change is the method `encode` of `java.net.URLEncoder` (the same applies to `decode` of `URLDecoder`). In this case, the initial method `encode(String)` was deprecated and replaced with `encode(String, String)`. In Java 10, the method `encode(String, Charset)` was added[4]. In total, due to the introduction of the character encoding feature in Java 1.4, there are now 25 overloaded methods with an additional *Charset* argument.

Similarly, Java 1.4 introduced a new abstract class `java.net.SocketAddress` with one subclass `java.net.InetSocketAddress` which implements an IP socket address consisting of an IP address and port. This was added alongside the existing `java.net.InetAddress` (which only consists of the IP address). When dealing with sockets, methods requiring an `InetAddress` therefore also need an argument for the port. As a result, through the addition of `SocketAddress`, those methods (such as in the classes `DatagramPacket`, `DatagramSocket`, and `MulticastSocket` of the `java.net` package) had to be overloaded with the alternative functionality. Interestingly, in `java.net.Socket` and `java.net.ServerSocket`, instead of overloading the existing constructors with a `SocketAddress` argument, each class has a constructor to create an unbound socket and an additional method accepting a `SocketAddress`, which needs to be called after instantiation.

We located another additional optional functionality in classes that write to files. The two classes `java.io.FileWriter` and `java.io.FileOutputStream` allow optionally to append to an existing file instead of writing from the beginning. This is specified using an additional `boolean` argument in the respective overloaded constructor. What is interesting to note is that `FileWriter` does not have an overloaded constructor allowing the character set to be speci-

[4]See https://bugs.java.com/bugdatabase/view_bug.do?bug_id=8178081 and https://bugs.java.com/bugdatabase/view_bug.do?bug_id=8183743

fied (see above). This makes it impossible to use both new features simultaneously, i.e., have a file writer that uses a specific character set and appends to an existing file. The creators of the Apache Commons IO library[5], which supplies input/output utilities, noticed this lack and provide a class `org.apache.commons.io.output.FileWriterWithEncoding` to accomplish this, i.e., it contains both optional functionalities to specify a custom character set *and* appending to a file.

8.4.1.2 Providing a Common Interface for Alternative Implementations

In Java, a design decision was made to use two disjoint class hierarchies for maps (`Map`) and collections (`Collection`). One of the reasons being that forcing maps to be collections or vice versa "*[...] leads to an unnatural interface*"[6]. As described in Section 8.3.1, this is problematic when collections should be treated in a uniform way. Furthermore, simply iterating over a map is not directly possible. Instead, the user needs to make the explicit choice to iterate over the *key*, *value* or *entry set*. The entries itself are key-value pairs that are exposed to the user. This can lead to inefficiencies when a user is not aware of this particularity, and writes code that iterates over the keys, only to then retrieve the corresponding value for each iteration step using the `get(key)` method.

In contrast, C# only uses one hierarchy, i.e., `Collection`, that also contains maps (`Dictionary`). However, because a collection is generic and typed to contain elements `E`, for maps the type `E` is a `KeyValuePair`, where each instance provides an entry of the map with a key and its value. Because the `add` operation is defined in the top-level class `Collection`, the user can call it also on a map, but needs to provide an instance of a `KeyValuePair` as a parameter. In order to help the user who does not need the additional layer of abstraction and hence does not mind writing dictionary-specific code, an additional, more convenient method (`add(K, V)`) is defined in `Dictionary` that transparently takes care of creating a `KeyValuePair` instance for the user.

8.4.2 Exploring Workarounds in Programming Languages

Looking at the Java Platform API confirmed the occurrence of the first two difficulties outlined in Section 8.3. In order to overcome these at the programming language level, there exist certain workarounds.

[5]See `https://commons.apache.org/proper/commons-io/`.

[6]`http://docs.oracle.com/javase/8/docs/technotes/guides/collections/designfaq.html#a14`

8.4.2.1 Adding Additional Functionality

In the situation where a reusable unit is already being used and additional functionality is added, the goal is often to keep binary compatibility [51]. Guidelines for defensive interface evolution suggest several ways to achieve minimal impact on the user in addition to the basic strategy outlined above [35]. Among them are *defender methods* (also known as *virtual extension methods*) introduced in Java 8 [46], abstract classes with default implementations, or Eclipse's way of specifying additional interfaces that extend the existing interface [52]. Other workarounds that are suggested for API evolution within Eclipse are marking the old method as *deprecated* and forwarding the call to the new method in its implementation.

We also observed an interesting case in the *Android Notifications API* (see Chapter 6) where notification channels were introduced as a new (mandatory) feature to the API. Beginning with its introduction, applications using notifications are required to create a notification channel and specify it when building notifications. To enforce this, the existing constructor of `NotificationCompat.Builder` was augmented with an additional argument specifying the channel ID. The existing constructor was left in the code base and marked as deprecated. In the resulting concern interface of *Android Notifications*, the new constructor is located within the *Notification Channel* feature. If a user chooses this feature, the resulting API, however, provides both constructors. The user therefore still sees and gets access to both versions of the constructor. This requires additional cognitive effort and can cause inconsistent usage of the API.

8.4.2.2 Providing a Callback Interface that fits all Reuse Contexts

The third problem is a problem developers commonly face. The query "*pass extra argument to callback function*" on *Stack Overflow* results in 259 questions (or their answers) being matched. Various programming languages provide workarounds to overcome the difficulty of the user to access extra information within a callback. Here is a non-exhaustive list of techniques that we have observed at the code level to deal with the situation, some depending on specific implementation language features.

1. The user can define a custom interface that contains a method with the additional parameters. The user then implements the original interface, which when invoked determines the value for the additional parameters and forwards the call to the custom interface with the additional arguments[7].

2. When a programming language supports anonymous inner classes, the user can declare an

[7]This technique is illustrated in the Android example *XYZTouristAttractions* in the `AttractionListFragment` Java class using the custom `ItemClickListener` interface [112], for example.

operation with parameters that hold the additional information. When called, the method creates an anonymous instance of the callback interface and returns it, which can then be registered with the framework. When the callback is received, the additional parameters of the operation are accessible from the anonymous inner class[8].

3. In Python, lambdas, partial functions [128], or function decorators [115] may be used to augment framework-defined callbacks with additional parameters.

4. In JavaScript, it is possible to call a function with more arguments than defined in its signature. Inside the function, arguments can be accessed using the `arguments` object [121]. This could help the user to access additional arguments in a callback, but the designer would also have to pass additional values in the call. Alternatively, it is possible to bind additional parameters to a function using the `bind` function [56, 76].

5. Similarly, in C++, a *bind* method was introduced in C++11 [110] and a separate library called *boost::bind* from the C++ collection of libraries called *boost* also provides a *bind* method [30, 88]. They allow a developer to create a new function pointer to be created with the original function's arguments bound or rearranged, and also to add additional parameters, if needed.

8.5 Summary

This chapter discussed the finality of method signatures once they are defined. We identified four difficulties that are caused by the finality of method signatures and outlined them with brief examples. The difficult situations affect both the designer of a reusable concern and the user of a reusable concern. To validate that this is indeed a problem on the implementation level, we conducted an empirical study on the Java Platform API. This API has evolved over a long period of time and we found large numbers of overloaded (35%) and deprecated methods. We further studied overloaded methods by grouping methods by name and found evidence for the first two difficulties—adding optional functionality and defining a common interface for alternative implementations. Furthermore, we reported on workarounds in several programming languages to overcome the difficulty of flexible callback interfaces. In the next chapter we propose a *Signature Extension* approach that helps to overcome these difficult situations.

[8]This technique is illustrated in the Android example *XYZTouristAttractions* in the `AttractionsGridPagerAdapter` Java class [113], for example.

9 The Signature Extension Approach

The previous chapter discussed the four difficult situations in detail and showed evidence of these at the implementation and modelling level. In this chapter we present an approach that improves reuse support by addressing the difficulties related to the finality of signature declarations. We refer to the approach as *signature extension*, because it allows the signature of an operation to be extended in a similar way as classes can be extended. We first describe in Section 9.1 how in our approach both the designer and the user can structurally extend signatures by declaring additional parameters. We then outline how we allow the designer and the user to specify extended behaviour that can process the additional parameter. In Section 9.2 we detail how class diagrams as defined in the RAM language in CORE is extended with *signature extension* support and Section 9.3 explains how the structural composition of class diagrams is updated. Section 9.4 discusses related work. The chapter then concludes with a summary in Section 9.5.

9.1 Requirements for Extending Signatures

This section describes first the requirements to extend signatures structurally that allows both a designer and user to structurally extend signatures by declaring additional parameters. We then describe the requirements to specify extended behaviour to support processing additional parameters when extending signatures structurally.

9.1.1 Extending Signatures Structurally

The *designer* of a reusable unit knows the design details of every optional or alternative feature encapsulated by the reusable unit. Her goal is to define signatures that provide a common interface for several variants, and then separately specify for a specific variant any additional information that might be needed. In order to support this, our approach proposes the following:

- The number of parameters of a method signature within a reusable unit is not set in stone.

- It is possible for the designer of a reusable unit to incrementally extend a method signature by *adding* additional parameters to it.

The *user* of a reusable unit on the other hand is agnostic about the implementation details of the unit (information hiding) and is only presented with the interface of the unit of reuse. To address the case where a user needs additional application-specific information from the reusable unit to implement a callback behaviour, our approach proposes the following:

- The designer can prepare the design such that signatures can be extended by users of the reusable unit.

- The user of a reusable unit may then add application-specific parameters to such a signature defined by the reusable unit when implementing reuse-context-specific callbacks requiring additional information to process.

In general, we observe two kinds of callbacks in terms of control flow. In the first situation, an operation of a reusable unit is invoked and in turn directly causes invocation(s) to the callback method. For example, in the example provided in Section 8.3.3, the `estimateTotalAllocationCost` operation directly calls the `estimateAllocationCost` operation of `Resource`. In the second situation, the callback occurs due to an event in the environment, i.e., it is not triggered by the user of the reusable unit. For example, listeners are usually registered when setting up an application, and the callbacks are triggered (indirectly) by the end user of the application, e.g., by clicking on a button.

We therefore propose that a designer of a reusable concern can explicitly declare a signature that extends another signature with additional parameters within the same concern. This explicit declaration distinguishes a signature extension from declaring a new, additional overloaded method. The extending signature can declare any number of additional parameters and may repeat the declaration of parameters already declared in the extended signature for convenience, if appropriate. In this case, existing parameters are repeated and mapped to the parameters of the extended method. Providing an explicit mapping for parameters also allows the designer to reorder parameters, if desired. Any unmapped parameter is automatically considered an additional parameter.

The user on the other hand should not be allowed to extend a signature from a reused concern since this would require the user to know the internals of the reused concern. The designer, however, can prepare the design for extension in the case where callback signatures need to be provided by the user.

9.1.2 Extending Signatures Behaviourally

The requirements for specifying the behaviour to deal with additional parameters introduced by signature extensions are very different for designers and for users. The designer, as the expert of the reusable unit, knows the detailed design of all variants that the concern offers, and can therefore add new, detailed behaviour that deals with the additional information, but also change the existing behaviour. She needs to be able to:

- Specify behaviour that defines what is done with the additional parameter(s) within an extended method, in case the existing behaviour is not affected at all, or
- specify how to adjust the existing behaviour when additional parameters are present. This can involve manipulating the return value of the original behaviour.

Also, whenever a method inside the unit of reuse calls a method that is potentially extended, she needs to be able to:

- Specify how to determine the value for the new parameters in all locations where an extended method is invoked. In some cases, she might simply want to delegate the responsibility for determining the additional parameter value further. She can accomplish this by extending the signature of the methods that call the extended method accordingly.

The user, however, does not know and should not have to deal with the implementation details of the reusable unit. Because the designer planned for the signature extension, he only has to focus on mapping to his own context and, if necessary, provide the callback value(s) to the reusable unit as intended by the designer.

Support for structural signature extension significantly increases the power of abstraction offered by signatures, and hence has a direct impact on the concernification of frameworks and their APIs presented in the first part of this thesis. Hence, we implemented support for structural signature extension within TouchCORE as outlined above and detailed in the following section. Support for behavioural signature extension does not have a direct impact on concernification, involves aspect-oriented sequence diagram composition and therefore is out of the scope of this thesis. We will therefore only briefly outline how behavioural signature extension support can be added to TouchCORE at the end of this chapter.

