Associations in MDE: A Concern-Oriented, Reusable Solution

Céline Bensoussan

MASTER OF SCIENCE

School of Computer Science

McGill University

Montréal, Québec, Canada

March 2016

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

DEDICATION

To my family and friends,

for all the support they have given me.

Thank you.

ACKNOWLEDGEMENTS

I would first like to thank my thesis advisor, Prof. Jörg Kienzle for all his time and energy. I enjoyed the long meetings and discussions.

Thank you to Matthias Schöttle for all the effort and time he invested in me while working in the lab and for reading my thesis.

I am also grateful to Vincent Foley for agreeing to also read my thesis and giving me feedback even without knowing anything about MDE.

Finally, I thank Nishanth and Berk as well as Matthias again for a great lab spirit and to all those lunches and laughs together.

ABSTRACT

Associations are a very common concept in software modelling, in particular when using class diagrams for expressing domain models or structural design models. Concern-Oriented Reuse (CORE) proposes a new way of structuring model-driven software development where models of the system are modularized by domains of abstraction within units of reuse called concerns. This thesis illustrates how many of the variations of associations and association implementations have been captured within a concern called *Association*, together with behavioural models that ensure uniqueness, minimum/maximum size constraints and referential integrity. Furthermore, the provided *Association* concern documents the impact of using a specific association variation on high-level system qualities. Finally, the thesis describes how the TouchCORE tool was streamlined to support concern-oriented, agile, UML-conformant modelling with associations and rapid selection of association variants according to desired high-level system qualities.

ABRÉGÉ

Les associations sont un concept très commun en modélisation de logiciel, particulièrement en utilisant des diagrammes de classe pour représenter des modèles de domaine ou des modèles de conception. La réutilisation orientée préoccupation (CORE) propose une nouvelle manière de structurer le développement de logiciel dirigé par des modèles où les modèles du système sont modulables à l'intérieur d'une unité réutilisable appelée une préoccupation. Cette thèse illustre les variations des associations et la mise en œuvre des associations capturées dans une préoccupation appelée *Association*, ainsi que les modèles de comportement qui garantissent l'unicité, les contraintes de taille minimum ou maximum d'une collection et de l'intégrité référentielle. De plus, la préoccupation *Association* documente les impacts sur les qualités du système lors de l'utilisation d'une certaine variation. Enfin, cette thèse décrit comment l'outil TouchCORE a été modifié pour faciliter la réutilisation de cette préoccupation de manière agile et adaptée aux normes UML. Ses modifications permettent une sélection rapide d'une variation de l'association selon les qualités du système choisies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
Introduction

Model-Driven Engineering (MDE) [12] is a unified conceptual framework in which software development is seen as a process of *model production*, *refinement,* and *integration*. To reduce the accidental complexity and the effort needed to move from a problem domain to a software-based solution, MDE advocates the use of different modelling formalisms, i.e., modelling languages, to represent and analyze the system from *multiple points of view*. For each level of abstraction, the modeller uses the best formalism that concisely expresses the properties of the system that are important to that level. During development, high-level specification models are refined or combined with other models to include more solution details, such as the chosen architecture, data structures, algorithms, and finally even platform and execution environment-specific properties. The manipulation of models is achieved by means of model transformations, ideally automated by model transformations tools [14].

In the context of MDE, *associations* play an important role. During the requirements engineering phase, they are used at a high level of abstraction to formalize relationships among domain concepts in so-called domain models. In later development phases, as the architecture of the software and the solution it implements begin to take form, properties are attached to the associations, e.g., *ordering*, *uniqueness*, *multiplicity, role name* and *navigability*. Finally, during the implementation phase, concrete data structures, such as *arrays*, *linked lists* or *hash tables*, are used to realize associations with multiplicity greater than one.

Because associations are widely used in MDE, modelling tools with code generators have to generate code from models that contain associations. However, most current code generators do not provide adequate support for associations [16, 5, 21, 18, 10]. For example, the properties of associations specified in the model, e.g., multiplicity constraints and bidirectionality, are rarely enforced in the generated code. Furthermore, there are many ways of implementing associations with multiplicity greater than one using different collection data structures. Each data structure has different run-time behaviour, and therefore affects the non-functional qualities of the software that is being developed, e.g., performance and memory use. Current modelling tools, however, shield the modeller from implementation details. As a result, they do not document or quantify the impact on non-functional qualities that underlying implementations for associations have. As a result, code generators typically resort to default implementation strategies for associations that do not take into account high-level goals and non-functional requirements of the application that is being built.

This thesis describes how the concern-orientation reuse paradigm (CORE) [7], which is extending MDE, is used to capture many different kinds of associations, their properties, behaviours, and various implementation solutions within a reusable artifact: the *Association* concern. It encapsulates models for all variants, and exploits aspect-oriented modelling techniques to modularize the structure and behaviour required for enforcing uniqueness and multiplicity constraints, as well as referential integrity for bidirectional associations. Furthermore, it bundles many of the Java collection implementations that can be used to realize associations. For each of the bundled implementation classes, the impact of their use on non-functional qualities, e.g., memory consumption and performance, has been determined and formalized within the concern. Furthermore, the thesis proposes to make the reuse of

the *Association* concern simpler with the help of a domain-specific language (DSL). Based on this UML-inspired visual notation, a modeller can apply the *Association* concern within his own models with minimal effort and time. The proposed DSL was implemented within the TouchCORE modelling tool.

Specifically, this thesis makes the following contributions:

- **Design of an Association Concern**: Creation of a concern that encapsulates many ways of dealing with associations in MDE.
  - Capture the different, user-relevant variations of associations in a feature model.
  - Specify the impacts of different implementations of associations on memory use and performance using impact models. This required running experiments to:
    (a) Compare execution times of insertion, access, iteration and removal operations.
    (b) Profile memory usage.
  - Encapsulate the structural properties of the different variations and possible implementations of associations in class diagrams.
  - Encapsulate behaviours to ensure uniqueness, multiplicities and referential integrity using sequence diagrams.
  - Deal with feature interactions.
- **Applying of the Association Concern**: Enable easy reuse of the *Association* concern.
  - Define a DSL for reusing the *Association* concern based on UML notations.
  - Explain how to update typical class diagram meta models to add support for reusing the *Association* concern.

- Describe updates to the CORE class diagram weaver required for supporting the DSL.
- Specify algorithms for automated selection of features and creation of customization mappings to ensure consistent use of the *Association* concern,
- Propose enhancements to the modelling tool GUI that prevent visualization clutter.

The thesis is structured as follows. Chapter 2 presents background information on concern-oriented reuse: the interfaces of a concern and how a concern is reused. It also presents the multi-view modelling notation Reusable Aspect Models (RAM) with its structural and message views. Chapter 3 presents how the *Association* concern was designed by describing each of its interfaces separately, followed by an overview of the structural and behavioural realization models. Chapter 4 explains ideas on how the reuse of the *Association* concern can be streamlined through a DSL. It also provides implementation details on how these ideas were realized in the TouchCORE tool, and how we improved its GUI. Chapter 5 describes the benchmarks that were performed on different Java collections and that supported the creation of impacts models. The related work is discussed in Chapter 6. The last chapter, Chapter 7 concludes this thesis and presents some thoughts on future work.

Chapter 2
Background

Reusability is an important concept of software development. When building a system, engineers need to break it down to components in a manner that those components could be used in other systems. This can happen from the very beginning or later, when components are detected to have a reuse potential [or: to be able to reuse them in other components or other systems]. MDE faces the same reusability challenge, and instead of defining models from scratch, it is desirable to reuse models as much as possible. This can be facilitated with modelling notations that explicitly support the concept of reuse. Reusable models could represent anything that can be applied at several locations within one or more software systems, such as libraries, design patterns, persistency or utilities.

A simple example of a stock application is shown in Figure 2–1. The structure defines the classes `Stock` and `StockWindow`. A stock has a price, which is displayed in the window. Every time the price of the stock changes, the window needs to reflect that change. This is a common behaviour and is known as the *Observer* design pattern [15]. Usually, the structure and behaviour required to integrate this into the system is to add it directly into the design. However, it would be best to work from a reusable entity. Concern-Orientation is a technique, which facilitates this.

The first section of the background deals with concern-oriented reuse. It describes the three interfaces of a concern and the three steps to reusing a concern. We then describe the CORE metamodel. Section 2.2 describes a multi-view modelling notation that extends the

CORE metamodel called Reusable Aspect Models. The last section present the modelling tool TouchCORE and how it uses the metamodel described in the two previous sections to provide an interface to build concerns.

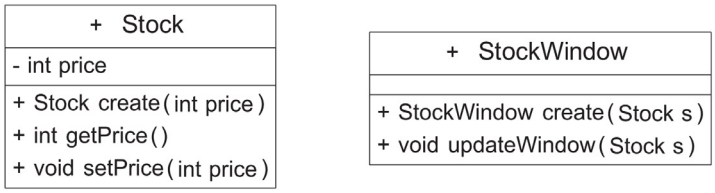


Figure 2–1: *StockExchange* Application

2.1 Concern-Oriented Reuse

Concern-Orientation (CORE) captures the variations of a domain of interest in a reusable artifact called a *concern* [7]. In addition, it documents what the impact of selecting a variation has on high-level system qualities. Concerns are designed by domain experts and all the complexity is hidden from the user behind well-defined interfaces [8]. They are designed as generic as possible to facilitate reuse. This section presents the interfaces of a concern, how it is reused in an application and the CORE metamodel used by the TouchCORE modelling tool [34, 31].

2.1.1 CORE Interfaces

Concerns define *interfaces* that detail how the unit should be reused. A concern is composed of three interfaces: a *variation interface*, a *customization interface* and a *usage interface*. To illustrate those interfaces, we use the *Observer* concern based on the Observer design pattern and show how it is applied in the *StockExchange* application.

Variation Interface

The variation interface describes the different variations of a concern and its impacts on user goals. The variations are defined using a feature model originally defined by Kang et al. [24]. A feature model shows all features in a tree and a feature has a relationship with its parent. Each feature represents a piece of the product. The relationship between a parent feature and its children can either be *optional* (○), *mandatory* (●), *OR* (▲), where at least one child feature needs to be selected, or *XOR* (△) where exactly one child feature must be selected. Cross-tree constraints are also part of a feature model through *require* constraints, where a feature forcing the selection of another, and *exclude* constaints, where a feature prohibiting the selection of another.

The Observer design pattern has different communication methods: push and pull. The push method happens when the observed element (the subject) sends directly the updated version of itself to the observer. The pull method requires the observer to retrieve the state from the subject once it was notified of a change. These are variations expressed in the feature model, which is shown in Figure 2–2. The two methods are generalized by a mandatory feature called *NotificationMethod*. Only one notification method can be selected, therefore *Pull* and *Push* are mutually exclusive in the feature model. The feature *Controller* is used when notifications should go through a controller instead of going directly from a model to a view. This is an optional feature.

In the *StockExchange* application, the window may either pull the changes of the price when it gets an update that the state of the subject has changed or the stock could push its new state to the window when it is modified.