9.2 Adding Structural Signature Extension Support to Class Diagrams

The signature extension approach should be applicable to any software development language that uses signatures to define interfaces. The approach allows a designer to incrementally define

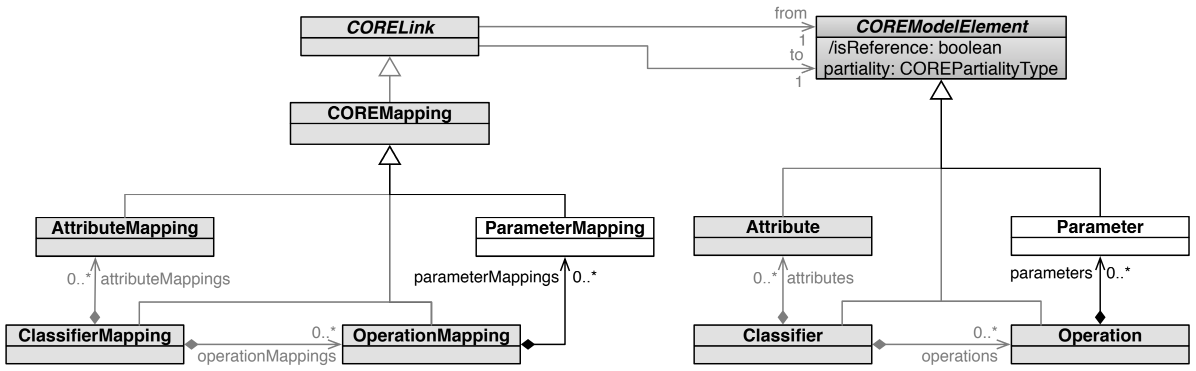


Figure 9.1: Class Diagram Metamodel Excerpt for Mappings and Classes

signatures, as well as use partially defined signatures before all extensions are known. We show how the class diagrams in CORE is extended with support for structural extension of operation signatures. The class diagrams are defined in the Reusable Aspect Models (RAM) language and are based on UML class diagrams [81].

RAM extends the CORE metamodel (see Section 2.4.5 and Section 2.5) and provides concrete `COREMapping` sub-classes for mapping specific types. It does so to put them into a containment hierarchy corresponding to the containment of the elements, i.e., an `OperationMapping` is contained within a `ClassifierMapping` to map operations of a class that is mapped. The RAM metamodel therefore already provides mappings for operations. Furthermore, each concrete mapping class restricts the mappable types for the mapping to their respective type in the class diagram (i.e., `ClassifierMapping` maps `Classifier` and so on).

However, support for mapping parameters is missing and needs to be added. As enforced by the super-class `CORELink`, the mappable elements need to be a sub-class of `COREModelElement`. Therefore, to introduce mappings for parameters, a `Parameter` itself needs to be a sub-class of `COREModelElement`. Consequently, this then allows the introduction of `ParameterMapping`. Figure 9.1 shows the excerpt from the class diagram metamodel with the new `ParameterMapping` class and the new super-class for `Parameter`. New or updated classes are shown in white whereas existing untouched classes are shown in grey. New relationships are shown in black to distinguish them from the existing ones (shown in grey).

By inheriting `Parameter` from `COREModelElement` parameters now also have partiality and mapping cardinalities (see Section 2.5.4). This is enough to provide full support for the requirements set out in the previous section. The designer can now specify an operation signature and add a partial parameter with a mapping cardinality of `{p=0..*}`. If the type of the parameter does not matter, e.g., because the current model itself does not make use of the data, the type of the

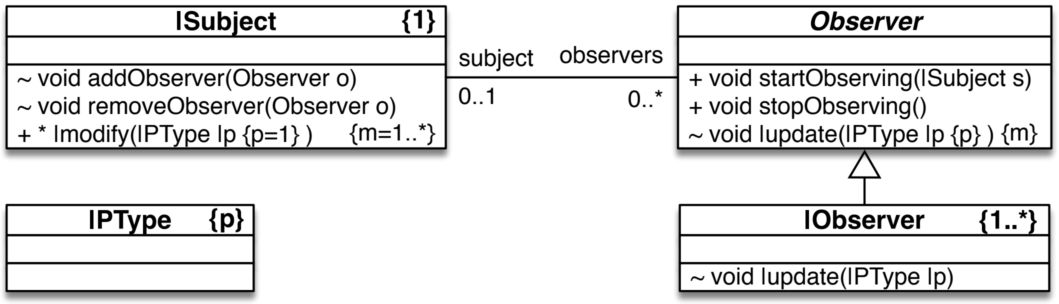

ISubject {1}
~ void addObserver(Observer o)
~ void removeObserver(Observer o)
+ * |modify(|PType |p {p=1}) {m=1..*}
subject
0..1
observers
0..*
Observer
+ void startObserving(ISubject s)
+ void stopObserving()
~ void |update(|PType |p {p}) {m}
|PType {p}
|Observer {1..*}
~ void |update(|PType |p)

Figure 9.2: The Updated Structural Design Model of the *Observer* Concern with Support for Signature Extension

parameter can be set to the so-called *any type*. This type is graphically represented with an asterisk ('*') and is used, for example, in the *Observer* design as the return type of the partial `modify` operation (see Section 2.5). If the parameter is required to have certain operations that can be called, the parameter type can be set to another partial class which defines the required (partial) operations. If within the design there is another operation that is called by the operation with the extended signature, and the designer intends to pass the additional parameters along in that call, the designer can add a partial parameter to this operation as well. The cardinality however depends on how many parameters the first operation has. Therefore, the mapping cardinality refers to its cardinality: `{p}`.

To illustrate this we consider again the *Observer* concern, which was used as an example in the background chapter (see Section 2.5) along with the mapping cardinalities in Figure 2.16 on page 26. In the original design the complete `Subject` instance is passed to registered observers as a parameter of the `update` operation. Assuming the designer wants to only send the updated state of the subject that has been modified instead of the entire subject, using the signature extension support, she could add a partial parameter `|p` to `modify` that will then be passed to the `update` operation. Figure 9.2 shows the updated structural design. The mapping cardinality of `update`'s parameter is dependent on the cardinality `{p}` of the parameter of `modify`. The same applies to the partial parameter type class `PType`, which ensures that the parameter type of both operations is the same.

A possible mapping for a reuse of this *Observer* in the context of a bank application with accounts is shown in Listing 9.1.

When mapping an operation in an extending model, the parameters that are used both in the extended and in the extending signature need to be mapped. By mapping the operations and declaring additional parameter(s), the designer explicitly states that the original signature is extended.

Partial parameters with mapping cardinalities and parameter mappings can handle the first two

Listing 9.1 Example Mappings when Reusing *Observer*

```
|Subject ⟶ Account
  |modify ⟶ setBalance
    |p ⟶ newBalance
|PType<newBalance> ⟶ double
|Observer ⟶ AccountWindow
  |update<setBalance> ⟶ balanceUpdated
    |p<newBalance> ⟶ newBalance
```

difficulties for designers (common high-level interface and optional functionality). We now look in more detail at how the other two difficulties—extending a callback interface and delaying design decisions—can be supported with signature extension.

Extending a Callback Interface As discussed earlier, the designer of a reusable unit knows the details of the design and therefore only the designer is aware of how to correctly handle additional parameters for callbacks. If the callback is executed within the control flow of an operation of the reusable unit that the user invokes explicitly, then the designer can include in her design of the operation the possibility of adding additional parameters. In the case where a callback happens at a point during execution that is not directly influenced by the user calling an operation of the framework, the designer can provide means to provide the value for the callback in advance. To support setting callback values, the designer needs to provide a way to store the callback value so that when the callback operation is invoked, the value can be accessed and passed as an argument.

In both cases, the designer must prepare her design to allow the extension. This makes sense, as the details of the design of the reusable unit are hidden from the user. Hence the user cannot specify the behaviour that deals with the additional parameter.

Figure 9.3 shows another variant of the *Observer* design pattern. The `notifyObservers` operation calls the `|update` operation of all the registered observers, but in this design the user can customize the signature of the `update` operation with one or several additional parameters. With the public operation `setCallbackValue` the user can explicitly set different parameter values for each registered observer. The subject stores these values in a qualified association with the observer as the key. Before invoking the `update` operation on an observer, the stored parameter value for that observer is read from this data structure and passed as an actual parameter to the call.

Delaying Design Decisions In CORE, when reusing another concern, automated information hiding reduces the visibility of any public elements provided by the reused concern in order to hide any implementation details to the next level up. With delaying decisions (see Section 2.4.4),

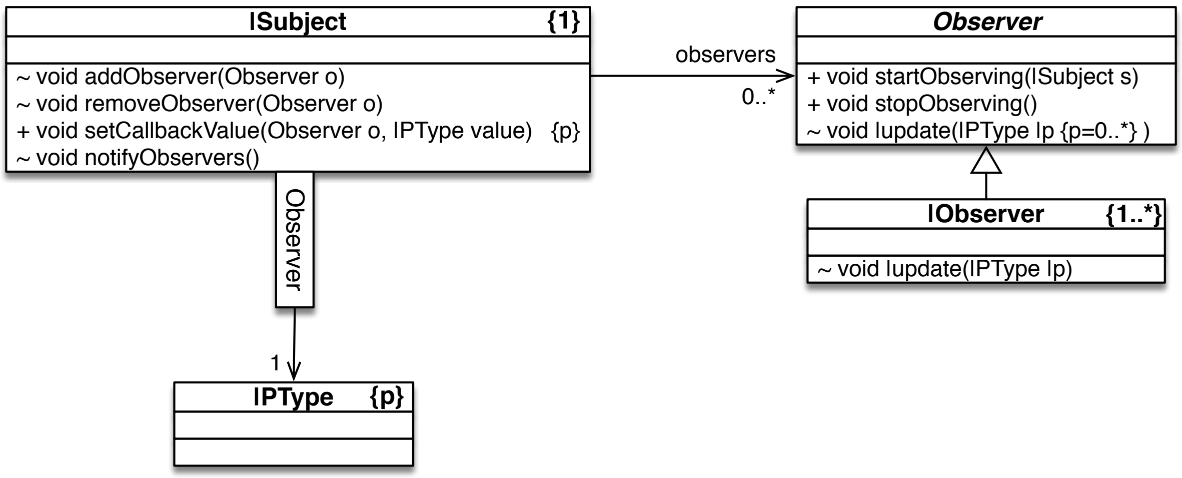

ISubject {1}
~ void addObserver(Observer o)
~ void removeObserver(Observer o)
+ void setCallbackValue(Observer o, IPType value) {p}
~ void notifyObservers()
observers
0..*
Observer
+ void startObserving(ISubject s)
+ void stopObserving()
~ void Iupdate(IPType Ip {p=0..*})
IObserver {1..*}
~ void Iupdate(IPType Ip)
Observer
1
IPType {p}

Figure 9.3: Supporting Custom Callback Values in the *Observer*

however, some decisions—feature selections—can be left undecided, and hence re-exposed, which allows the user at the next level up to make the best feature selection for his reuse context. For example, when reusing the *Association* concern, the designer of a reusable concern might decide to delay the decision of which concrete ordered collection to choose. In general, this works very well when the signatures are final (as is the case for ordered collections). If the final signature depends on the final feature selection, this poses a problem. For example, for key-indexed collections, choosing the *Database* feature requires an additional parameter for most operations that identifies the database connection.

When delaying the decision, the designer—who is the user of the reusable unit—can already use the common, high-level interface to complete her design. It is the responsibility of the designer who makes the choice to delay the decision to account for potential changes in the signature. Hence, the designer must test the outcome of the individual final selections to verify that the re-exposed features work with the design. In the case that a re-exposed feature changes a signature, due to the information hiding and the potential impact it has on the existing design, the designer needs to elaborate a design that supports the augmented signature. To do this, the designer can, for example, elaborate an optional feature that provides the design changes necessary for dealing with the extended signature of the reusable unit. The optional feature extends the existing reuse and selects the corresponding feature.

9.3 Structural Composition of Signatures

This section discusses how the existing structural composition algorithm is updated to support dealing with signature extension and the updated metamodel. The composition algorithm, also called weaving, is executed to compose models pair-wise when reusing a concern or when composing the final application.

Generally, in CORE, the philosophy is to only allow additive changes. The RAM class diagram composition merges mapped elements and copies unmapped elements to the resulting model (see Section 2.5.5). I.e., when a class from a lower-level model is mapped to a class in a higher-level model—either within an extension hierarchy or across concern boundaries through a reuse—the contents of the classes are merged.

During the design, if the designer wishes to use an element defined in another model—either within the extension hierarchy or from a reused concern—instead of referring to the element directly, a local copy is created in the current model. A mapping is established between the element in the other model and the local copy. This marks the element as a reference. I.e., the derived property `isReference` of `COREModelElement` shown in Figure 9.1 is `true` in such a case. Because of this, when extending a signature, the already defined parameters of the operation are part of the extending operation and are therefore mapped.