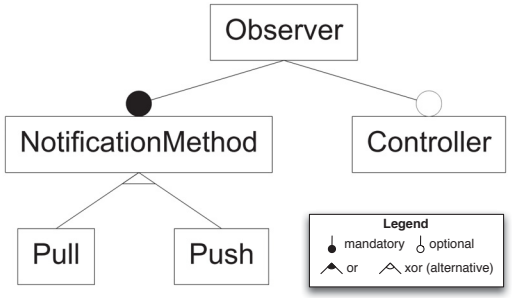


Figure 2–2: Feature Model of the *Observer* Concern

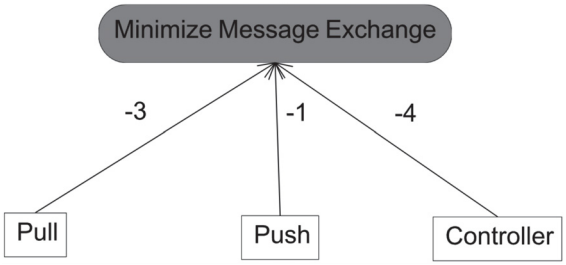


Figure 2–3: *Minimize Message Exchange* Impact Model of the *Observer* Concern

The impacts of a feature on non-functional requirements are defined through impact models, which are based on goal models [9]. The impact model captures how features influence system qualities, focusing on one system quality at a time. The impact model in Figure 2–3 shows an example of a user goal *Minimize Message Exchange* in the *Observer* concern. The feature *Push* minimizes the most the message exchange, because when the object is modified the subject notifies the observer with the necessary information. With *Pull*, however, the subject just notifies the observer of a change, which is itself responsible for pulling the information from the subject. The *Controller* increases the message exchange even more as messages do not go directly from the model to the view but from the model to the controller, and then from the controller to the view and vice versa. Other user goals for this concern are *Increase Performance for Small Data* and *Increase Performance for Big Data*. When the user selects features, the impact models are evaluated and give feedback to the user. Impacts are of great value to users when reusing a concern as they allow to understand the tradeoffs when making a particular selection.

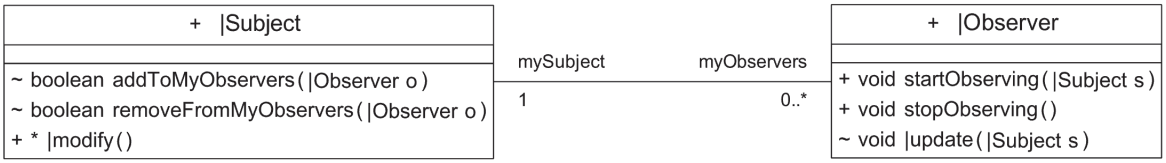


Figure 2–4: Model of the *Push* Variation of the *Observer* Concern

Customization Interface

Features are realized by models that describe its structure. More information about these models, called realization models, is presented in Section 2.1.3 and the modelling notation RAM is described in Section 2.2. From those models, we define a *customization interface*. This interface describes which generic elements of a concern need to be adapted to concrete elements in an application. Elements that are part of the customization interface are represented with a vertical bar '|'. This denotes elements that are incomplete and have to be completed by the reusing concern. When reusing the concern, they are mapped to elements in the current application. As shown in the structural model in Figure 2–4, the *Observer* concern is composed of a subject and an observer. Here, the customization interface is `|Subject`, `|Observer` and their respective operations `|modify` and `|update`. When reusing the concern, a class has to be the subject and another the observer. In the *StockExchange* application, `Stock` is the subject and `StockWindow` is the observer. The operation `|modify` needs to be mapped to an operation of the subject and the operation `|update` needs to be mapped to an operation of the observer. Omitting the mapping of elements of the customization interface is a misuse of the concern.

Usage Interface

The *usage interface* is the part of the concern that is accessible to the application, it is available for the user once it is reused. It is defined by the public elements of the concern. Looking again at Figure 2–4, `|modify` of the subject as well as `startObserving` and `stopObserving` of the observer are part of the usage interface. When `|Observer` is mapped to `StockWindow`, `StockWindow` gets the `startObserving` and the `stopObserving` operations. However, it is not the case for `|Subject` as the public operation is partial. It should be mapped when reusing and therefore is not used directly but through the operations it is mapped to.

2.1.2 Reusing a Concern

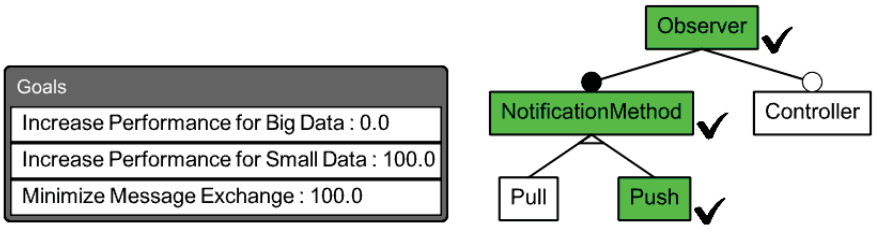
Reusing a concern involves three steps from the concern user, once he selected the concern of interest through the reusable concern library provided by the tool TouchCORE:

1. Then the *variation interface* of the concern is presented to the user as shown in Figure 2–5a. The modeller must make a selection of the desired variant. Evaluation of the impact of the selection on each high-level goal is presented to help the user make a decision and allow to perform a tradeoff analysis. The contributions are calculated from the impact models in a way that the highest possible value is `100` and the lowest is `0`, `100` being the best and `0` the worst. In the given example, when reusing the *Observer* concern, the user selects *Push,* which fully minimizes the message exchange and fully increases performance for small data, but not at all for big data. This is because a change in the observed object triggers an update to the observer by sending the whole object even if it is only interested by a small portion of it.

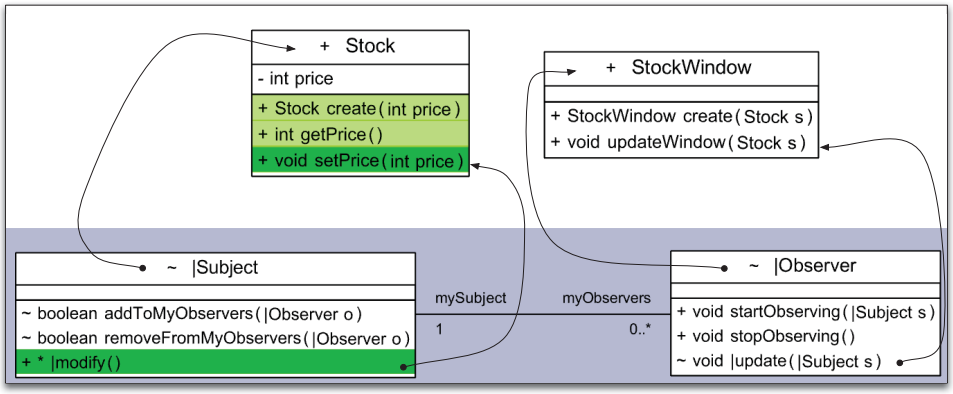
2. Once a selection is made, the tool weaves the models that realize the selected features together. Weaving means combining models, the weaver takes the model in the lower feature in the tree and combines it with its parent and it goes up the feature tree until the root.Then, when the weaving is done, the concern is ready to be customized to the current application. As displayed in Figure 2–5b, the customization interface allows selecting an element from the concern and mapping it to an element in the application. In our example, the reusing concern is the *StockExchange* concern. The `StockWindow` is notified every time there is a change in the price of a `Stock` and it updates the display of the price. Here, `|Subject` is mapped to `Stock` and `|Observer` is mapped to `StockWindow`. `|modify` is mapped to `setPrice` and `|update` is mapped to `updateWindow`.
3. The modeller is now able to use the concern through the usage interface by using the provided operations.

2.1.3 CORE Metamodel

A metamodel formally defines the structure that all models have to conform to, i.e., it defines a common language and can be seen as a grammar. A metamodel itself is a model at a higher level of abstraction. The CORE metamodel formalizes the concepts of CORE. This enables the integration of other modelling languages within its framework [29]. Concerns are built based on the CORE metamodel. An excerpt of the metamodel is shown in Figure 2–6. A *COREConcern* is built from one or more *COREModel*s, some of which we showed in the previous section: a *COREFeatureModel* and a *COREImpactModel*. Other models are left to be designed by a modelling language, which can be extended to support CORE [33].



(a) Step 1: Feature Selection for the Observer Concern



(b) Step 2: Establishing the Mapping with Observer Customization Interface (below) and Application (above)

Figure 2–5: Reusing the *Observer* Concern

A *COREFeatureModel* is composed of *COREFeatures.* Those features may be realized by many models. The realization models are the ones reusing other concerns. When a *COREModel* reuses a concern, it contains a *COREModelReuse*, which is associated to a *COREReuse* that holds the selected configuration, i.e, the features selected by the user when reusing. The *COREReuse* also holds a reference to the reused concern.

A *COREImpactModel* is composed of *COREImpactNodes.* These impact models contribute to one another through *COREContributions* that are represented with a number called a *weight.* Features are also impact nodes and contribute to goals.

Models build on top of other models that realize parent features. We describe in the next section a concrete modelling language and how models can extend each other.

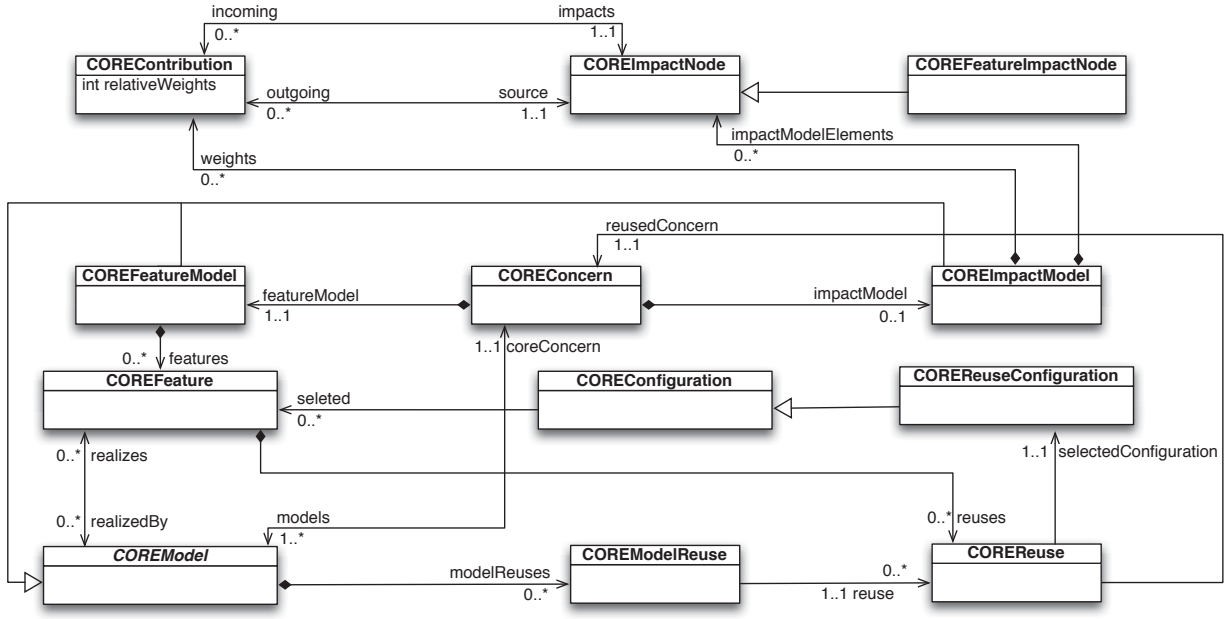


Figure 2–6: Excerpt of the CORE Metamodel

2.2 Reusable Aspect Models

The multi-view modelling notation Reusable Aspect Models (RAM) [6] extends the CORE metamodel and provides a specific modelling language. A reusable aspect model extends three diagrams from the Unified Modeling Language (UML) [20]: class diagrams, sequence diagrams and state diagrams, respectively called in RAM *structural view*, *message view* and *state view.* As seen in the CORE metamodel in Figure 2–6, a model realizes features and a feature is usually realized by a model. An aspect, also called a *realization model*, is a *COREModel* as shown in the excerpt of the RAM metamodel in Figure 2–7. It is used to describe the structure and behaviour of a concern by using aspect-oriented techniques. The structural view and the message view are presented in Section 2.2.1 and Section 2.2.2 respectively. State views are omitted as they were not used for the work of this thesis.