If a signature is extended by several sibling features, the structural composition needs to merge the signatures by creating a new operation that has the union of all the parameters. Additional parameters are added at the position relative to where the extending signature introduces them.

In order to give a designer the ability to prepare a signature for extension, we allow the specification of a partial parameter that is optional to be mapped. The lower bound of its mapping cardinality is `{0}`. Not mapping it in this case means that the user does not need additional parameters. In that case, the composition needs to ignore the partial parameter when merging signatures.

In addition to updating the actual composition algorithm, the data structure that keeps track of which model elements have been merged by the structural weaver also needed updating to keep track of merged and added parameters. This is because in RAM, the message views depend on the structural view [97]. The behavioural composition algorithm needs the data structure produced by the structural weaver containing the mapping from the original element to the target (composed) element in order to function correctly [101].

9.4 Related Work

As already mentioned in Section 8.4.2 of the previous chapter, there are many programming languages that provide features that make it possible to define methods with varying signatures (default values for parameters, variadic functions (Go [116] and Python [87]), the JavaScript *bind* feature, C++ *boost*, etc.). However, these features are limited in their use. For example, for variadic functions, all parameters have to use the same type. A parameter of type `Object` can be declared, but these instances then need to be downcast by the user. The same inconvenience occurs when the designer adds a single dummy object parameter to the signature that can be customized by the user.

Furthermore, a workaround that sets a custom callback value can be observed in AWT/Swing of Java. In order to facilitate using the same listener instance for multiple events/actions, components that fire an `ActionEvent` to registered `ActionListeners` provide a way to set an *action command* [83]. The action command is a string that is set using the `setActionCommand` method when building the visual components. When the `actionPerformed` callback method of a listener is invoked, the `ActionEvent` instance argument provides access to the action command. However, this only serves the purpose of understanding which action was invoked. The user has no means to provide any additional callback values. Furthermore, when an action command is not needed, the caller still has to pass `null` as the action command.

Some of the techniques described in Section 8.4.2, such as Python's *partial* function or the *bind* function of C++ and JavaScript, allow a developer to bind additional parameters to a method. These techniques make use of *partial application* which "*refers to the process of fixing a number of arguments to a function, producing another function of smaller arity*" [125]. I.e., the method with the greater number of arguments is the custom callback method desired by the user, the additional arguments are bound to certain values and the resulting method is the callback method that has the required signature. However, the user has to be aware of this advanced feature and apply it manually. The *Chromium* developers, the open source project behind the *Chrome* browser, also faced the need to support custom callback values [114]. They implemented their own *Bind* function with type-safety.

To the best of our knowledge none of these techniques make it straightforward for the designer of a reusable unit to express how the parameters in a signature of a method depend on the features the user of the reusable unit wants to use. Our signature extension approach should be applicable to any software development language that uses signatures to define interfaces. It is independent of the underlying programming language. Furthermore, signature extension makes it possible for the designer to specify how to deal with the additional parameters behaviourally.

The proposed signature extension approach has lots of similarities with aspect-oriented approaches [38, 59]. However, *AspectJ* [58], an aspect-oriented extension of Java, does not allow a programmer to change the signature of an existing method using an aspect. It is not even possible to add a new checked exception to the signature [62]. AspectJ provides behavioural extension points, i.e., it is possible to declare a *before*, *after* or *around* execution advice that augments a method with additional behaviour that executes either before the main body of the method executes, or after, or instead of it. We are proposing a similar mechanism to specify additional behaviour to deal with additional parameters.

9.5 Summary

This chapter described the signature extension approach and its integration into the class diagrams of the RAM language. The designer of a reusable unit is the expert with detailed knowledge of the unit's implementation. As such, it is the designer who knows best which signatures need extensions for which features, and how to extend the signatures. We argue that in order for extensions to be used by the user of a reusable unit, the designer of the reusable unit must prepare for the extension. To accomplish this, we allow the definition of partial parameters with a minimum mapping cardinality of `0`. If a user does not require additional parameters, they do not need to be mapped and will not appear in the final signature. A mapping cardinality can be referred to in another model element. This ensures that dependencies within operation signatures can be handled by enforcing that such parameters are mapped.

We showed that with the added support in the metamodel, the first two difficulties can be solved for the designer. In order for a user to be able to extend a callback interface, the designer has to prepare the extension. To accomplish this, she might need to include structure and behaviour that deals with the storage of additional values so they can be accessed when the callback is involved. For delaying of decisions, we argue that the designer needs to prepare this as well in the case where a re-exposed feature affects the signature of an operation. This makes sense, because due to information hiding the user does not see and should not be bothered by the implementation details of the reusable unit. In this situation, the designer that wants to delay a decision needs to elaborate an optional feature that provides the required design to support a changing signature for a specific feature selection.

We showed how the metamodel of class diagrams in RAM had to be changed to support the structural definition of signature extensions. We explained how the structural composition algorithm was changed.

Adding behavioural support for signature extension is beyond the scope for this thesis, and therefore only briefly outlined here. In fact, two different requirements are necessary for behavioural signature extension that are not solely related to signature extension but more prominently expose a lack of power in the way advices and pointcuts can currently be expressed in RAM. RAM already has support for specifying partial behaviour. In [97] the author specified a metamodel for message views (based on UML sequence diagrams) with support for partial behaviour. Aspect message views (see Section 2.5.2) are used to specify an advice to augment behaviour of partial operations. Currently, the supported pointcuts in RAM are limited. The only supported pointcut is the *execution* pointcut, i.e., the execution of an operation can be augmented with behaviour that should be executed before, after, or around the original behaviour. To properly support signature extension, calls within an execution need to be matched (*call* pointcut) such that existing calls to operations with extended signatures can be augmented to add behaviour that deals with the new parameters that are required. This could be done by adding behaviour before the call to retrieve a value, or by passing in an argument of the operation whose behaviour is being defined. In addition, it might be necessary to manipulate the return value of the original behaviour. Currently, this is not possible, and poses a problem when behaviour needs to do something with the return value.

Furthermore, the composing of behaviour then requires to match the *call* pointcuts for extended signatures and update the call with the actual arguments for additional parameters. It is also possible that additional behaviour needs to be added before the call, e.g., to retrieve or compute the argument for a parameter. In that case, this additional behaviour needs to be woven into the message view as well.

With the support for signature extension, we can now design concerns specifying a common high-level interface. To evaluate whether the approach does indeed overcome the difficulties discussed in Chapter 8 we show in the next chapter how the *Association* and *Resource Management* concerns can be re-designed and reused.

10 Case Studies

In Chapter 8 we discussed how signatures are final and how this causes problematic situations in the context of reuse. Chapter 9 proposed the *signature extension* approach to overcome the four difficulties we identified. In this chapter, we revisit the two concerns mentioned earlier to showcase the difficulties—*Association* and *Resource Management*. A fairly common validation approach in the modelling and software engineering communities are case studies. To determine whether the *signature extension* approach presented in Chapter 9 overcomes the difficulties identified in Chapter 8, we present in this chapter *prima facie* evidence by modelling the two concerns using the *signature extension* approach.

Section 10.1 presents a redesign of the *Association* concern using the signature extension approach, which makes it possible to define a common, high-level interface for adding and removing elements from alternative collection data structures. Section 10.2 revisits the *Resource Management* concern, presenting in detail the base design of resource management. Subsection 10.2.1 shows the design of the *Capability* feature which adds new functionality that requires extending some of the signatures defined in the base. Subsection 10.2.2 showcases the *Allocation Cost* feature that exemplifies how a callback interface that fits all reuse contexts can be provided. Finally, an example for delaying of decisions with non-final signatures is shown in Subsection 10.2.3. Section 10.3 concludes this chapter with a summary.

10.1 Association Concern Design with Signature Extension

In CORE, *Association* is a low-level concern that was designed to abstract from the various Java *Collection* and *Map* implementations and to provide the different variations, impacts and design models to support associations in a reusable way [16]. It represents a *concernification* of Java collections *and* maps including a significant amount of "glue code". *Association* abstracts Java's collection and map implementation classes and organizes them within a feature model.

As described in Section 8.3.1, higher-level features such as *Many* (describing at a high level

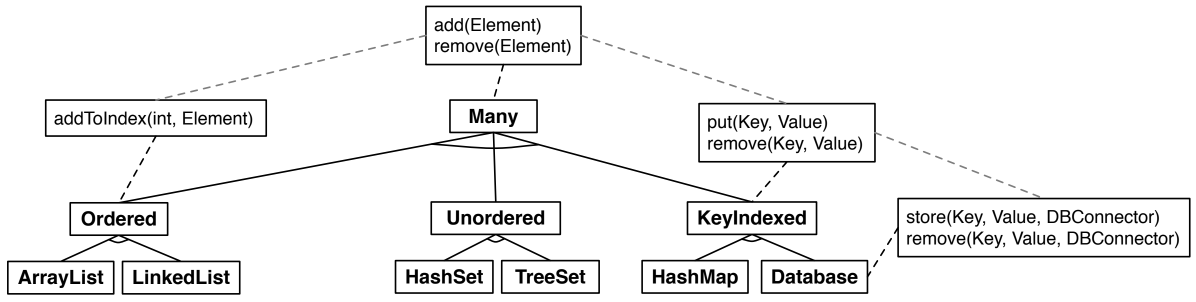


Figure 10.1: Association Signature Extensions

the common design of a *to-many* association) can not introduce a common interface for all subsequent features due to the limitation of the finality of signatures. Because of that limitation, the *Association* concern was originally designed in [16] by introducing the operations lower within the feature hierarchy. For the designer of the Association concern this makes the design activity more tedious, as common functionality needs to be repeated. For a user of the *Association* concern that is designing some other reusable concern this also has consequences. Since the reuse context of the reusable concern that is being designed is not known, it is difficult to choose the specific concrete collection class. The best choice depends on the final application context where the non-functional properties are known. Ideally, the designer would like to delay the decision, but thus far it was not possible to choose *Many*, e.g., while delaying the decision of the concrete type, and at the same time already start using the common interface of *Many*, e.g., to add elements.

We re-designed *Association* using the signature extension support. Figure 10.1 shows the *Many* branch of the *Association* feature model. Two operations are now first introduced in the realization design model of the *Many* feature defining a common signature for adding and removing elements: `add(Element)` and `remove(Element)`.

KeyIndexed then extends the signature of the `add` and `remove` operations to add a parameter for the key. This means that the `Element` in *KeyIndexed* represents the value. Furthermore, the *Database* feature further extends the two operations to add a third parameter for the database connection.

The realization model of the *Ordered* feature extends the signature of the `add` operation to add a parameter for the index at which to add the element to.

Listing 10.1 shows the mappings of *KeyIndexed, Database*, and *Ordered* for the `add` operation and the repeated parameters. I.e., the left-hand side of the mappings shows the structural element from the extended model, the right-hand side shows the element of the extending model. The mappings reflect the hierarchy of their elements by indentation, i.e., class, operation, and then parameter.

Listing 10.1 Signature Extension Mappings of the `add` Operation

```
Many is extended by KeyIndexed
  add(Element element) ⟶ put(Key key, Value value)
    Element element ⟶ Value value

KeyIndexed is extended by Database
  put(Key key, Value value) ⟶
  store(Key key, Value value, DBConnector db)
    Key key ⟶ Key key
    Value value ⟶ Value value

Many is extended by Ordered
  add(Element element) ⟶ addToIndex(int i, Element element)
    Element element ⟶ Element element
```

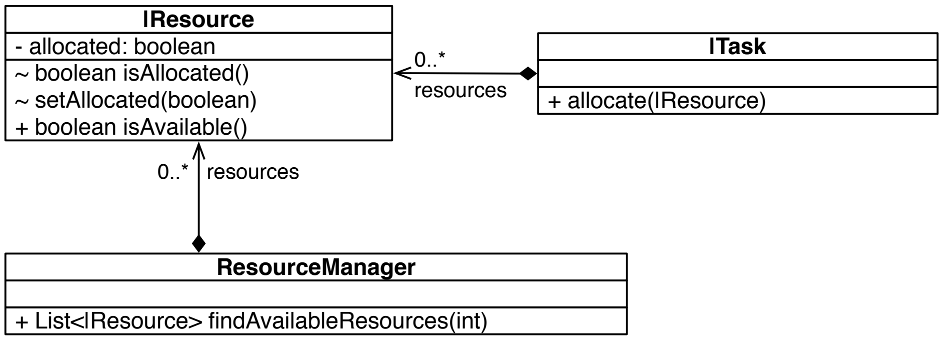


Figure 10.2: Resource Management Base Design

We will show the reuse of this *Association* concern within the *Resource Management* concern in the following section to show how delaying of decisions is supported by signature extension.