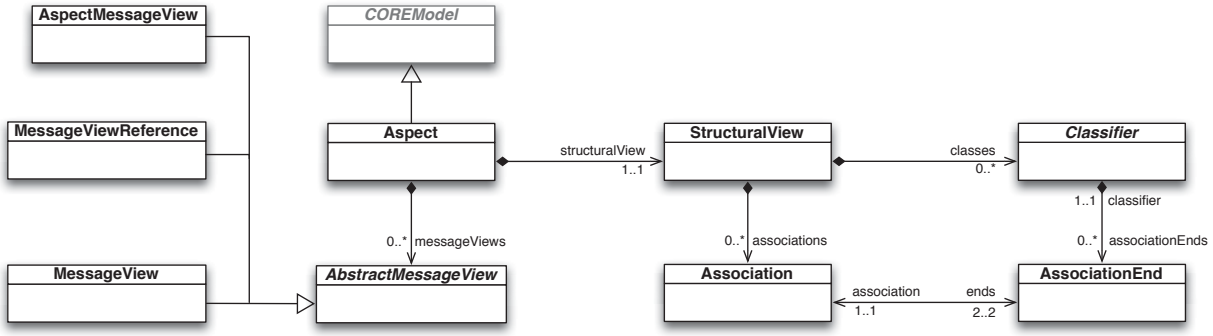


Figure 2–7: Excerpt of the RAM Metamodel

2.2.1 Structural View

An aspect is composed of a structural view as we observe when looking again at the RAM metamodel in Figure 2–7. The structural views in TouchCORE are class diagrams based on UML abstractions, they define classes with their attributes and operations and

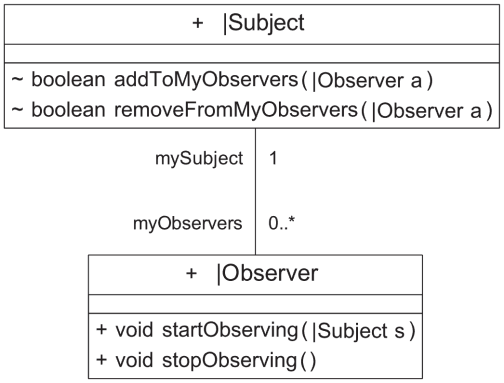


Figure 2–8: *Observer* model

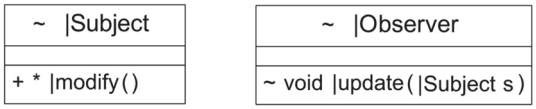


Figure 2–9: *Push* model

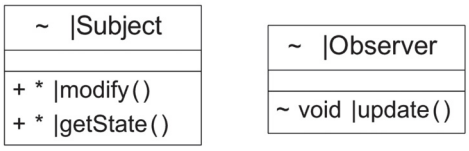


Figure 2–10: *Pull* model

associations between those classes. A structural view is composed of *Classifiers*. A classifier contains the association ends and the structural view contains all the associations in the view. We already saw a RAM model in Figure 2–4 when describing the customization and usage interface. *Classifiers* may be of two kinds: classes defined by a user or *implementation classes*. Implementation classes are defined by a target programming language or a framework and can be be imported into a model using the TouchCORE tool [28].

Models may extend other models defined for a parent feature. Each model only defines what is needed for the particular feature it realizes. Models that are extended are called extended models. The woven model of the feature *Push* in Figure 2–4 results from combining elements in the structural view of the feature *Observer* (see Figure 2–8) and the structural view of the feature *Push* (see Figure 2–9). This allows to create another variation such as *Pull* (see Figure 2–10), without redefining the common structure and behaviour. It is beneficial to break down the concern in smaller models as it favours reuse and extraction of shared patterns.

2.2.2 Message View

Message views are based on UML's sequence diagrams, but add more detail and enforce that elements can only be used if they are defined in the structural view [27]. They are used to describe the behaviour of an operation. An operation may be defined only by one message view. Message views in TouchCORE support variable declaration, assignments, calls and fragments (if statements, while and for loops and try/catch). Message views are connected to the elements in the structural model, therefore elements in the message view correspond to elements in the structural view. A lifeline, i.e., vertical lines representing objects, can only be created for an element if its class is defined or imported in the model. The operations called on a lifeline need to have been created for the classifier in the structural view.

As an example, we use the message view of the operation `startObserving(|Subject s)` that belongs to `|Observer` (see Figure 2–11). The target observer sets its subject to `s`, i.e., the element to start observing, and adds itself to the list of observers held by `s`.

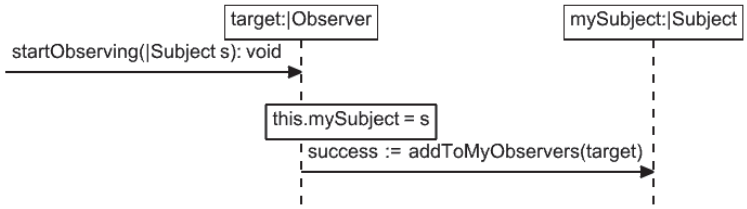


Figure 2–11: Message View of the `startObserving` Operation

Aspect Message View

Using aspect-oriented techniques, it is possible to augment the original behaviour of an operation. Aspect message views are used to advise operations. We also say that operations are *affected by* aspect message views, as they are aware of the aspect message views advising them. Operations defined in a concern that are part of the customization interface do

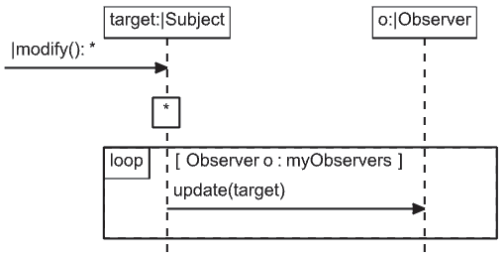


Figure 2–12: Aspect Message View *notification* of the `|modify` Operation

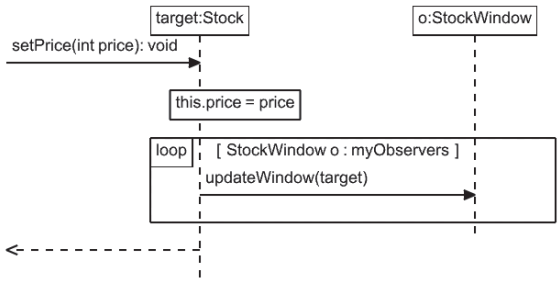


Figure 2–13: Message View of `setPrice` Once Mapped and Woven

not define message views but aspect message views are defined to advise them. Concerns are generic and mapping an operation from the concern to an operation in the application augments the original behaviour through this aspect message view, it does not replace it.

Figure 2–12 shows the aspect message view that is defined to advise the partial operation `|modify`. The aspect message view is required because `|modify` is going to be mapped to an operation in the application that already defines a message view. The aspect message view *notification* adds behaviour after the original behaviour, which is represented by the white box. It iterates through the observers of the subject and calls `|update` on each of them. In the *StockExchange* example, `|modify` is mapped to `setPrice`, as seen previously in Figure 2–5b. Therefore, the aspect message view *notification* advises the behaviour of `setPrice`. Then, the weaver, described in Section 2.2.2, merges the original behaviour defined in the message view of the operation and the behaviour defined in the aspect message view that advises the operation in the application. This results in the structural view as shown in Figure 2–4 with the message view for `setPrice` shown in Figure 2–13.

Message View Reference

A user might need to advise an operation of the usage interface from a reusing concern. For example, in the *StockExchange* example application, the user might want to add behaviour to the `startObserving` or the `stopObserving` operation. However, the user can not directly modify the reused concern, because it is only accessible through the defined interfaces. Similarly, a concern designer might need to advise an operation of an extended model. However, an extending model can only add additional elements.

In order to support this, a message view reference can be defined. It allows one to connect an aspect message view containing an advice with the message view defining the original behaviour. The referenced message view is located in the reused concern or extended model.

For example, in the case of the model of the feature *Push* in Figure 2–9, which is extending *Observer*, the user might want to augment the behaviour of `startObserving` and `stopObserving` without overwriting the one defined in *Observer*.

Weaver

The weaver combines extended models when reusing a concern and also the current application models with the reused concerns to produce the final application models. It takes care of both the structural view and the message views. When reusing a concern, elements are mapped. The weaver uses those mappings to merge classes and operations. Elements that are not mapped are copied over to the woven model. Similarly, when combining with extended models, it merges elements that are mapped or that have the same name and copies over the rest. Figure 2–4 illustrates the woven model of *Push* (see Figure 2–9) and

its extended model *Observer* (see Figure 2–8). The message views are also woven, as seen previously in Figure 2–13, when an operation is affected by an aspect message view.

2.3 TouchCORE Modelling Tool

TouchCORE is a multi-touch enabled, software design modelling tool that implements two concepts seen in the previous section: Concern-Orientated Reuse (CORE) and Reusable Aspect Models (RAM). TouchCORE uses the CORE framework through the metamodel we described. The tool provides a graphical interface that allows using theses concepts together. It supports concern reuse which involves features and impact models, as well as realization models expressed using class, sequence and state diagrams, and Java implementations. The tool aims at building scalable et reusable software design models.

Chapter 3
Concern-Oriented Modelling of Association

A concern is a reusable artifact. Building a concern is a tedious task that requires a very good understanding of the domain. The concern needs to provide a complete and understandable interface to the user while covering all issues that might arise from applying this particular concern to an application. Our goal is to build an *Association* concern to be reused every time an association is created between two classes. This concern has to offer common features from associations while dealing with multiplicity and uniqueness constraints and referential integrity.

In this chapter, we present the *Association* concern we designed by describing each of its interfaces. As seen previously, a concern is composed of three interfaces: a *variation interface*, a *customization interface* and a *usage interface*. Section 3.1 presents the variation interface that is composed of a feature model and an impact model that describes the impacts each feature has on different user goals. The customization interface and the usage interface are presented in Section 3.2 and Section 3.3 respectively. Section 3.4 describes the structural realization: the models for each feature and how they extend each other to build a complete model. The behavioural realization, designed through sequence diagrams, is presented in Section 3.5. When some features are selected together, the structure or behaviour may change and to redefine them, the feature interaction realization is described in Section 3.6.

3.1 Variation Interface

The variation interface exposes the different features of the concern through a feature model and describes the impact that those features have on non-functional requirements through impact models. The feature model presents the different features of the concern to the user and how those features interact and depend on each other. Coming up with a feature model requires breaking down the domain into pieces and finding a way to fit them back together through common properties. The impacts play an important role in providing the user with a sense of how a feature affects high-level goals and non-functional qualities, such as performance or memory usage.

3.1.1 Feature Model

The variation interface of the *Association* concern is represented by the feature model in Figure 3–1. The features of the *Association* concern can be classified into three different groups according to the properties they ensure: the properties of the association (maximum, minimum, bidirectional and unique), the properties of the collection for associations with `multiplicity > 1` (key-indexed, ordered and unordered), and the implementation classes used to implement the collections (currently supporting `ArrayList`, `LinkedList`, `Stack`, `HashSet`, `TreeSet`, `HashMap` and `TreeMap`). The implementation choices are captured from the `java.util` package for ordered lists (`ArrayList`, `LinkedList`, and `Stack`), for unordered sets (`HashSet` and `TreeSet`), and for key-indexed variations (`HashMap` and `TreeMap`). The properties of the collections were discovered by comparing individual implementation classes and extracting their common properties. The feature model also supports constraints that will be discussed later.

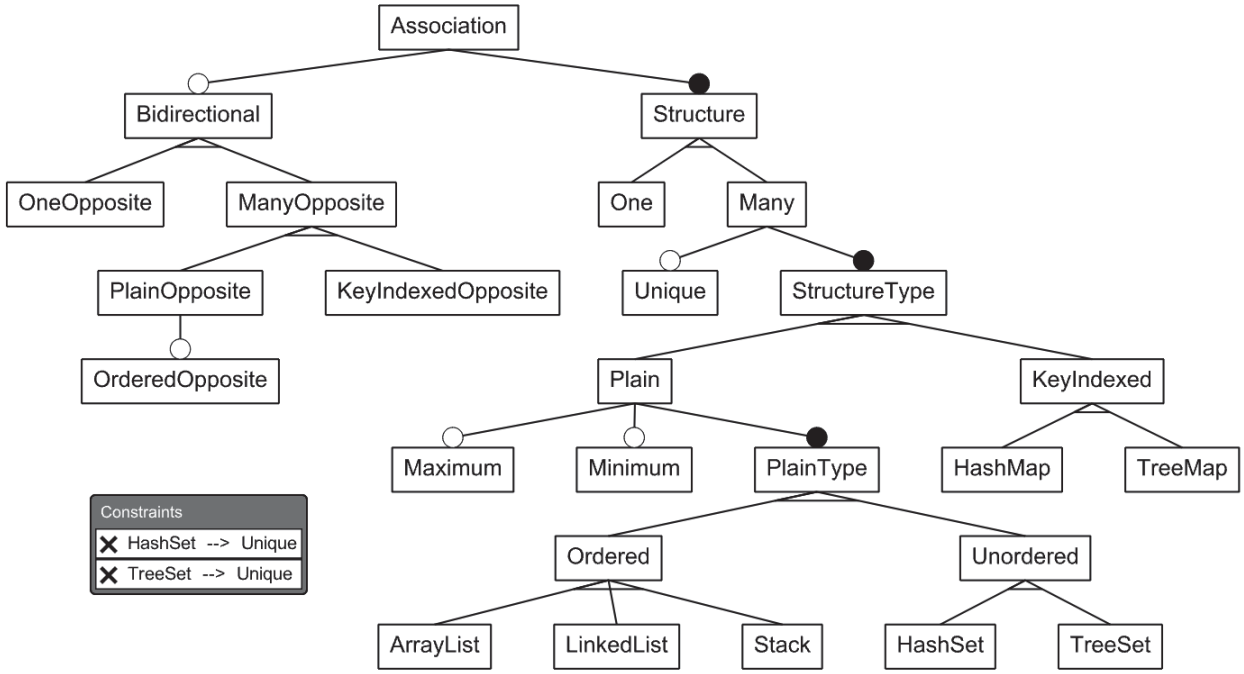


Figure 3–1: The Feature Model of the *Association* Concern

Properties of Collections and Implementation Classes

The root feature *Association* has two subtrees. The first one, *Structure*, groups all the features related to properties of the collections, and its leaf features represent the implementation classes made available by the concern. The subtree is separated into two categories: *One* and *Many*. They differentiate between an association with a multiplicity of one (a single object) and associations with a multiplicity of many (a collection of objects). *One* is responsible for providing getter and setter functionality to the class containing the association. *Many* is responsible for providing all operations related to manipulating collections. We noticed two types of collections: some collections have their elements associated with a key of a chosen type, generally known as maps that we represent with the feature *KeyIndexed*. It

currently has two children features *HashMap* and *TreeMap* that provide the implementation classes with the same name. We generalized all the other collections (not key-indexed) in a feature called *Plain*. *Plain* is further divided into three sub-features, two of them being properties of the association, *Minimum* and *Maximum*, and the last one, *PlainType* is itself divided into two sub-features: *Ordered* and *Unordered*. Some collections keep track of a certain order in which its elements were added while others do not. Provided ordered implementation classes are *ArrayList*, *LinkedList* and *Stack* and unordered implementation classes are *HashSet* and *TreeSet*. Breaking down *Association* into many features that incrementally specify the association in more detail helps reusing the concern without making a final decision, i.e., without selecting a leaf feature. A user might not be ready to choose a concrete implementation class when creating the association, and CORE allows one to delay the decision for later. For example, the user could just select the *Many* feature and start using the association without choosing a concrete implementation.

Properties of Associations

In addition to properties of the collection, the concern needs features to define the properties of the association in order to provide appropriate behaviour. Associations can be bidirectional, which is captured in the *Bidirectional* subtree. An association is bidirectional as soon as it is navigable in both directions, in which case any implementation must ensure referential integrity. The child features of *Bidirectional* correspond to the multiplicity and the feature selection of the opposite association end. A `1-1` association would correspond

to the features *One* and *OneOpposite*. A `1-*` association[1] would correspond to the features *One* and *ManyOpposite*, or one of its children depending on the chosen kind of the opposite association. All the sub-features of *Bidirectional* encapsulate the behaviour needed to ensure referential integrity. This is achieved by advising insertion and removal operations in the design models of those features. For example, in a bidirectional many association, the insertion operation on one side needs to know whether to set or insert on the other side and vice versa. The concern deals with all combinations.

For some associations with `multiplicity > 1`, it makes sense to decide whether the same element can be part of the same association instance more than once or not. The optional feature *Unique* encapsulates this property and can apply to a *Plain* or *KeyIndexed* collection. It ensures that the insertion of an element into the collection realizing the association is only allowed if the element is not already contained inside. Some implementations of unordered collections are collections that do not allow for duplicates, such as *HashSet* and *TreeSet*. To capture this, we added cross-tree constraints to the feature model. It is hence invalid to select one of those features without also selecting *Unique*.

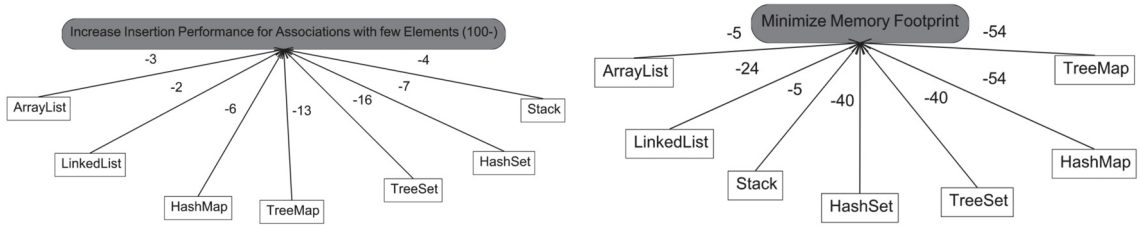
Collections can have a maximum and a minimum number of entries. *Maximum* is used when the upper bound of the multiplicity is greater than `1` and not infinitely many. It affects the behaviour of the insertion operations, because they have to ensure that the maximum was not reached before adding. *Minimum* is used when the lower bound of the multiplicity is greater than `0`. It affects the removal operations, since they need to check that the

[1] UML uses the star character to represent an unlimited upper bound and it is usually referenced as many or infinitely many.

minimum size restriction is not violated before removing. The *Minimum* and *Maximum* features are children of *Plain*, not children of *Many*, in order so they cannot be selected with the *KeyIndexed* subtree. This decision was made because UML does not support any restriction on the number of keys and neither do the implementation classes offered. If the modeller wants to restrict the number of keys, he could add behaviour to do so or use an enumeration as key [20, p. 201]. Following UML standards again, the multiplicity of a qualified association corresponds to the number of elements per key and not to the number of keys. However, the current supported key indexed implementation classes only allow one element per key. If the modeller requires more elements to be mapped with a key, he should use a collection as the value. Support for this could however be added in the future.

3.1.2 Impact Model

Different implementation choices behave differently at run-time, and a concern captures such non-functional properties in the impact model. The *Association* concern provides impact models for 5 goals: *Increase Insertion Performance*, *Increase Iteration Performance*, *Increase Random Access Performance*, *Increase Removal Performance* and *Minimize Memory Footprint*. For each performance related goal, there are 4 sub-goals for different collection sizes. For example, Figure 3–2a shows the impact model for *Increase Insertion Performance for 1,000 elements*. The values are negative because in increasing the performance, the higher the value the better, however, the lower the time it takes to perform the operation, the better. Adding *1,000* elements in an `ArrayList` takes less time than adding them in a `TreeSet`, therefore, `ArrayList` increases the performance and the contribution is higher. Figure 3–2b shows the *Minimize Memory Footprint* impact model. The values are negative for the same reason. We can observe that lists occupy less space and maps occupy more.



(a) *Increase Insertion Performance* (b) *Minimize Memory Footprint*

Figure 3–2: Impact Models of the *Association* Concern

The values of the contributions result from benchmarks we have run on Java collections. We describe those benchmarks in Chapter 5.

3.2 Customization Interface

The *customization interface* is defined by the elements from the concern that need to be mapped to elements in the application. In an association, we need to define the classes of origin and destination, i.e., the class that holds one association end and the class that holds the opposite end. We call `|Data` the class of origin and `|Associated` the class of destination as shown in Figure 3–3a. The pipe prefix ('|') is a convention in TouchCORE to specify elements of the customization interface. When the selection is a key-indexed collection, the customization interface varies as can be seen in Figure 3–3b. It is still composed of `|Data` but also of `|Key` and `|Value`. `|Data` is still mapped to the class of origin. Similarly to `|Associated`, `|Value` is mapped to the destination of the association and `|Key` is mapped to the qualifier.

(a) Common Customization Interface (b) KeyIndexed Customization Interface

Figure 3–3: Customization interface

3.3 Usage Interface

The *usage interface* is defined by the public elements in the concern that can be used by the application. In the case of the *Association* concern, the *usage interface* is composed of `|Data` and its public operations. The features from the concern do not have a common *usage interface* as the operations of `|Data` vary with the properties of the collection. When a class holds a single object reference, i.e., the feature *One* is selected, the *usage interface* consists of a getter and a setter. When it holds a reference to a collection, it has more operations. If it is a plain collection, it contains operations to add and remove elements, and if the collection is ordered, it additionally has operations to add and remove at a specific index as shown in Figure 3–4 when selecting the *ArrayList* feature. If it is a key-indexed collection, it contains operations to add a mapping with a key and a value and to remove the mapping associated with a key.