10.2 Resource Management: Running Example Revisited

In Section 8.3 we discussed the difficult situations due to the finality of signature declarations using the example of a *Resource Management*. Figure 10.2 shows the base structural design of the Resource Management concern. The base model provides the ability to have tasks that can allocate resources, and a resource manager to manage all resources. To aid the reader in understanding of the signature extensions that follow, we also show the behaviour related to the base design. This provides a better understanding of the interactions between the different classes.

Figure 10.3 specifies the behaviour of checking whether a resource is available. A resource is available if it is currently not allocated.

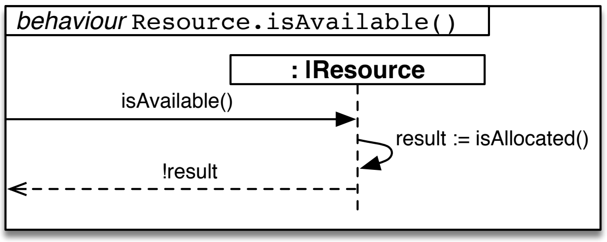


Figure 10.3: Behaviour for Checking the Availability of a Resource

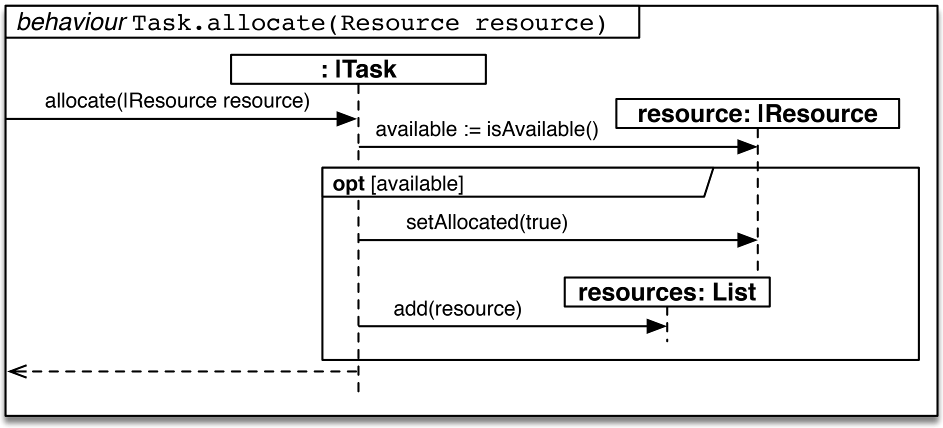

behaviour Task.allocate(Resource resource)
: ITask
allocate(IResource resource)
resource: IResource
available := isAvailable()
opt [available]
setAllocated(true)
resources: List
add(resource)

Figure 10.4: Behaviour for Allocating a Resource to a Task

When a resource is to be allocated to a task, the resource can not already be allocated. Figure 10.4 shows the behaviour of `Task.allocate(Resource)` which invokes the `isAvailable()` operation on the given resource. If the resource is available, it is marked as allocated and associated with the task.

The `ResourceManager` manages all resources and provides an operation to find a given number of available resources. Figure 10.5 shows the corresponding behaviour. The `ResourceManager` loops through all resources and invokes `isAvailable()` on each resource that it considers until it finds the required number of resources (or reaches the end of the collection).

10.2.1 Capability Design Extension

The designer intends to provide an optional feature that allows users to differentiate resources based on capabilities. In order to support this, the desired capability for which resources are sought for, or need to be allocated, has to be specified. I.e., some of the operation signatures of the base design of *Resource Management* need an additional parameter to identify this capability. Because the designer is aware of the internals of the base design, she can directly extend the signatures.

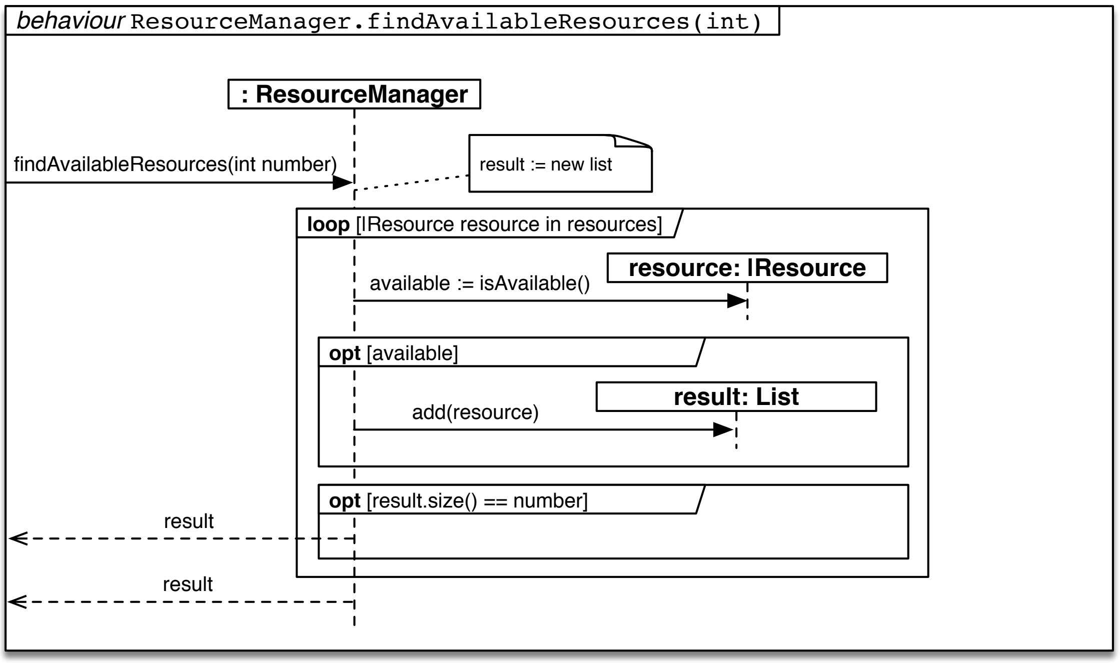

behaviour ResourceManager.findAvailableResources(int)
: ResourceManager
findAvailableResources(int number)
result := new list
loop [IResource resource in resources]
available := isAvailable()
resource: IResource
opt [available]
add(resource)
result: List
opt [result.size() == number]
result
result

Figure 10.5: Behaviour for Finding Available Resources

The realization model for the *Capability* feature extends resources with capabilities. It allows a resource to be allocated to a task based on a specific capability. When a user chooses this variant, several of the existing operation signatures defined for general *Resource Management* are extended. This is because `allocate` needs an additional parameter identifying the capability. In turn, `allocate` calls `isAvailable` on the resource. Figure 10.6 shows the structural design of *Capability* with the additional operations.

In addition to defining the operations, the appropriate mappings need to be provided. Listing 10.2 shows the mappings specified for the *Capability* feature extending the design of the base *Resource Management*.

10.2.2 Allocation Cost Design Extension

The *Allocation Cost* feature offers the ability to consider individual resource allocation cost. To accomplish this, it adds the `estimateTotalAllocationCost` operation to the `Resource-Manager` which can determine the total cost of allocation for the given list of resources. Since the actual resources and their properties are unknown for the designer of the *Resource Management* concern, a partial operation `estimateAllocationCost` is introduced. This operation needs to be provided by the user for each different resource. Figure 10.7 shows the structural design of

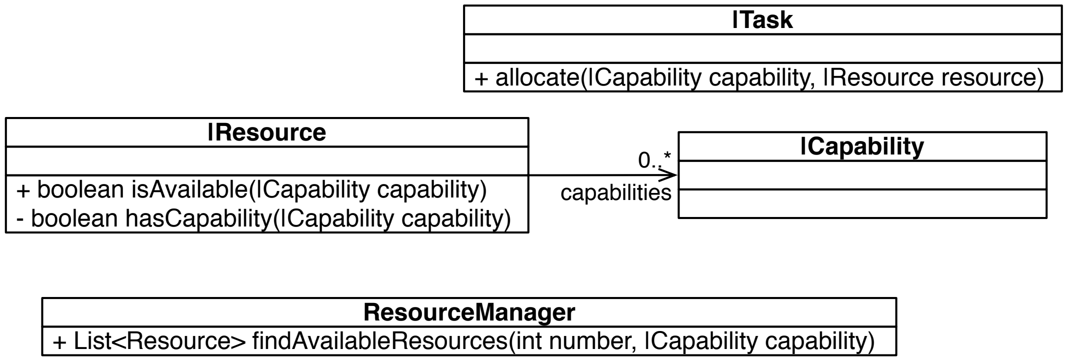

ITask
+ allocate(ICapability capability, IResource resource)
IResource
+ boolean isAvailable(ICapability capability)
- boolean hasCapability(ICapability capability)
0..*
capabilities
ICapability
ResourceManager
+ List<Resource> findAvailableResources(int number, ICapability capability)

Figure 10.6: *Capability* Structural Design

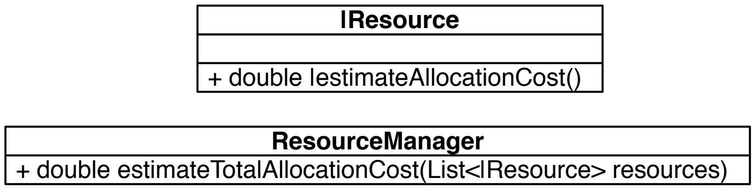

IResource
+ double IestimateAllocationCost()
ResourceManager
+ double estimateTotalAllocationCost(List<IResource> resources)

Figure 10.7: *Allocation Cost* Structural Design

Allocation Cost.

Figure 10.8 provides the details of the behaviour for estimating the total allocation cost. The behaviour iterates through all the given resources and invokes the callback operation on each resource. However, the allocation cost of a resource might depend on the specific context the resource is used in. For example, in the context of a *Crisis Management System* (CMS), workers (the resources) are allocated to missions (the tasks). Determining the allocation cost of a specific worker to a mission depends on the distance between the worker's current location and the location of the mission. As we described in Section 8.3.3, while the current location of the worker is accessible from within the resource, the location of the mission is not. To provide a flexible design that supports such use cases, the designer of *Resource Management* can prepare for the extension of the signature of the `estimateAllocationCost` callback with additional parameters.

Figure 10.9 shows the updated structural design for the *Allocation Cost* feature. The designer introduced a partial parameter in the `Resource.estimateAllocationCost` operation with a mapping cardinality multiplicity of `0..*`. Because `ResourceManager.estimateTotal-AllocationCost` calls this method, it also needs to be extended with a partial parameter whose mapping cardinality references that of the callback method. Any additional parameter passed into `estimateTotalAllocationCost` can then simply be forwarded to `estimateAlloca-tionCost`. This is sufficient in this case because the user invokes the `estimateTotalAllo-`

Listing 10.2 Mappings of the *Capability* Feature

```
|Resource ⟶ |Resource
  isAvailable() ⟶ isAvailable(Capability)

|Task ⟶ |Task
  allocate(|Resource resource) ⟶
  allocate(|Capability capability, |Resource resource)
    resource ⟶ resource
  add(|Resource resource) ⟶
  add(|Capability capability, |Resource resource)
    resource ⟶ resource

ResourceManager ⟶ ResourceManager
  findAvailableResources(int number) ⟶
  findAvailableResources(int number, |Capability capability)
    number ⟶ number
```

`cationCost` operation, which in turn invokes the callback.

10.2.3 Delaying of Decisions

In order to realize the relationship between tasks and resources, the designer can reuse the *Association* concern. Since *Resource Management* is a reusable concern itself, the designer cannot determine the most appropriate variant of *Association* to reuse. This decision depends on the context in which *Resource Management* will be reused. Therefore, thanks to the changes described in Section 10.1, the designer can now choose *Many*, the feature that provides a common, high-level interface for adding and removing elements. The child features below *Many* are re-exposed. With this, the designer can already specify the base design, including the behaviour, by making use of the `add` operation when allocating a resource to a task, for example.

The *Capability* feature, however, requires resources to be allocated for a specific capability. Therefore, the *Capability* design model extends the reuse of *Association* and augments the feature selection by choosing *KeyIndexed*. The specific child features of *KeyIndexed* stay re-exposed, as the specific data structure to use affects non-functional properties which depends on the reuse context. This changes the signature of `add` to require an additional parameter for the key. Section 10.2.1 described how the designer delegated this parameter to the callers by extending all affected operations with relevant invocations.