The operations of `|Data` are not part of the customization interface, i.e., they *do not have to be* mapped to operations in the target model. Once `|Data` is mapped to the class holding the association, the operations belonging to the usage interface are added to the class and may be used. However, users may want to rename the operations for better usability, for example, rename `add` to `addToMyObservers`. To do so, they would create an operation with this desired name and map the operation from `|Data` to it.

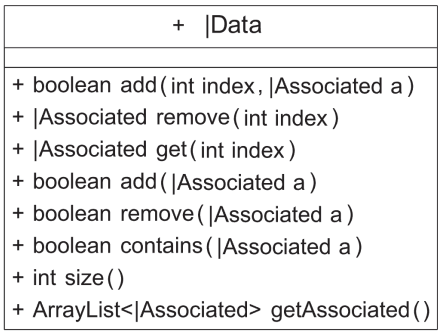


Figure 3–4: The Usage Interface of the *Association* Concern when Selecting Feature *ArrayList*

3.4 Structural Realization

Models associated with features are called *realization models*. We say that a model realizes a feature and it typically realizes one feature. The *realization model* of the root feature *Association* defines the two classes `|Data` and `|Associated`. They are marked as *public partial* with a vertical bar '|', which means that they are part of the customization interface of the model, and need to be completed by the application.

The realization model of the feature *One* is shown in Figure 3–5a, which is selected when the upper bound of the multiplicity of the association is `1`. It defines an association between `|Data` and `|Associated` with multiplicity `1` (lower bound is `1` and upper bound is `1`), as well as a getter and a setter for the association end. The lower bound of the multiplicity could also be `0` when reusing the concern. However, users may change this lower bound to `1`.

The realization model for *Many* in Figure 3–5b defines an additional class for the collection called `¦CollectionOfAssociated`. It is marked as *concern partial* prefixed with a discontinuous vertical bar '¦', which means that it is also incomplete, but needs to be further defined *within* the concern. Therefore, when a complete selection is made, i.e., a leaf feature of the *Many* subtree is selected, a concrete implementation class needs to have been mapped to the `¦CollectionOfAssociated` class. The operations of `¦CollectionOfAssociated` are also concern partial. Since `¦CollectionOfAssociated` and its operations are mapped inside the concern, they are not part of the customization interface. The feature *Many* also defines the association between `|Data` and `|Associated`, but this time the multiplicity has a lower bound of `0` and an upper bound of many. The associations from `|Data` to `¦CollectionOfAssociated` and from `¦CollectionOfAssociated` to `|Associated` are defined in the *Many* feature since they are shared by all of its children. `|Data` always has one

collection, but the number of collections depends on the application and is overwritten when reused. The model also provides the operations of `|Data` shared by all children:

- `int size()` that returns the number of elements currently contained in the collection.
- `boolean contains(Object)` that returns true if the given object is contained in the collection.
- `¦CollectionOfAssociated getAssociated()` that returns the entire collection.

The features between *Many* and its leaves provide additional operations to the `|Data` class. The *Plain* feature, shown in Figure 3–5c, defines the operations shared by ordered and unordered collections:

- `boolean add(|Associated)`that adds the given element to the collection.
- `boolean remove(|Associated)` that removes the given element from the collection.

The *Ordered* feature (see Figure 3–5d) defines the operations shared by all ordered collections:

- `boolean add(int, |Associated)` that adds an element at the specified position in the collection.
- `|Associated remove(int)` that removes an element at the specified position in the collection.
- `|Associated get(int)` that returns the element at the specified position in the collection.

The *Unordered* feature does not provide any additional operations to `|Data`. Of course, the list of operations is not exhaustive. The number of operations for Java collections is far greater, but for now we decided to only use a subset of most commonly used operations.

The *KeyIndexed* feature realizes the qualified associations. It contains a class `|Key` and the class `|Associated` is renamed to `|Value`, since this is the name typically used to refer

to the entities that are reachable with a key. To perform the renaming, a class `|Value` is created, and the realization model extends *Many* and maps `|Associated` to `|Value`. The class `|Data` in *KeyIndexed* holds all common operations related to any kind of maps such as `put`, `containsKey and containsValue`.

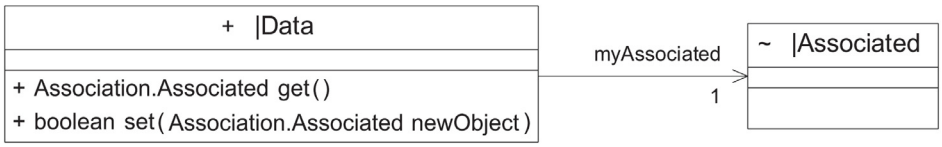
In leaf features, for example *ArrayList* in Figure 3–5f, a concrete implementation class is imported as well as its operations. The mappings of the class and the operations are established as shown in Figure 3–5e. All concern partial elements are mapped at this point.

None of the optional features of the variation interface have an impact on the realization structure, i.e., they do not add classes or update the list of operations, but they have an impact on the behavioural structure of the operations. Therefore they are described in the next section.

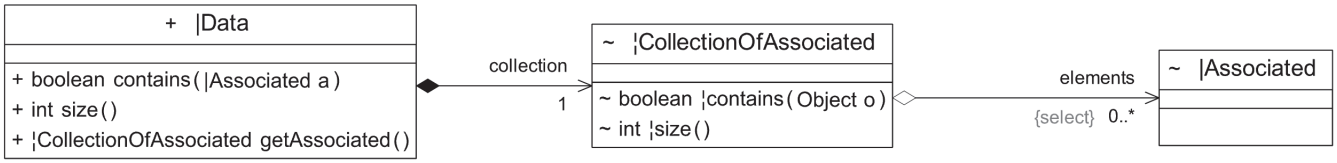
3.5 Behavioural Realization

As we saw in the realization model of the feature *Plain* in Figure 3–5c, `|Data` has an operation `add` and the `¦CollectionOfAssociated` has a concern partial operation with the same signature. The concern partial operation is mapped to a concrete operation of an implementation class in a leaf feature as previously shown in Figure 3–5e. Behaviours are always and only defined for the operations of `|Data.`

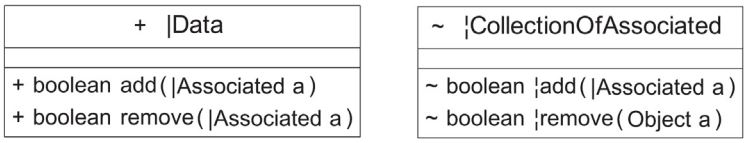
In a concern, the behaviour of an operation is defined using a sequence diagram called a *message view* in the tool TouchCORE. As shown in Figure 3–6, a call to the `add` operation on `|Data` triggers a call to the `add` operation on the `¦CollectionOfAssociated` and returns `true` if the element was successfully added. Operations in `|Data` that have an equivalent operation in the collection, with the same name and signature, have a similar behaviour: they forward the call to the operation of the collection with the corresponding parameters.



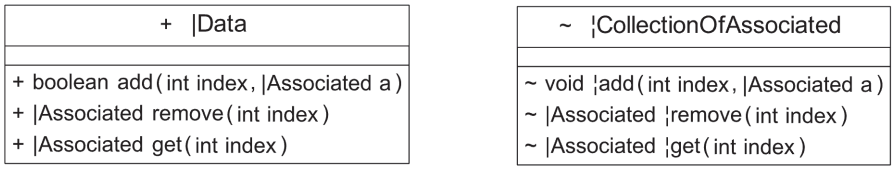
(a) Realization Model *One*



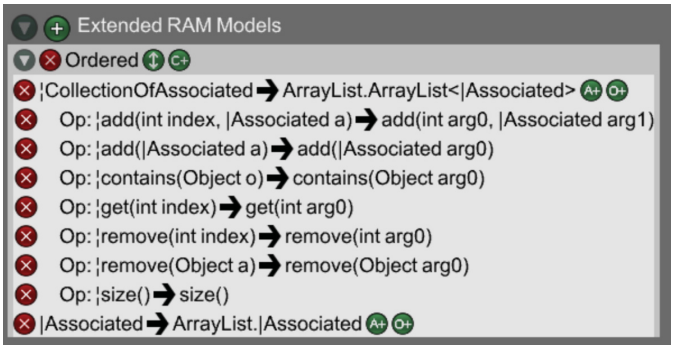
(b) Realization Model *Many*



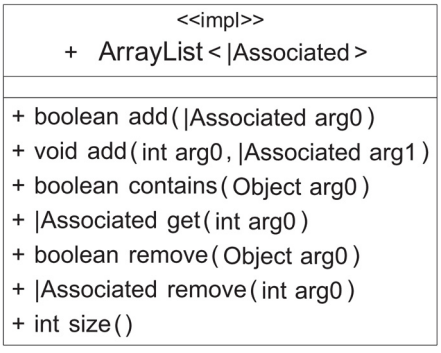
(c) Realization Model *Plain*



(d) Realization Model *Ordered*



(e) Mappings to the `ArrayList` Implementation Class



(f) Realization Model *ArrayList*

Figure 3–5: Realization Models of the *Association* Concern

Some features may affect the behaviour of other features. In the variation interface, there are four optional features: *Unique, Maximum*, *Minimum* and *Bidirectional*. When they are selected, they impact operations by specifying additional behaviour. *Unique* impacts the behaviour of *insertion* operations: before adding, a check is performed to determine whether the element is already in the collection. *Maximum* also impacts insertion operations: if the maximum is already reached, the operation returns `false` and the addition is not performed. *Minimum* impacts *removal* operations: if the collection already contains the minimum number of elements, it returns `false` and the element is not removed. *Bidirectional* ensures referential integrity. It impacts *constructors*, *setters*, *insertion* and *removal* operations. When an element is added to a collection and the association is bidirectional, depending on whether the opposite side is one or many, the element needs to be set or added on the opposite side.

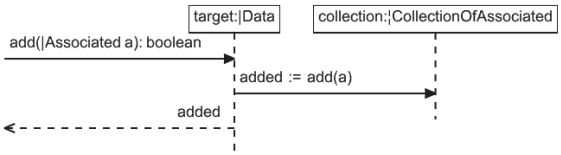


Figure 3–6: Message View of the `add` Operation

We use aspect-oriented techniques to augment the behaviour of operations—called advising in aspect-oriented terminology—using *aspect message views*. For example, the realization model of the feature *Maximum* defines an aspect message view *checkMaximum,* shown in Figure 3–7, that adds behaviour to an existing operation, whose behaviour is represented by the white box. It verifies whether the collection already contains the maximum number of elements before adding. Advising an operation from an extended model through an aspect message view is done using *message view references* 2.2.2. *Maximum* is a child feature of *Many*, therefore it has access to the `add` operation of `|Data`. In Figure 3–7, the message view reference specifies that the `add(|Association a)` operation defined in the *Plain* feature is

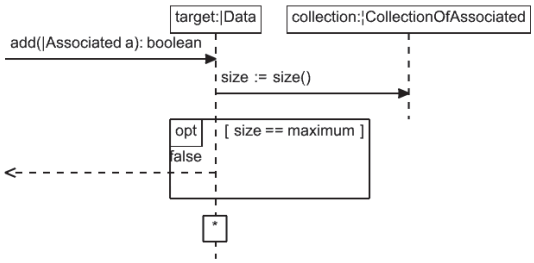


Figure 3–7: Aspect Message View *checkMaximum*

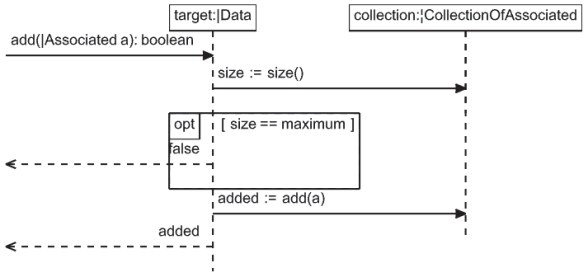


Figure 3–8: Message View of the `add` Operation in the Woven Aspect *Maximum*

affected by the aspect message view *checkMaximum*. As a result, when the feature *Maximum* is woven in, the message view of the operation `add` has its original behaviour wrapped in the behaviour of the aspect message view, which is shown in Figure 3–8.

3.6 Feature Interaction Realization

In a concern, if feature A is realized by realization model M_{A} and feature B is realized by realization model M_{B}, when features A and B are both selected, simply combining models M_{A} and M_{B} does not always yield in a correct realization model that provides the combined functionality of A and B. This situation is called a *feature interaction*. To resolve it, another realization model called *conflict resolution model* M_{AB} needs to be defined. In the context of the *Association* concern, several feature interactions have been identified, and hence there are many conflict resolution models, which are shown in Table 3–1. Conflict resolution models may redefine the structure as well as the behaviour. In our current concern, they only impact the behaviour of existing operations, and few of them add additional protected operations.

3.6.1 Behavioural Adaptation to Ensure Referential Integrity

Within the *Association* concern, the first reason for using conflict resolution models is that in order to ensure referential integrity, the standard behaviour of the operations provided

	One	Plain	Ordered	KeyIndexed
Bidirectional	-	PlainBidirectional	OrderedBidirectional	KeyIndexedBidirectional
OneOpposite	OneToOne	PlainToOne	OrderedToOne	KeyIndexedToOne
PlainOpposite	OneToPlain	PlainToPlain	OrderedToPlain	KeyIndexedToPlain
OrderedOpposite	OneToOrdered	PlainToOrdered	OrderedToOrdered	KeyIndexedToOrdered
KeyIndexedOpposite	OneToKeyIndexed	PlainToKeyIndexed	OrderedToKeyIndexed	-

Table 3–1: Bidirectional Conflict Resolution Aspects

by `|Data` needs to be adapted depending on the specific combination of properties of the two association ends. The answers to the following questions determine what should be the correct behaviour: is the association bidirectional? If yes, is the opposite side a single object or a collection of objects? Based on the answers, the behaviours of a *set*, *insertion* or *removal* operation are different as they need to update the opposite side of the association.

Consider the following bidirectional associations between class A and class B with objects a_1, b_1 and b_2:

- `1-1`: When calling a_1`.set(`b_1`)`, a_1 needs to update its reference, but also needs to ensure b_1's reference is updated by calling b_1`.set(`a_1`)`.
- `1-0..*`: When calling a_1`.set(`b_1`)`, a_1 needs to update its reference, but also needs to ensure b_1 has a reference to a_1 by calling b_1`.add(`a_1`)`. Similarly, when calling a_1`.set(null)`, which unsets a_1's reference to b_1, b_1 needs to remove its reference to a_1, and therefore a_1 needs to call b_1`.remove(`a_1`)`. Calling a_1`.set(`b_2`)` when a_1's reference is already set to b_1 is equivalent to calling a_1`.set(null)` first.

- `0..*-0..*`: When calling a_1`.add(`b_1`)`, a_1 needs to add b_1 to its collection and needs to call b_1`.add(`a_1`)`, so that b_1 also adds a_1 to its collection. Similarly, calling a_1`.remove(`b_1`)` triggers it to call b_1`.remove(`a_1`)`. Note that if the collection is of type *KeyIndexed*, the operation `put` needs to be called instead of `add`.

From those cases, we can see that the behaviour to ensure referential integrity could lead to infinite recursion. A `set` operation in a_1 triggers a `set` operation in b_1, which in turn triggers another `set` operation in a_1, and so on. To avoid this, we need to keep track of who first initiated the process in order to perform the operation only once on each side. For example, a `set` operation could check that the reference is actually being updated, i.e, given a different object (or not setting to `null` if it was already `null`). However, with a non-unique collection, it is impossible to verify if the object was already just added or not, because it could be found in the collection from a previous add, therefore we need another solution which we will now explain further.

Conflict resolution models are required for all combinations, e.g., One-to-One (*One* and *OneOpposite*), One-to-Plain (*One* and *PlainOpposite*), Plain-to-One (*Plain* and *OneOpposite*), Plain-to-Plain (*Plain* and *PlainOpposite*), etc. The exhaustive list is shown in Table 3–1.

We describe here some conflict resolution models to explain how referential integrity is dealt with within the *Association* concern. We present the conflict resolution model*s* (CRM) *OneToOne*, *OneToPlain* and *PlainToOne*. To understand how they work, we also describe the realization models (RM) *OneOpposite* and *PlainOpposite:*

OneOpposite [RM] in Figure 3–9 realizes only the feature *OneOpposite* and is extended by conflict resolution models where the opposite association end is a single reference.

It defines the `setSimple` operation in `|Associated,` which is a helper operation for dealing with referential integrity. It should not be available to the outside world, and therefore its visibility is set to *protected* (#). The behaviour of `setSimple` is not defined in *OneOpposite* as it depends on information about the original end. We will be looking at the two different behaviours of `setSimple`, one is defined in the realization model *OneToOne* (see Algorithm 2) and the other one is defined in the realization model *PlainToOne* (see Algorithm 6). They differ because the association end of origin is either a single reference or a collection.

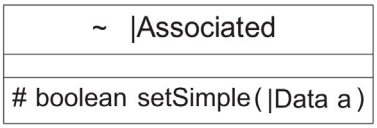


Figure 3–9: Realization Model *OneOpposite*

OneToOne [CRM] realizes two features, *One* and *OneOpposite,* and extends the realization models *One* and *OneOpposite*. The realization model *One* (see Figure 3–5a) defines a getter and a setter in `|Data` and *OneOpposite* (see Figure 3–9) defines the operation `setSimple` used as a helper method for dealing with referential integrity. In the model *OneToOne,* the `set` operation of `|Data` is affected by the aspect message view *setOppositeWhenSetting*, shown in Algorithm 1, to deal with the bidirectional association. Also, a behaviour is defined for `setSimple`, as shown in Algorithm 2, as we now know that the opposite reference is a single object. Let's look at an example.

Consider two classes A and B with a `0..1-0..1` bidirectional association between the two. Looking at Figure 3–10, we have four objects $\mathtt{a}_1$, $\mathtt{a}_2$, $\mathtt{b}_1$ and $\mathtt{b}_2$ where $\mathtt{a}_1$ and $\mathtt{b}_1$ are associated and $\mathtt{a}_2$ and $\mathtt{b}_2$ are associated as in step 1. Now, we call $\mathtt{a}_1\mathtt{.set(b}_2\mathtt{)}$. In

Algorithm 1 Aspect Message View *setOppositeWhenSetting* to advise `set(newAssociated)` in feature *OneToOne*

```
if newAssociated is not null then
  if oldAssociated is not null then
    oldAssociated.setSimple(null)
  end if
end if
newAssociated.setSimple(this)
[original behaviour of set]
```

Algorithm 2 Message View `boolean setSimple(newAssociated)` in feature *OneToOne*

```
if newAssociated is not null then
  if oldAssociated is not null then
    oldAssociated.setSimple(null)
  end if
end if
myAssociated ← newAssociated
return true
```

step 2, a_1 calls `b1.setSimple(null)`, which unsets b_1's reference to a_1 (and note that b_1 does not try to unset a_1's reference to b_1). In step 3, a_1 calls `b2.setSimple(a1)` triggering `a2.setSimple(null)` that unsets a_2's reference. Then, b_2 sets its reference to a_1 in step 4. And in step 5, a_1 finally sets its reference to b_2.

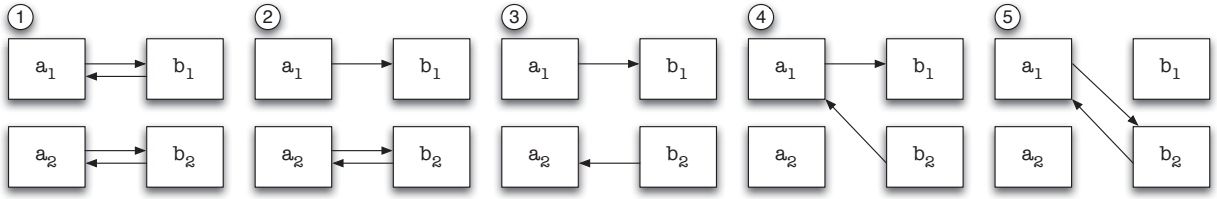


Figure 3–10: Steps When Updating a Reference

PlainOpposite [RM] in Figure 3–11 realizes only the feature *PlainOpposite* and is extended by conflict resolution models where the opposite association end is a plain collection such as *OneToPlain*, *PlainToPlain*, *OrderedToPlain*, etc. It provides the operations `addSimple` and `removeSimple` for `|Associated`. The `addSimple` operation simply adds the elements to the collection without taking care or referential integrity. It is a helper that is never called directly by the user. It is only called when a *set* or an

insert is performed on its opposite end to avoid infinite recursion. The `removeSimple` operation works the same way for removing an element from the collection, it simply removes without dealing with any other references.

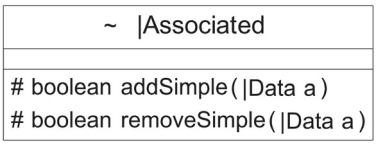


Figure 3–11: Realization Model *PlainOpposite*

OneToPlain [CRM] realizes two features, *One* and *PlainOpposite,* and extends the realization models *One* and *PlainOpposite*. The realization model *One* defines a getter and a setter in `|Data` and *PlainOpposite* defines the `addSimple` and `removeSimple` of `|Associated`. In *OneToPlain*, the aspect message view *referenceOppositeWhenSetting* in Algorithm 3 is defined. It is used to advise the `set` operation of `|Data`. Consider the association A `1-0..*` B with objects $\mathtt{a}_1$, $\mathtt{b}_1$, $\mathtt{b}_2$ where $\mathtt{a}_1$ and $\mathtt{b}_1$ are associated. The `set` operation is triggered by calling $\mathtt{a}_1$`.set(`$\mathtt{b}_2$`)`, $\mathtt{a}_1$ first calls $\mathtt{b}_1$`.removeSimple(`$\mathtt{a}_1$`)` and then calls $\mathtt{b}_2$`.add(`$\mathtt{a}_1$`)`. Finally, $\mathtt{a}_1$ updates its reference from $\mathtt{b}_1$ to $\mathtt{b}_2$.