However, the signature of `add` is still not final. If a user of *Resource Management* who chooses the *Capability* feature also selects the re-exposed *Database* feature as the storage structure to keep

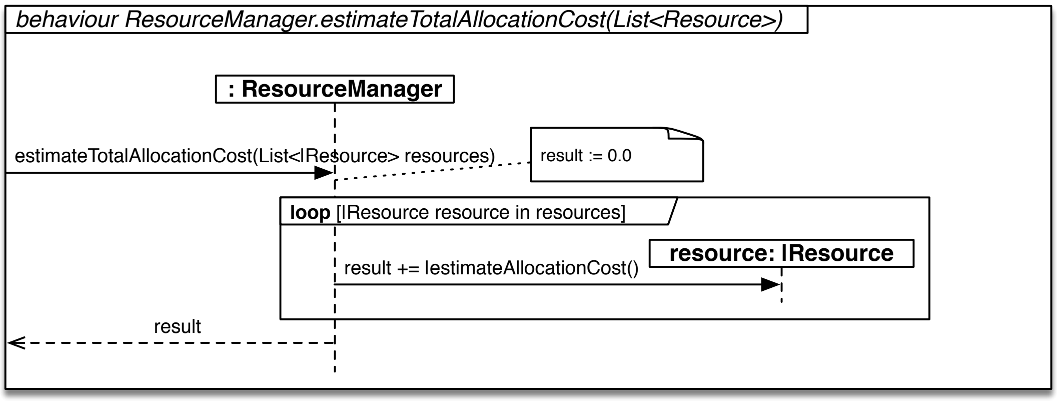

behaviour ResourceManager.estimateTotalAllocationCost(List<Resource>)
: ResourceManager
estimateTotalAllocationCost(List<IResource> resources)
result := 0.0
loop [IResource resource in resources]
resource: IResource
result += IestimateAllocationCost()
result

Figure 10.8: Behaviour for Finding Available Resources

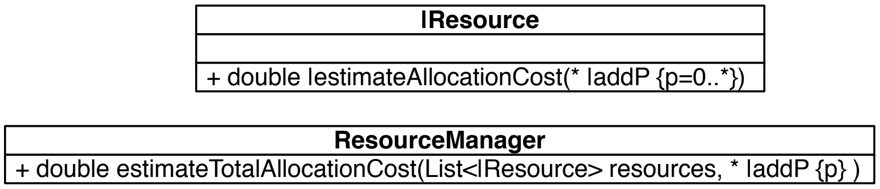

IResource
+ double IestimateAllocationCost(* IaddP {p=0..*})
ResourceManager
+ double estimateTotalAllocationCost(List<IResource> resources, * IaddP {p})

Figure 10.9: Updated *Allocation Cost* Structural Design

track of the task-resource mapping, then the signature of `add` requires an additional third database connector parameter. As discussed in Section 9.2, the designer of *Resource Management* has the knowledge of the internals of this concern and as such knows best how to support this change in the signature. The designer could choose to not support *Database* by not re-exposing it. Alternatively, the designer can provide a design model that deals with the re-exposed *Database* feature. For example, an additional optional child feature of *Capability* could offer the database feature. The designer can then choose to use the same implementation technique that she used for adding capabilities by adding and passing along the database connector parameter. Alternatively she could also decide to offer an operation that sets the required *DBConnection* parameter once during initialization, for instance, as an operation offered by the *ResourceManager*, and then use that value whenever it is needed in calls to the *Association* concern.

Figure 10.10 shows an example of a structural design for the database support. The signature of the constructor of `ResourceManager` is extended with an additional parameter requiring the `DBConnection` instance to be provided. The behaviour of `Task.allocate(...)` can thus be augmented such that the connection is retrieved from the `ResourceManager` and passed into the call to `add`.

Listing 10.3 provides the mappings between *Capability* and *Database Support*.

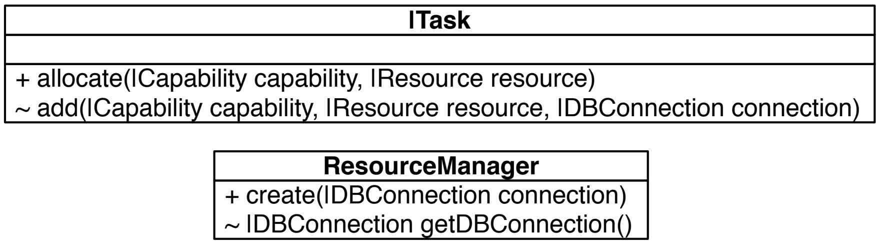

ITask
+ allocate(ICapability capability, IResource resource)
~ add(ICapability capability, IResource resource, IDBConnection connection)
ResourceManager
+ create(IDBConnection connection)
~ IDBConnection getDBConnection()

Figure 10.10: Optional *Database Support* Structural Design

Listing 10.3 Mappings of the Optional *Database Support* Feature

```
ResourceManager ⟶ ResourceManager
  create() ⟶ create(|DBConnection)

Task ⟶ Task
  add(|Capability, |Resource) ⟶
  add(|Capability, |Resource, |DBConnection)
    capability ⟶ capability
    resource ⟶ resource
```

10.2.4 Composed Interface

With the updated design of *Resource Management* using *signature extension* support, the usage interface of the concern now is always perfectly in line with the features that were selected. Figure 10.11 shows the interface and supporting design structure of *Resource Management* of different feature selections: *Base*, *Base* + *Capability*, *Base* + *Cost*, *Base* + *Capability* + *Cost*, and *Base* + *Capability* + *Database*.

10.3 Summary

In this chapter we revisited the examples that exhibited the four difficult situations identified in Chapter 8. We showed that these difficulties can be overcome with the proposed signature extension approach. The designer as the expert of a reusable concern with detailed knowledge of the inner workings can define a common interface for alternative implementations at a high level and extend the common interface for lower level features with additional parameters if necessary. The designer is also able to add new functionality that requires additional parameters in signatures. This ensures that when a user wants to use the additional functionality, the usage interface of the reusable concern includes the required parameters and hence ensures correct use. In order to allow a user to have a custom callback interface for any user-specific reuse context, the designer can prepare for this by making the signature extendable using partial parameters. Furthermore, to deal

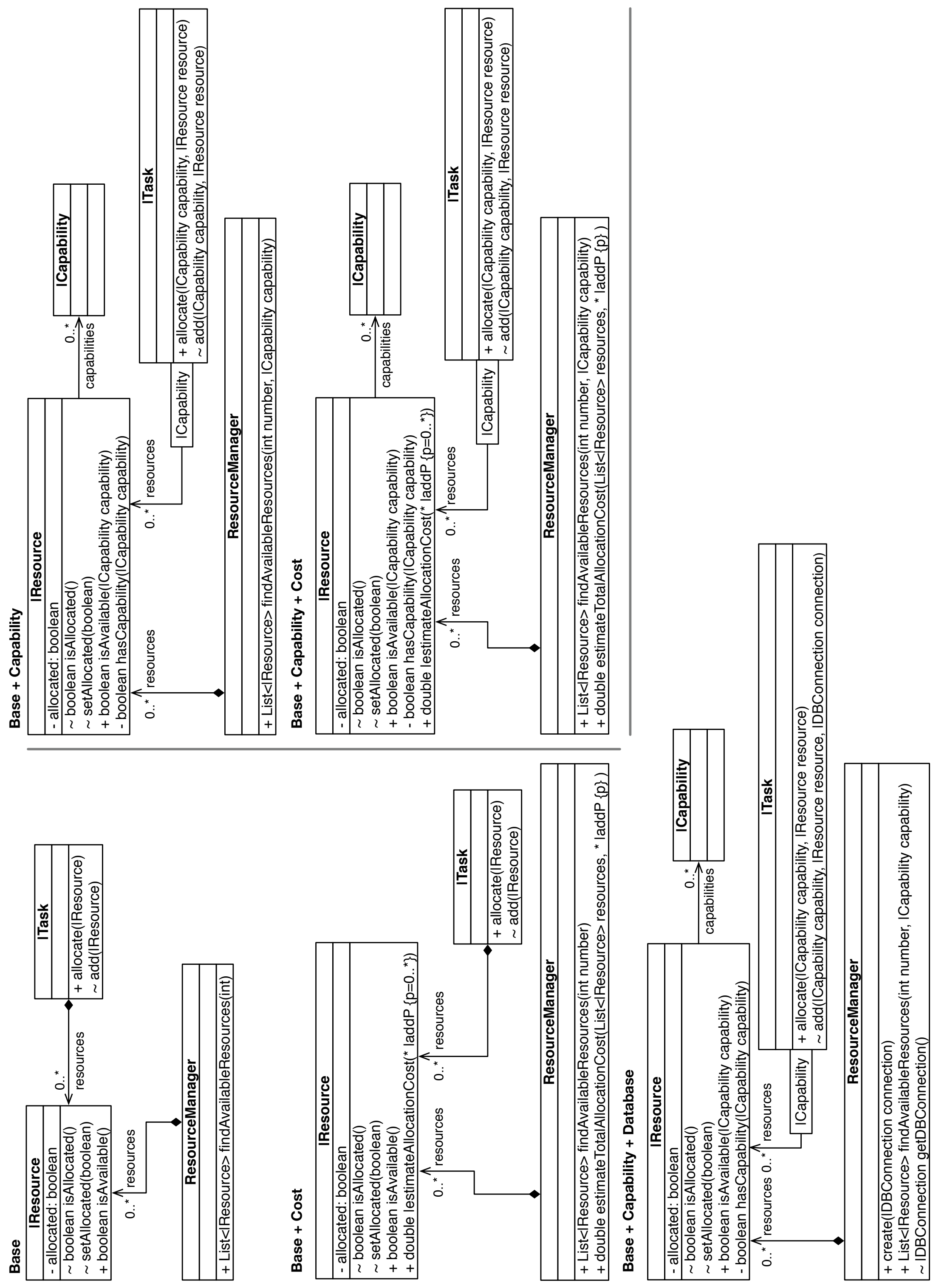

Base
IResource
- allocated: boolean
~ boolean isAllocated()
~ setAllocated(boolean)
+ boolean isAvailable()
ITask
+ allocate(IResource)
~ add(IResource)
resources
0..*
ResourceManager
+ List<IResource> findAvailableResources(int)
Base + Cost
IResource
- allocated: boolean
~ boolean isAllocated()
~ setAllocated(boolean)
+ boolean isAvailable()
+ double IestimateAllocationCost(* IaddP {p=0..*})
ITask
+ allocate(IResource)
~ add(IResource)
ResourceManager
+ List<IResource> findAvailableResources(int number)
+ double estimateTotalAllocationCost(List<IResource> resources, * IaddP {p})
Base + Capability + Database
IResource
- allocated: boolean
~ boolean isAllocated()
~ setAllocated(boolean)
+ boolean isAvailable(ICapability capability)
- boolean hasCapability(ICapability capability)
capabilities
ICapability
ITask
+ allocate(ICapability capability, IResource resource)
~ add(ICapability capability, IResource resource, IDBConnection connection)
ResourceManager
+ create(IDBConnection connection)
+ List<IResource> findAvailableResources(int number, ICapability capability)
~ IDBConnection getDBConnection()
Base + Capability
IResource
- allocated: boolean
~ boolean isAllocated()
~ setAllocated(boolean)
+ boolean isAvailable(ICapability capability)
- boolean hasCapability(ICapability capability)
ICapability
ITask
+ allocate(ICapability capability, IResource resource)
~ add(ICapability capability, IResource resource)
ResourceManager
+ List<IResource> findAvailableResources(int number, ICapability capability)
Base + Capability + Cost
IResource
- allocated: boolean
~ boolean isAllocated()
~ setAllocated(boolean)
+ boolean isAvailable(ICapability capability)
- boolean hasCapability(ICapability capability)
+ double IestimateAllocationCost(* IaddP {p=0..*})
ICapability
ITask
+ allocate(ICapability capability, IResource resource)
~ add(ICapability capability, IResource resource)
ResourceManager
+ List<IResource> findAvailableResources(int number, ICapability capability)
+ double estimateTotalAllocationCost(List<IResource> resources, * IaddP {p})

Figure 10.11: Composed Structure of *Resource Management* with Different Feature Selections

with re-exposed features when delaying decisions, the designer needs to prepare any design for re-exposed features that affect the signatures. The user on the other hand does not need to deal with the implementation details of the reusable concern and can extend signatures where prepared by the designer. As such, information hiding is maintained according to the philosophy of CORE.

Part IV

Epilogue

11 Conclusions & Future Work

This chapter first summarizes the contributions of this thesis in Section 11.1, followed by a discussion of potential future research avenues in Section 11.2.