PlainToOne [CRM] realizes two features, *Plain* and *OneOpposite,* and extends the realization models *Plain* and *OneOpposite*. In the realization model *PlainToOne,* two aspect message views are defined: *setOppositeWhenAdding* that advises the `add` operation on `|Data` and *setOppositeWhenRemoving* that advises the `remove` operation on `|Data`, as shown in Algorithm 4 and Algorithm 5 respectively. When adding an element to the collection, the reference of this element is set and when removing an element, its reference is unset. It also defines the behaviour for `setSimple` in Algorithm 6. In *PlainToOne*, we know `setSimple` is being called from either an `add` or a `remove` operation

Algorithm 3 Aspect Message View *referenceOppositeWhenSetting* to advise `set(newAssociated)` in feature *OneToPlain*

```
if oldAssociated is not null then
  done ← oldAssociated.removeSimple(this)
  if not removed then
    return false
  end if
end if
if newAssociated is not null then
  newAssociated.addSimple(this)
  if not removed then
    return false
  end if
end if
[original behaviour of set]
```

(again, never from the user). It needs to remove itself from a previous list when being added in a new one in case its previous reference was already set. Consider another association A `0..*-1` B with non-associated objects a_1 and b_1. Calling a_1`.add(`b_1`)` triggers a_1 to call b_1`.setSimple(`a_1`)`. Similarly, calling a_1`.remove(`b_1`)` triggers a_1 to call b_1`.setSimple(null)`.

Many-to-Many conflict resolution models handle referential integrity in the same manner: an `add` operation triggers an `addSimple` and a `remove` operation triggers a `removeSimple`. *KeyIndexedToKeyIndexed* is not supported. When an element is added in a map it requires a key, keeping the referential integrity would require a second key. When putting a value in one map by specifying the key and the value, it would need the opposite key to put on the opposite map. Therefore, the user would have to specify two keys when putting an element in a map. We believe this complicates usability and therefore do not support such a scenario.

Algorithm 4 Aspect Message View *setOppositeWhenAdding* to advise `add(newAssociated)` in feature *PlainToOne*

```
if newAssociated is null then
  return false
end if
set ← newAssociated.setSimple(this)
[original behaviour of add]
```

Algorithm 5 Aspect Message View *setOppositeWhenRemoving* to advise `remove(oldAssociated)` in feature *PlainToOne*

```
if oldAssociated is null then
  return false
end if
set ← oldAssociated.setSimple(null)
[original behaviour of remove]
```

Algorithm 6 Message View for `boolean setSimple(newAssociated)` in feature *PlainToOne*

```
if oldAssociated is not null then
  oldAssociated.removeSimple(this)
  if not removed then
    return false
  end if
end if
myAssociated ← newAssociated
return true
```

3.6.2 Unique, Maximum and Minimum

Unique, *Maximum*, and *Minimum* are also optional features that need conflict resolution aspects. *Unique* and *Maximum* impact the *insertion* operations and *Minimum impacts* the *removal* operations. *Unique* is also selectable with a key-indexed collection, therefore it also impacts the operations that insert elements for a given key. We created *Unique*, *Maximum*, *Minimum* models that define the aspect message views *checkUnique*, *checkMaximum* and *checkMinimum* respectively. The aspect message view *checkUnique* verifies that the element is not already contained in the collection, *checkMaximum* ensures the maximum was not reached before adding and *checkMinimum* ensures the minimum size restriction is not being violated before removing an element. The aspect message views are applied through message view references, because they advise operations from extended models. The operations are advised in the realization models as follows:

- `add(|Associated a)` is advised in *Maximum* and *UniquePlain*
- `remove(|Associated a)` is advised in *Minimum*
- `add(int index, |Associated a)` is advised in *MaximumOrdered* and *UniqueOrdered*
- `remove(int index)` is advised in *MinimumOrdered*
- `put(|Key k, |Value v)` is advised in *UniqueKeyIndexed*

Minimum is not selectable with *One* even if a multiplicity with a lower bound of `1` and an upper bound of `1` is possible. A unidirectional association end with a multiplicity of `1` requires a constructor that takes as parameter the reference to set, and prevents setting the reference to `null`. However, a bidirectional association with at least one end with a minimum multiplicity of `1` implies many more constraints [16]. When one object is created, the constructor needs to immediately assign or create the other one to be able to reference

it right away. Also, the `set` operation cannot update the reference to `null`, and if it needs to update to a new object, it needs to set the opposite reference to `null`, which is not possible. We believe these are unpractical constraints and therefore, we never enforce the minimum multiplicity of `1`. Similarly, with a many association with a minimum constraint, we do not enforce this constraint when creating the object. Once the collection has reached its minimum capacity, it cannot be violated anymore. The user needs to ensure to have a valid model once the updates are done. This could be enforced by the tool through protocol models.

3.6.3 Combinations

When optional features are simultaneously selected and as a result, an operation is advised by more than one aspect message view, the resulting behaviour depends on the order in which the models are woven. This works well for *Unique* and *Maximum*, since the behavioural modifications they perform are independent from one another. They both advise *insertion* operations, and any order of application results in a valid message view. *Minimum* does not augment the same operation as *Unique* and *Maximum*, therefore there is no conflict between them.

However, the order matters when selecting *Bidirectional* in combination with any other optional feature. In the message view, it needs to do the maximum/unique checks before adding references on the opposite side. To ensure the right ordering, we had to define conflict resolution models for all combinations of *Bidirectional* and *Maximum*/*Minimum*/*Unique* as listed in Table 3–2.

The bidirectional realization model is extended first, and as a result it will be applied first when weaving. Hence, its behaviour is inserted right before the original behaviour of the

	Maximum	Minimum	Unique
Plain	Maximum	Minimum	UniquePlain
PlainBidirectional	MaximumPlainBirectional	MinimumPlainBirectional	UniquePlainBirectional
PlainToOne	MaximumPlainToOne	MinimumPlainToOne	UniquePlainToOne
PlainToPlain	MaximumPlainToPlain	MinimumPlainToPlain	UniquePlainToPlain
PlainToOrdered	MaximumPlainToOrdered	MinimumPlainToOrdered	UniquePlainToOrdered
PlainToKeyIndexed	MaximumPlainToKeyIndexed	MinimumPlainToKeyIndexed	UniquePlainToKeyIndexed
Ordered	MaximumOrdered	MinimumOrdered	UniqueOrdered
OrderedBidirectional	-	-	-
OrderedToOne	MaximumOrderedToOne	MinimumOrderedToOne	UniqueOrderedToOne
OrderedToPlain	MaximumOrderedToPlain	MinimumOrderedToPlain	UniqueOrderedToPlain
OrderedToOrdered	MaximumOrderedToOrdered	MinimumOrderedToOrdered	UniqueOrderedToOrdered
OrderedToKeyIndexed	MaximumOrderedToKeyIndexed	MinimumOrderedToKeyIndexed	UniqueOrderedToKeyIndexed
KeyIndexed	-	-	UniqueKeyIndexed
KeyIndexedBidirectional	-	-	UniqueKeyIndexedBidirectional
KeyIndexedToOne	-	-	UniqueKeyIndexedToOne
KeyIndexedToPlain	-	-	UniqueKeyIndexedToPlain
KeyIndexedToOrdered	-	-	UniqueKeyIndexedToOrdered

Table 3–2: *Maximum, Minimum* and *Unique* Conflict Resolution Aspects

operation. Figure 3–6, seen previously, shows the message view of `add` when it is not advised, Figure 3–12 shows the message view of `add` when it is advised by *setOppositeWhenAdding* to ensure referential integrity, and finally Figure 3–13 show the message view when it is also advised by *checkMaximum*. Due to the conflict resolution model, the check for maximum is happening first.

Creating a large number of conflict resolution models seems like a lot of work, but most of those conflict resolution models are easily created, since they only need to extend other models in a specific order and do not contain any new structure or behaviour. With the current *Association* concern, there are 225 possible selectable feature combinations. To cover those combinations, we built 72 models out of which 30 were empty and just extended other models. While this shows that building a concern is time consuming, the fact that we did not need to define conflict resolution models for all possible combinations still represents a reduction in development effort. Moreover, this work was done once in the scope of this thesis, and from now on the concern is ready to be used in any situation without the modeller having to deal with these intricate feature dependencies.

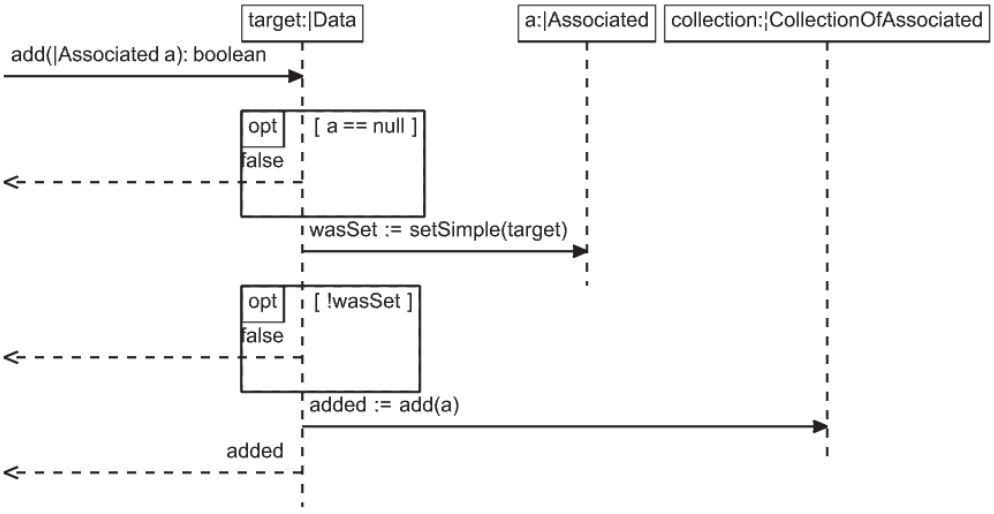


Figure 3–12: Message View of the `add` Operation in *PlainToOne*

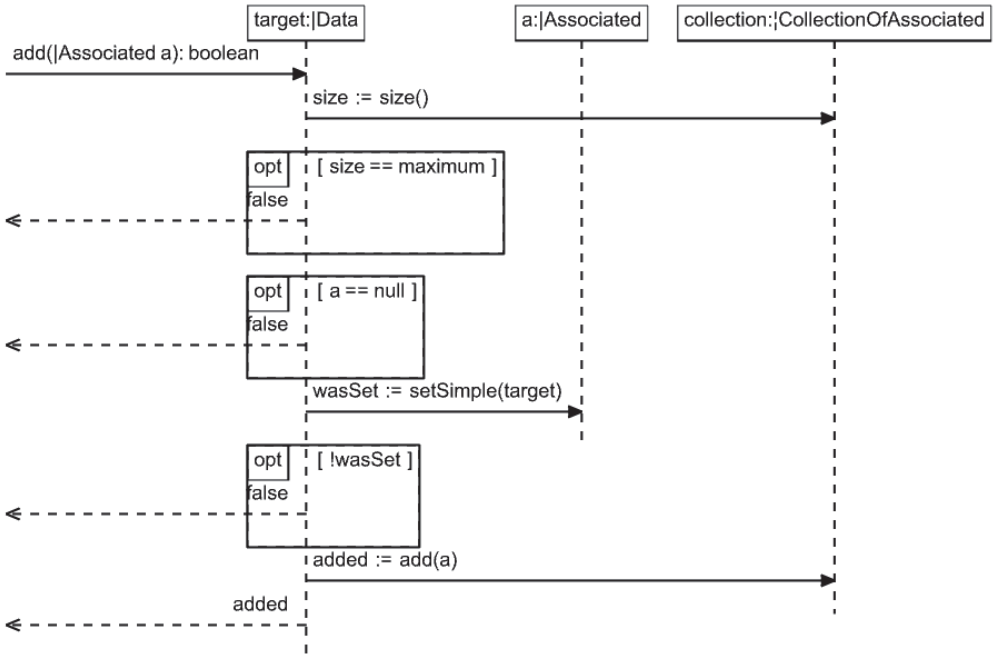


Figure 3–13: Message View of the `add` Operation in *MaximumPlainToOne*

Chapter 4
Simplifying the Use of the Association Concern

Chapter 3 described the detailed design of the *Association* concern with many possible variations and with intricate behaviour to ensure the desired properties maximum, minimum, uniqueness, and bidirectionality. Although the design of the *Association* concern required considerable effort, the design knowledge is now encapsulated behind the variation, customization and usage interfaces, and available to be reused.