11.1 Conclusions

Methodical reuse of software artefacts is considered key to software engineering [65, 70]. Instead of creating all functionality from scratch, common and recurring functionality is reused. While there are many frameworks available to be reused at the implementation level, at the modelling level reuse is not common, which puts modelling at a disadvantage. Furthermore, reuse at the implementation level is not perfect. Frameworks provide many features in *one piece* (monolithic code block) and developers spend a lot of time trying to understand how to use and customize a framework. In fact, when we posted the invitation for participation for the user study with Android Notifications some developers confirmed this. Some of the mentioned obstacles were poor documentation, not clear how to use the API for more advanced scenarios, and that the API was not working as expected. The last point was especially mentioned in terms of maintaining compatibility, i.e., when trying to support features on as many platform versions as possible. Some developers also mentioned that they use an additional framework on top of Android Notifications that aims at making it easier to create and configure notifications. However, this creates an additional layer of complexity.

This thesis makes several important contributions towards bridging the gap between the modelling and the programming worlds. Thanks to our proposed *concernification* and *signature extension* approaches, existing functionality encapsulated in libraries and frameworks can not only be reused at the modelling level, but our proposed, high-level interfaces for frameworks also benefit the reuse at the implementation level.

Concernification This thesis presented the *Concernification* approach that applies the principles of Concern-Oriented Reuse (CORE) to raise the level of abstraction of existing frameworks to the modelling level. At the modelling level the benefits of the higher level of abstraction can be exploited to benefit reuse. A concern interface for a framework formalizes and documents what the framework provides to the user. This includes the different features and their relationships. Furthermore, the API of the framework is decomposed and the API elements are modularized according to the features they belong to, effectively reducing the API to the user's needs. In addition, the concern interface makes it possible to specify the API elements that need to be customized by the user, and can contain repetitive "glue code" that is necessary to add to the application in which the framework is used. We created a concern interface for the *Minueto* framework with the *concernification* steps we defined. Using the resulting concern interface we showed that for a minimal feature selection, only 26% of the API elements are exposed to the user. For a newcomer who wants to write a *hello world* application, this is a significant reduction in cognitive effort.

Creating a concern interface manually represents a high effort. This thesis proposed an automated concernification algorithm to help a developer with the initial concern interface creation. As a preparation for our work on an algorithm to automatically create a concern interface, we conducted a qualitative study with the two developers of *Minueto*. As the domain experts of the framework, they helped us in validating our feature model of *Minueto*. Additionally, they created their own feature model of *Minueto*. This confirmed that there is no one single correct feature model. It depends on the desired granularity and what one considers a feature. The Minueto developers also confirmed our intuition of which information of an API to use when determining a concern interface.

Automated Concernification To support developers in *concernifying* their framework we designed an algorithm that automatically creates an initial concern interface, which can then be fine-tuned and adjusted by the developer who is the domain expert. The algorithm exploits the object-oriented hierarchy of the framework implementation, the cross-references among the API elements and the way classes are organized in packages and inner classes in an API. In addition, the algorithm examines the use of the API in code examples that are commonly provided with a framework. These examples typically show how to use certain features/accomplish a task.

Our algorithm exploits common principles of good API design that we always expect to hold. We additionally defined hypotheses about the API and the code examples that might not always hold, but that can improve the result of our algorithm if they do. For example, the code examples do not always showcase all possible combination of features. If not all combinations are shown, XOR relationships can not be determined. However, if all possible combinations are shown in the examples, our algorithm is designed to perform additional processing steps which in this case make it possible to generate the OR, XOR and cross-tree constraints of the feature model with 100% accuracy. The algorithm works with a directed acyclic graph (DAG) where the nodes represent potential features and the edges relationships between them. The example usage is used to put those API elements together that are used in the same examples. Once the graph is populated with all the available information, it is simplified in order to retrieve a tree that represents the feature model. As a result, the automated concernification algorithm produces a concern interface consisting of the feature model and a design model for each feature containing the corresponding API. To be able to perform automated concernification on larger frameworks we fully implemented the algorithm on top of the backend components of the TouchCORE tool. In order to evaluate whether the algorithm provides an accurate concern interface, we validated the algorithm by applying it to three frameworks and analyzed the results. The conciseness of results is dependent on the quality of examples in order to determine which API elements belong to the same feature. More framework APIs need to be concernified to gather evidence on how the conciseness can be improved. A limitation of the algorithm is that it currently lacks support for other artefacts and code elements that are typically used in frameworks, especially those frameworks making use of dependency injection. For example, many frameworks make use of XML configurations or annotations. Our algorithm currently does not consider these and hence does not detect usage of framework API in such a case. Furthermore, the algorithm currently works only on Java. However, the algorithm itself is designed such that it is independent of a specific language. Other programming languages could be supported by providing an importer and example parser for that language.

Signature Extension The higher level of abstraction revealed a lack of support for the incremental refinement of interfaces. By separating an API across features, there are difficult situations—such as adding optional functionality—that require a signature of a method to be extended with additional parameters. We identified and discussed in detail four difficult situations that the finality of method signatures causes. We performed an empirical study on the base module of the Java Platform API and investigated workarounds in programming languages. We found evidence of these difficulties at the programming level. To overcome these problems at the modelling level, we presented the *signature extension* approach. The approach makes it possible to extend the signature of methods. We extended class diagrams as defined in the Reusable Aspect Models (RAM) language of CORE with structural signature extension support. This extension allows a signature to be incrementally refined based on the features that a user selected. To evaluate whether the signature extension approach overcomes the identified difficulties, we re-designed two concerns that are affected by these difficulties. Due to the use of features and aspect-oriented techniques, it would be difficult to integrate this approach at the programming level. However, the concernification approach provides a higher level view that helps a user understand what a framework provides. The support of flexible callback signatures is limited in that the designer needs to prepare for the extension of callback signatures in the design. This limitation is necessary to maintain the information hiding principles.

Summary Overall, the contributions in this thesis help close the gap between the implementation and modelling levels. First, by raising frameworks to the higher level of abstraction that the modelling level provides. Second, by exploiting this higher level of abstraction. This can help the user in understanding and using a framework and avoid making mistakes. Third, by providing a solution for the rigid nature of signatures that is present at the implementation level. This therefore provides a bridge back to the implementation level.

11.2 Future Work

The work presented in this thesis opens up many potential future research avenues. We present some of these in the following paragraphs.

Behavioural Support for Signature Extension The integration of signature extension into the class diagrams of the RAM language in CORE currently lacks support for the behavioural specification on how to deal with additional parameters. The expression of pointcuts to advise existing behaviour is limited in its expressibility. Additional support is required to match calls and use the return value. Furthermore, the composition algorithm for sequence diagrams needs to be extended.

Mining Other Framework Artefacts Our Automated Concernification algorithm currently does not determine the usage protocol of the API and impacts of features to non-functional goals. Furthermore, names of features are not identified. For this, other framework artefacts, such as textual documentation and the API reference could be mined by using information retrieval or NLP techniques. Furthermore, besides the API reference, the source code can contain additional information, such as which exceptions might be thrown at runtime and under which circumstances. In the case of usage protocols, there exist approaches to mine the client code of frameworks to determine usage patterns (e.g., [95]). These could potentially be helpful in identifying usage protocols of the API. For some non-functional goals it is necessary to perform benchmarks in order to determine appropriate values on how a feature contributes to a certain goal. For example, the performance of different collection implementations can be measured using benchmarks [16].

Using the Automated Concernification Algorithm to Improve Framework Example Quality As we identified in the validation of the automated concernification algorithm in Chapter 6, the performance of the algorithm depends on the quality of the code example use scenarios. This fact can be exploited: the developer of the framework can use the result of the algorithm to judge the quality of the code examples.

Increase Concernification Algorithm Conciseness Our algorithm currently uses static analysis on the examples to determine API usage. Examples sometimes don't actually invoke certain code or show different usage scenarios. Dynamic analysis could be used to detect this. Furthermore, information retrieval techniques could be used to determine the similarity of API elements and exploit this information to group API elements. This could particularly be useful if there are lots of API elements that have not been used in any examples. Alternatively, unused elements could be collected and afterwards presented to the developer. The framework designer is the domain expert who can then assign these elements to features and fine-tune the result. In general, a tool that allows a framework designer to quickly and intuitively adjust the automatically determined concern interface is essential. The interactive website of our user study on the Android Notifications API provides an easy way to browse the API for features and could be adapted to allow modifications to the feature model and API.

Training Tool for Learning a Framework One of the developers of the Minueto framework in Chapter 4 suggested that our concernification approach be used as a training tool to learn a framework. In industry, besides third-party frameworks, companies often develop their own in-house frameworks. When new developers join the company, the expectation is to get familiar with such frameworks quickly. Companies often invest heavily in training for their employees. A training tool could allow new developers to browse the different framework features and their relevant artefacts, and allow them to choose features to retrieve the relevant API and, as discussed above, other parts of artefacts. In addition, the training tool could tell the user which examples to look at based on the feature selection, or even generate specific examples that showcase exactly the selected features. The availability of such a training tool would then make it possible to perform more in-depth user studies to evaluate whether 1) the high level view provided by concernification helps developers in understanding a framework better and 2) concernification helps improve reuse compared to traditional approaches. Here, the time in which a task is completed is of interest, but also whether concernification helps new developers to avoid/reduce making mistakes when reusing a new framework.

Concern Interface Artefact at Implementation Level To go even further, we see a great potential in making the concern interface an integral part of implementation. If the concern interface is packaged along with a framework, it could assist developers in the reuse process. Nowadays, build systems (such as Maven, Gradle, etc.) are used to define dependencies. For instance, the concern interface could also be used to generate the required dependency configuration based on the features the user chose to use. This can be helpful if a certain feature has a dependency to another framework or a framework is modularized into several artefacts. Furthermore, the artefacts that build systems include in the compilation process usually comprise the complete framework. For very large frameworks or platforms, a current trend is to *unbundle* them into smaller pieces, i.e., an existing bundle is taken and divided into smaller ones. For instance, with Java 9 the Java Platform was split into modules. Also, *Spring Boot* provides starter packs for common uses for the *Spring* framework and a web configurator[1] to choose the desired feature set. However, these modules/bundles tend to still be large and each contain themselves a large feature set. This is where we see a great potential for concernification to allow providing a more fine-grained bundling mechanism. An interesting avenue of research would be to investigate the feasibility of specifying the desired features within the dependency management of build systems. This could facilitate many additional benefits at the implementation level, such as feature-specific code completion, verifying the usage protocol, ensuring correct customization, etc. In addition, a possible direction to investigate is whether signature extension can be integrated at the implementation level as well to benefit users there.

Concernifying Other Framework Artefacts Currently, the *Concernification* approach concernifies the API of a framework and hence makes it possible to tailor the API to the user's needs. There is also supplementary information relevant to the API, such as textual documentation. Often, textual documentation consists of many pages with sections and hyperlinks between them. For example, the guides on notifications of Android consist of several pages with a large number of sections. While the sectioning helps, it is not necessarily clear which feature the documentation is referring to. Furthermore, the parts of interest might be spread over several pages. Ideally, such other artefacts of a framework would be *concernified* as well. Then, based on a feature selection of the user, those artefacts could be composed to present to the user a tailored version of the artefacts containing only what is relevant to those features. Besides the textual documentation, this could also be done for the example code. For instance, depending on the feature selection, the user could receive a suggestion on which examples to look at. Ideally, an example could be generated specifically for a specific feature selection showcasing exactly how to use the selected features.

[1]https://start.spring.io

Bibliography

[1] Wisam Al Abed, Valentin Bonnet, Matthias Schöttle, Engin Yildirim, Omar Alam, and Jörg Kienzle. TouchRAM: A Multitouch-Enabled Tool for Aspect-Oriented Software Design. In *SLE 2012*, pages 275–285. Springer, 2012.

[2] Wisam Al Abed, Matthias Schöttle, Abir Ayed, and Jörg Kienzle. *Behavior Modeling – Foundations and Applications: International Workshops, BM-FA 2009-2014, Revised Selected Papers*, chapter Concern-Oriented Behaviour Modelling with Sequence Diagrams and Protocol Models, pages 250–278. Springer International Publishing, Cham, 2015.

[3] R. Al-msie'deen, A. D. Seriai, M. Huchard, C. Urtado, and S. Vauttier. Mining Features from the Object-Oriented Source Code of Software Variants by Combining Lexical and Structural Similarity. In *2013 IEEE 14th International Conference on Information Reuse Integration (IRI)*, pages 586–593, Aug 2013.

[4] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Concern-Oriented Software Design. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, editors, *Model-Driven Engineering Languages and Systems (MODELS)*, pages 604–621, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[5] Romain Alexandre, Cécile Camillieri, Mustafa Berk Duran, Aldo Navea Pina, Matthias Schöttle, Jörg Kienzle, and Gunter Mussbacher. Support for Evaluation of Impact Models in Reuse Hierarchies with jUCMNav and TouchCORE. In *Proceedings of the MODELS 2015 Demo and Poster Session co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, Canada*, volume 1554 of *CEUR-WS*, pages 28–31, 2015.

[6] Android Open Source Project. Android Developers. `https://developer.android.com`, 2019.

[7] Android Open Source Project. Android Developers Guides: Build messaging apps for Android Auto. `https://developer.android.com/training/cars/messaging`, 2019.

[8] Android Open Source Project. Android Developers Guides: Create a Notification. `https://developer.android.com/training/notify-user/build-notification`, 2019.

[9] Android Open Source Project. Android Developers Guides: Create a Notification on Wear OS. `https://developer.android.com/training/wearables/notifications/creating`, 2019.

[10] Android Open Source Project. Android Developers Guides: Create and Manage Notification Channels. `https://developer.android.com/training/notify-user/channels.html`, 2019.

[11] Android Open Source Project. Android Developers Guides: Notifications Overview. `https://developer.android.com/guide/topics/ui/notifiers/notifications.html`, 2019.

[12] Android Open Source Project. Android Platform Libraries: Support Library. `https://developer.android.com/topic/libraries/support-library`, 2019.

[13] Michal Antkiewicz. *Framework-Specific Modeling Languages*. Ph.D. dissertation, University of Waterloo, Waterloo, ON, Canada, 2008.

[14] Vallabh Anwikar, Ravindra Naik, Adnan Contractor, and Hemanth Makkapati. Domain-driven Technique for Functionality Identification in Source Code. *SIGSOFT Software Engineering Notes*, 37(3):1–8, May 2012.

[15] Wesley K. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. Reengineering Legacy Applications into Software Product Lines: A Systematic Mapping. *Empirical Software Engineering*, 22(6):2972–3016, December 2017.

[16] Céline Bensoussan, Matthias Schöttle, and Jörg Kienzle. Associations in MDE: A Concern-Oriented, Reusable Solution. In *Modelling Foundations and Applications - 12th European Conference, ECMFA 2016, Held as Part of STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings*, pages 121–137. Springer International Publishing, 2016.

[17] Sunit Bhalotia and Jörg Kienzle. Reusable Model Interfaces with Instantiation Cardinalities. In *Modelling Foundations and Applications (ECMFA 2015)*, volume 9153 of *Lecture Notes in Computer Science*, pages 108–124, Cham, Switzerland, July 2015. Springer International Publishing.

[18] Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. Clafer: Unifying Class and Feature Modeling. *Softw. Syst. Model.*, 15(3):811–845, July 2016.

[19] Joshua Bloch. How to Design a Good API and Why It Matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 506–507, New York, NY, USA, 2006. ACM.

[20] Grady Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition).* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[21] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice: Second Edition.* Morgan & Claypool Publishers, 2nd edition, 2017.

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

[23] Kunrong Chen and Václav Rajlich. Case Study of Feature Location Using Dependence Graph. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, pages 241–247, June 2000.

[24] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Using Feature Models. In *Software Product Lines: Third International Conference, SPLC 2004*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283, Heidelberg, Germany, 2004. Springer Berlin / Heidelberg.

[25] Krzysztof Czarnecki and Andrzej Wąsowski. Feature Diagrams and Logics: There and Back Again. In *11th International Software Product Line Conference (SPLC 2007)*, pages 23–34. IEEE Computer Society, Sept 2007.

[26] Robertas Damaševičius, Paulius Paškevičius, Eimutis Karčiauskas, and Romas Marcinkevičius. Automatic Extraction of Features and Generation of Feature Models from Java Programs. *Information Technology and Control*, 41(4):376–384, 2012.

[27] Alexandre Denault and Jörg Kienzle. Minueto, a Game Development Framework for Teaching Object-Oriented Software Design Techniques. In *FuturePlay 2006*, 2006.

[28] Edsger Wybe Dijkstra. *A discipline of programming*, volume 1. Prentice-Hall Englewood Cliffs, 1976.

[29] Edsger Wybe Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, New York, USA, 1982.

[30] Peter Dimov. Boost.Bind Documentation: Chapter 1. Boost.Bind. `https://www.boost.org/doc/libs/1_69_0/libs/bind/doc/html/bind.html`, 2018.

[31] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[32] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39:41–47, 2006.

[33] Ekwa Duala-Ekoko and Martin P. Robillard. Asking and Answering Questions About Unfamiliar APIs: An Exploratory Study. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 266–276, Piscataway, NJ, USA, 2012. IEEE Press.

[34] Mustafa Berk Duran, Gunter Mussbacher, Nishanth Thimmegowda, and Jörg Kienzle. On the Reuse of Goal Models. In *Proceedings of the 17th International System Design Languages Forum (SDL 2015), Berlin, Germany, October 2015*, volume 9369 of *Lecture Notes in Computer Science*, pages 141–158, Cham, Switzerland, 2015. Springer International Publishing. SDL 2015: Model-Driven Engineering for Smart Cities.

[35] Lucas Eder. Defensive API Evolution With Java Interfaces. `https://dzone.com/articles/defensive-api-evolution-java`, 2013.

[36] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating Features in Source Code. *IEEE Transactions on Software Engineering*, 29(3):210–224, March 2003.

[37] Andrew David Eisenberg and Kris De Volder. Dynamic Feature Traces: Finding Features in Unfamiliar Code. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 337–346, Sep. 2005.

[38] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing Aspects of AOP. *Commun. ACM*, 44(10):33–38, October 2001.

[39] George Fairbanks. *Design Fragments*. Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 2007.

[40] George Fairbanks, David Garlan, and William Scherlis. Design Fragments Make Using Frameworks Easier. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on*

Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06, pages 75–88, New York, NY, USA, 2006. ACM.

[41] Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit. *Aspect-Oriented Software Development*. Addison-Wesley Professional, First edition, 2004.

[42] Eclipse Foundation. Eclipse Java development tools (JDT). `https://www.eclipse.org/jdt/`.

[43] Eclipse Foundation. Acceleo. `https://www.eclipse.org/acceleo/`, 2019.

[44] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[45] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM.

[46] Brian Goetz. Interface evolution via virtual extension methods. `http://cr.openjdk.java.net/~briangoetz/lambda/Defender%20Methods%20v4.pdf`, June 2011.

[47] Emmanuel Henry and Benoît Faller. Large-Scale Industrial Reuse to Reduce Cost and Cycle Time. *IEEE Software*, 12(5):47–53, Sep. 1995.

[48] Daqing Hou, Kenny Wong, and H. James Hoover. What Can Programmer Questions Tell Us About Frameworks? In *Proceedings of the 13th International Workshop on Program Comprehension*, IWPC '05, pages 87–96, Washington, DC, USA, 2005. IEEE Computer Society.

[49] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven Engineering Practices in Industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 633–642, New York, NY, USA, 2011. ACM.

[50] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical Assessment of MDE in Industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.

[51] IBM Corporation. Evolving Java-based APIs 2. `http://wiki.eclipse.org/Evolving_Java-based_APIs_2`, 2017.

[52] IBM Corporation. Evolving Java-based APIs 3. `http://wiki.eclipse.org/Evolving_Java-based_APIs_3`, 2017.

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

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

[55] JGraph. JGraphX. `https://github.com/jgraph/jgraphx`, 2019.

[56] JotaBe. Pass extra parameters to jquery ajax promise callback. `https://stackoverflow.com/a/21985202`, February 2014.

[57] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, USA, November 1990.

[58] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In Jørgen Knudsen, editor, *ECOOP 2001 —– Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin / Heidelberg, 2001.

[59] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, chapter 10, pages 220–242. Springer Berlin / Heidelberg, 1997.

[60] Jörg Kienzle, Wisam Al Abed, Franck Fleurey, Jean-Marc Jézéquel, and Jacques Klein. *Transactions on Aspect-Oriented Software Development VII*, volume 6210 of *Lecture Notes in Computer Science*, chapter Aspect-Oriented Design with Reusable Aspect Models, pages 272–320. Springer, Berlin, Heidelberg, 2010.

[61] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-Oriented Multi-View Modeling. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development*, AOSD '09, pages 87–98, New York, NY, USA, March 2009. ACM.

[62] Jörg Kienzle and Rachid Guerraoui. AOP: Does It Make Sense? The Case of Concurrency and Failures. In *ECOOP 2002 - Object-Oriented Programming*, pages 37–61, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[63] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoît Combemale, Julien DeAntoni, Jacques Klein, and Bernhard Rumpe. VCU: The Three Dimensions of Reuse. In *Proceedings of the 15th International Conference on Software Reuse (ICSR 2016), Limassol, Cyprus, June 2016*, volume 9679 of *Lecture Notes in Computer Science*, pages 122–137, Cham, Switzerland, 2016. Springer International Publishing. ICSR 2016: Bridging with Social-Awareness.

[64] Jörg Kienzle, Gunter Mussbacher, Philippe Collet, and Omar Alam. Delaying Decisions in Variable Concern Hierarchies. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2016, pages 93–103, New York, NY, USA, October 2016. ACM.

[65] Charles W. Krueger. Software Reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, June 1992.

[66] Wayne C. Lim. Effects of Reuse on Quality, Productivity, and Economics. *IEEE Software*, 11(5):23–30, September 1994.

[67] Jihen Maâzoun, Nadia Bouassida, and Hanêne Ben-abdallah. A Bottom Up SPL Design Method. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2014, pages 309–316, Portugal, 2014.

[68] Andrian Marcus, Andrey Sergeyev, Václav Rajlich, and Jonathan I. Maletic. An Information Retrieval Approach to Concept Location in Source Code. In *11th Working Conference on Reverse Engineering*, pages 214–223, Nov 2004.

[69] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-up Adoption of Software Product Lines: A Generic and Extensible Approach. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, pages 101–110, New York, NY, USA, 2015. ACM.

[70] M. D. McIlroy. Mass-Produced Software Components. In Peter Naur and Brian Randell, editors, *Proceedings of NATO Software Engineering Conference*, pages 138–155, Garmisch, Germany, October 1968.

[71] Samuel G. McLellan, Alvin W. Roesler, Joseph T. Tempest, and Clay I. Spinuzzi. Building More Usable APIs. *IEEE Software*, 15(3):78–86, May 1998.

[72] Ashley McNeile and Nicholas Simons. Protocol modelling: A modelling approach that supports reusable behavioural abstractions. *Software & Systems Modeling*, 5(1):91–107, April 2006.

[73] Minueto Development Team. Official Minueto Website. `http://minueto.cs.mcgill.ca/`.

[74] Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering*, 12(5):471–516, 2007.

[75] Parastoo Mohagheghi and Vegard Dehlen. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture – Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 432–443. Springer Berlin Heidelberg, 2008.

[76] Mozilla MDN web docs. Function.prototype.bind(). `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind`, 2018.

[77] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 140–151, New York, NY, USA, 2014. ACM.

[78] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 935–946, New York, NY, USA, 2016. ACM.

[79] Barak Naveh and Contributors. JGraphT – a Java library of graph theory data structures and algorithms. `https://jgrapht.org`, 2018.

[80] Object Management Group. Meta Object Facility (MOF) Specification. OMG Document Number formal/16-11-01 (`http://www.omg.org/spec/MOF/2.5.1/`), November 2016. Version 2.5.1.

[81] Object Management Group. Unified Modeling Language (UML) Specification. OMG Document Number formal/17-12-05 (`http://www.omg.org/spec/UML/2.5.1/`), December 2017. Version 2.5.1.

[82] Oracle. Varargs. `https://docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html`, 2010.

[83] Oracle. The Java Tutorials — How to Write an Action Listener. `https://docs.oracle.com/javase/tutorial/uiswing/events/actionlistener.html`, 2017.

[84] Oracle. The Java Tutorials — Lesson: Java Applets. `https://docs.oracle.com/javase/tutorial/deployment/applet/index.html`, 2017.

[85] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[86] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin, Heidelberg, 2005.

[87] Python Software Foundation. The Python Tutorial: Arbitrary Argument Lists. `https://docs.python.org/3/tutorial/controlflow.html#arbitrary-argument-lists`, 2019.

[88] Radman Games. How to use Boost.Bind. `http://www.radmangames.com/programming/how-to-use-boost-bind`, 2011.

[89] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring Software Library Stability Through Historical Version Analysis. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 378–387. IEEE, 2012.

[90] Martin P. Robillard. Topology Analysis of Software Dependencies. *ACM Trans. Softw. Eng. Methodol.*, 17(4):18:1–18:36, August 2008.

[91] Martin P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 26(6):27–34, November 2009.

[92] Martin P. Robillard and Robert Deline. A Field Study of API Learning Obstacles. *Empirical Software Engineering*, 16(6):703–732, December 2011.

[93] Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 406–416, New York, NY, USA, 2002. ACM.

[94] ronmamo. Reflections – Java runtime metadata analysis. `https://github.com/ronmamo/reflections`, 2017.

[95] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. Mining Multi-level API Usage Patterns. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015)*, pages 23–32. IEEE, March 2015.

[96] Andrea Schauerhuber, Wieland Schwinger, Elisabeth Kapsammer, Werner Retschitzegger, Manuel Wimmer, and Gerti Kappel. A Survey on Aspect-Oriented Modeling Approaches. Technical Report, Business Informatics Group, Vienna University of Technology, 2006.

[97] Matthias Schöttle. Aspect-Oriented Behavior Modeling In Practice. Master's thesis, Department of Computer Science, Karlsruhe University of Applied Sciences, September 2012.

[98] Matthias Schöttle, Omar Alam, Abir Ayed, and Jörg Kienzle. Concern-Oriented Software Design with TouchRAM. In *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 29 - October 4, 2013.*, pages 51–55. CEUR-WS.org, 2013.

[99] Matthias Schöttle, Omar Alam, Franz-Philippe Garcia, Gunter Mussbacher, and Jörg Kienzle. TouchRAM: A Multitouch-enabled Software Design Tool Supporting Concern-oriented Reuse. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, pages 25–28. ACM, 2014.

[100] Matthias Schöttle, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. On the Modularization Provided by Concern-Oriented Reuse. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 184–189. ACM, 2016.

[101] Matthias Schöttle and Jörg Kienzle. On the Challenges of Composing Multi-View Models. *In the GEMOC'13 Workshop co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, October 2013.

[102] Matthias Schöttle and Jörg Kienzle. Concern-Oriented Interfaces for Model-Based Reuse of APIs. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, pages 286–291. IEEE, 2015.

[103] Matthias Schöttle, Nishanth Thimmegowda, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Feature Modelling and Traceability for Concern-Driven Software Development with TouchCORE. In *Companion Proceedings of MODULARITY 2015*, pages 11–14. ACM, 2015.

[104] Carolyn B. Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, July 1999.

[105] Bran Selic. What will it take? A view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, 2012.

[106] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. Reverse Engineering Feature Models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 461–470, New York, NY, USA, 2011. ACM.

[107] Diomidis Spinellis and Clemens Szyperski. Guest Editors' Introduction: How Is Open Source Affecting Software Development? *IEEE Software*, 21(1):28–33, January 2004.

[108] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling - State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 57–76, 2007.

[109] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, Boston, MA, USA, 2nd edition, 2009.

[110] Bjarne Stroustrup. C++11 - the new ISO C++ standard: std::function and std::bind. `http://www.stroustrup.com/C++11FAQ.html#std-function`, 2016.

[111] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM.

[112] The Android Open Source Project, Inc. Android XYZTouristAttractions Sample. https://github.com/googlesamples/android-XYZTouristAttractions/, April 2018.

[113] The Android Open Source Project, Inc. Android XYZTouristAttractions Sample. https://github.com/googlesamples/android-XYZTouristAttractions/, April 2018.

[114] The Chromium Authors. Callback<> and Bind(). https://chromium.googlesource.com/chromium/src/+/master/docs/callback.md, 2019.

[115] The Code Ship. A guide to Python's function decorators. https://www.thecodeship.com/patterns/guide-to-python-function-decorators/, 2014.

[116] The Go Authors. The Go Programming Language Specification: Function Types. https://golang.org/ref/spec#Function_types, 2018.

[117] The ReMoDD Team. Official ReMoDD Website. http://www.cs.colostate.edu/remodd/.

[118] Nishanth Thimmegowda, Omar Alam, Matthias Schöttle, Wisam Al Abed, Thomas Di'Meco, Laura Martellotto, Gunter Mussbacher, and Jörg Kienzle. Concern-Driven Software Development with jUCMNav and TouchRAM. In *Proceedings of the Demonstrations Track of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, October 1st and 2nd, 2014*. CEUR-WS.org, 2014.

[119] C. Treude, M.P. Robillard, and B. Dagenais. Extracting Development Tasks to Navigate Software Documentation. *Software Engineering, IEEE Transactions on*, 41(6):565–581, June 2015.

[120] Gias Uddin and Martin P. Robillard. How API Documentation Fails. *Software, IEEE*, 32(4):68–75, July 2015.

[121] w3schools.com. JavaScript Function Parameters. https://www.w3schools.com/js/js_function_parameters.asp, 2019.

[122] Thomas Weigert. Practical Experiences in Using Model-Driven Engineering to Develop Trustworthy Computing Systems. In *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing -Vol 1 (SUTC'06) - Volume 01*, SUTC '06, pages 208–217, Washington, DC, USA, 2006. IEEE Computer Society.

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

[124] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2013.

[125] Wikipedia contributors. Partial Application — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.php?title=Partial_application&oldid=882475287`, 2019.

[126] Norman Wilde and Michael C. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance*, 7(1):49–62, January 1995.

[127] Karl Erich Wolff. A First Course in Formal Concept Analysis: How to Understand Line Diagrams. In F. Faulbaum, editor, *SoftStat'93, Advances in Statistical Software 4*, pages 429–438. Gustav Fischer Verlag, Stuttgart, 1994.

[128] wxPyWiki. Passing Arguments to Callbacks. `https://wiki.wxpython.org/Passing%20Arguments%20to%20Callbacks`, 2011.

[129] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. Feature Location in a Collection of Product Variants. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, WCRE '12, pages 145–154, Washington, DC, USA, 2012. IEEE Computer Society.

[130] Tewfik Ziadi, Luz Frias, Marcos Aurelio Almeida da Silva, and Mikal Ziane. Feature Identification from the Source Code of Product Variants. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 417–422, Washington, DC, USA, 2012. IEEE Computer Society.

A
CORE Metamodel

Figure A.1 provides the complete overview of the CORE metamodel, which has been shown separated in logical parts in section 2.4.5. Grey classes denote classes that have to (in the case of *abstract* classes) or may be sub-classed. The latter is the case with `COREMapping`, which is not *abstract* and can therefore be used as is. However, modelling languages might want to further specialize a mapping. Blue associations show relationships between the concern and a model, i.e., they span across several models. This requires special care to ensure that the models are always in sync.

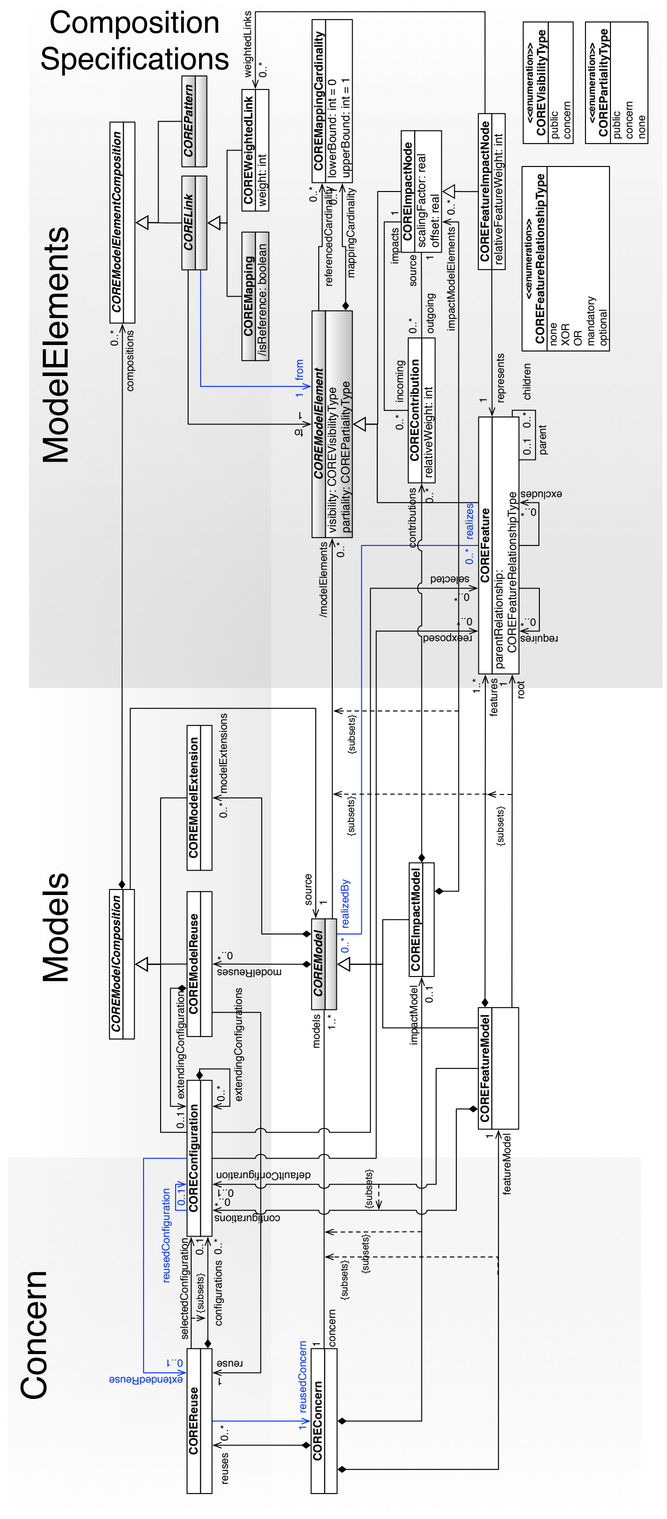


Figure A.1: The Complete CORE Metamodel.

B

Interview Guide for Developers of Minueto

The following contains the interview guide with the main questions that were part of our interviews with the developers of the Minueto framework. Most questions led to follow-up questions, because we performed a semi-structured interview. Furthermore, we also asked clarification questions based on the feature model the participants produced prior to the interview.

The purpose of this interview is to gain deeper insight into how you came up with the feature model of *[framework]*, which information you used, and learn more about the design of <*framework*>.

Let's start by talking about your feature model of *[framework]*.

1. How did you come up with the feature model?
2. What information did you use?
3. Do you think it is complete? I.e., it represents all the features that your framework provides.
4. Are the constraints and groupings complete? I.e., are there any combinations of feature selections that shouldn't be possible.

Now that we looked at the user-perceived features, let's look at the API of *[framework]* and how the individual API elements correspond to the features.

1. Can you identify which parts of the API belong/correspond to which feature?

[framework] is packaged with runnable examples/tutorials.

1. Can you identify the features from your feature model that each example makes use of?

Here is a feature model of *[framework]* that we came up with after familiarizing ourselves with it.

1. Are there features you agree or disagree with?

   (a) Did we include features that are not actually features?
   (b) Did we split something into sub-features that can't be separated?

2. Did we miss any features?
3. Are there structural problems in terms of groupings?
4. Are there any dependency problems in terms of parent-child relationships?
5. Do you agree with the constraints we elaborated?

C
List of Examples Provided with Minueto

The following contains the list of runnable examples that are provided with Minueto. Their fully qualified names (i.e., the class name prefixed with its package name) are used.

1. `org.minueto.sample.HelloWorld`
2. `org.minueto.sample.LineDemo`
3. `org.minueto.sample.fireinthesky.FireInTheSky`
4. `org.minueto.sample.image.AlphaImageDemo`
5. `org.minueto.sample.image.CircleDemo`
6. `org.minueto.sample.image.DrawInImageDemo`
7. `org.minueto.sample.image.GetSetPixelDemo`
8. `org.minueto.sample.image.ImageDemo`
9. `org.minueto.sample.image.LoadingFileDemo`
10. `org.minueto.sample.image.RectangleDemo`
11. `org.minueto.sample.image.text.FramerateDemo`
12. `org.minueto.sample.image.text.TextDemo`
13. `org.minueto.sample.image.text.TextDemo2`
14. `org.minueto.sample.image.transformation.CropDemo`

15. `org.minueto.sample.image.transformation.HighlightDemo`
16. `org.minueto.sample.image.transformation.RotateDemo`
17. `org.minueto.sample.image.transformation.RotateDemo2`
18. `org.minueto.sample.image.transformation.RotateDemo3`
19. `org.minueto.sample.image.transformation.ScaleDemo`
20. `org.minueto.sample.image.transformation.ScaleFlipDemo`
21. `org.minueto.sample.input.HandlerDemo`
22. `org.minueto.sample.input.HandlerDemo2`
23. `org.minueto.sample.input.HandlerDemo3`
24. `org.minueto.sample.input.TriangleRover`
25. `org.minueto.sample.input.TriangleRover2`
26. `org.minueto.sample.swing.launch.LauncherDemo`
27. `org.minueto.sample.swing.panel.PanelDemo`
28. `org.minueto.sample.window.LineBenchmark`
29. `org.minueto.sample.window.MultiWindowDemo`
30. `org.minueto.sample.window.ResolutionChangeDemo`
31. `org.minueto.sample.window.ScreenshotDemo`