Associations are a very common concept in software design modelling. They are widely used, and hence, it is important to streamline the reuse of the *Association* concern as much as possible. The standard CORE reuse process, described in Section 2.1.2, which is defined for reusing *any concern,* can also be used for reusing the *Association* concern. It is explained in more detail in Section 4.1 of this chapter. Unfortunately, due to its generic nature, the process is unnecessarily tedious and, in this case, potentially error prone for the modeller. We therefore decided to devise a domain-specific language (DSL) for reusing the *Association* concern, which is inspired by the UML notation for associations. This DSL is described in Section 4.2. Details about the implementation of this DSL in the TouchCORE tool are presented in Section 4.3.

4.1 Reusing Association with the Standard CORE Reuse Process

Before the creation of the *Association* concern, the user could draw an association between two classes as seen in Figure 2–3 of the background chapter. When doing so, the code generator would only create a reference in the class if the multiplicity was `1` and it would

create an `ArrayList` for an association end of multiplicity many. Moreover, none of the operations were created to add or remove elements to the collection.

Now that a concern was designed, the user may reuse the *Association* with the standard CORE reuse process. This section, however, elaborates on the effort that it takes for the modeller to reuse the concern through this process, and then discusses mistakes that the user could make to highlight the need for a streamlined interface for reusing the *Association* concern. For every association end the following happens:

1. The modeller indicates a desire to reuse *Association*. This involves browsing through the reusable concern library to find the *Association* concern, which typically involves navigating down the folder hierarchy.
2. Once the variation interface is displayed, the modeller must make a selection of the desired variant. The feature model is large and it takes cognitive effort to visually browse through it.
3. Once the selection is done, the *customization interface* for the desired variant is displayed, as shown for the *Observer* concern in Figure 2–5b. Now, the modeller has to manually establish the mappings of the source and destination classes of the association as presented in Figure 4–1. This means that `|Data` and `|Associated` have to be mapped. The mappings of the operations are not necessary, because they are not part of the customization interface. However, due to the fact that mappings have to be used to rename the generic names of operations to more specific names, e.g., `add` to `addToMyObservers`, it leads modellers most of the time to specify mappings for the operations nevertheless. Furthermore, whenever two associations connect the same classes, operations must be mapped to prevent duplicate operation signatures. Again,

establishing these mappings manually is tedious and time consuming, especially since the list of operations provided by `|Data` for some of the variants of the *Association* concern can be quite extensive.

4. There is no visualization in the class diagram for the association when using textual mappings. Therefore, the user could still draw it and this could lead to inconsistencies.
5. Finally, a bidirectional association requires two manual reuses. This not only constitutes a duplication of effort, but is also fairly unintuitive. Without a DSL, we cannot enforce that both directions have a reuse associated to it.

As detailed above, reusing the *Association* concern with the standard CORE reuse process is very time consuming and unintuitive. In addition, the association is represented by the reuse and textual mappings, and is not linked to the visual representation of the association shown in the class diagram built with TouchCORE. In Figure 4–1, the association was drawn independently from the reuse. If there were two associations drawn between `|Subject` and `|Observer`, there would be no way to distinguish which one the reuse was referring to. Therefore, it would be possible to have associations drawn between classes without the *Association* concern being reused, or reusing the concern without any visual representation of the *Association* linking the classes. This is confusing to the modeller and in the worst case could again lead to inconsistencies in the model.

In light of these problems, we streamline the process for the users to minimize their effort when reusing *Association* and to eliminate any risk of misuse. In MDE, in general, applying domain knowledge is done through Domain-Specific Languages. The next section describes how we defined our own DSL [30] for applying the *Association* concern, which is inspired

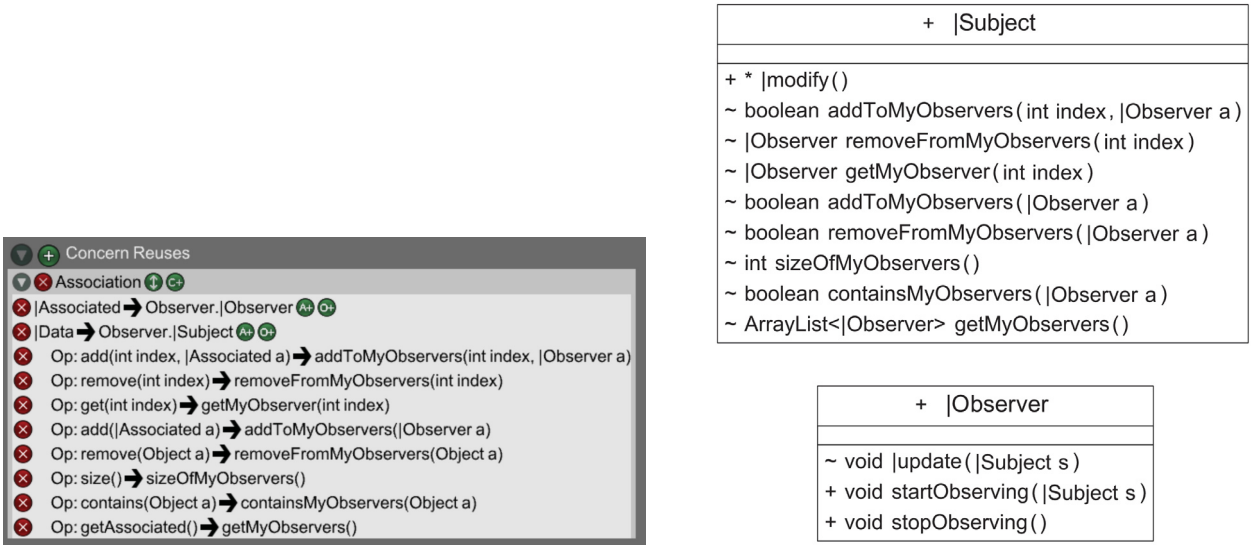


Figure 4–1: Association through Concern Reuse and Mappings

by the concrete syntax of UML associations. Section 4.3 then presents how this DSL was implemented in the tool.

4.2 A DSL for Applying Association

UML already defines a visual notation for associations. Therefore, we first looked into what UML offers and extended the notation where necessary to better fit our situation.

4.2.1 Associations in UML

To come up with a better solution for reusing the *Association* concern, we looked into the UML specification [20]. UML already provides visual elements that allow the modeller to express some of the desired properties of an association, e.g., arrows for navigability, boxes for qualified associations and multiplicity elements. The multiplicity elements are of greater interest to us as they cover most of the features offered in the variation interface. UML describes their representation as follows [20, p. 34]:

> "*The specific notation for a MultiplicityElement is defined for each concrete kind of MultiplicityElement. In general, the notation will include a multiplicity specification, which is shown as a text string containing the bounds of the multiplicity and a notation for showing the optional ordering and uniqueness specifications.*"

A multiplicity element contains *isOrdered* and *isUnique* attributes. For qualified associations, UML handles them as follows [20, p. 201]:

> "*A qualifier is shown as a small rectangle attached to the end of an association path between the final path segment and the symbol of the Classifier that it connects to. The qualifier rectangle should be smaller than the attached class rectangle, unless this is not practical. The qualifier rectangle is part of the association path, not part of the Classifier. The qualifier rectangle is attached to the end of the association path that represents the memberEnd that owns the qualifier.*"

An example of the UML notation with the *isOrdered* attribute set to `true` and an example of the notation for a qualified association are shown in Figure 4–2a. From the documentation, we conclude that UML handles visualization for the multiplicities, the ordering and uniqueness specification and the qualified associations. Looking at Table 4–1, we conclude that UML handles most of the features offered in the concern, but is missing low level selections for specific implementation choices.

4.2.2 The proposed DSL

From the multiplicity specifications proposed by UML, order-designators and uniqueness-designators, it is possible to infer a data type for the implementation of the collection. For

Feature from the *Association* concern	UML notation
Minimum	Lower bound of multiplicity > 0
Maximum	Upper bound of multiplicity > 1 and not *
Bidirectional	Both ends with arrows $\rightarrow$
Unique	*isUnique* attribute on the multiplicity element
Ordered	*isOrdered* attribute on the multiplicity element
Unordered	*isOrdered* attribute on the multiplicity element
KeyIndexed	Small rectangle attached to the end of an association path
ArrayList	-
LinkedList	-
Stack	-
HastSet	-
TreeSet	-
HashMap	-
TreeMap	-

Table 4–1: UML Notations to Represent the Features of the *Association* Concern

(a) UML Notation

(b) Proposed Extension of UML

Figure 4–2: Comparison between UML and our DSL

example, if *unique* and not *ordered*, the collection is a set. However, it does not allow expressing an implementation choice (e.g., *TreeSet* vs. *HashSet*). Since impacts on system qualities are available in the feature model, we want users who know their system requirements to be able to make a complete selection by selecting a specific implementation data structure. Therefore, multiplicity specifications in our DSL also include implementation specification as shown in Figure 4–2b.

When a user draws an association, we know the class of origin and the class of destination. As they are the only elements of the customization interface, the mappings can be deduced automatically. Therefore, we propose an automatic reuse of the concern. Feature selection is also done automatically to some extent as we are aware of the navigability and multiplicity bounds of each association end. However, to decide on an implementation class, the user still needs to be presented with the variation interface to make a decision. To do so, he simply has to double click on the multiplicity element and the feature model is displayed. We describe in the next section how this was implemented.

4.3 Implementing the DSL in TouchCORE

The DSL described in the previous section has been implemented in TouchCORE and is now ready to use. This section presents an overview of the steps involved in updating the tool:

the modifications to the RAM metamodel and to the RAM weaver, the automated generation of mappings and feature selection, and the necessary updates to the GUI. This involved modifying the association visualization to support the DSL graphically. The last subsection describes further UI changes to increase usability by hiding automatically created operations unless they are used, by hiding the standard visual representation of the *Association* reuse from the concern reuses box, and by moving the concern outside of the library to prevent users from reusing it manually.

4.3.1 Modifications to the Metamodel

The first change that had to be applied in order to simplify the use of the *Association* concern was to extend the RAM metamodel. Concern reuses are represented in the metamodel with the class *COREModelReuse.* As we saw in the CORE metamodel in Figure 2–6, a *COREModelReuse* is contained by a *COREModel*, which means it is saved within the *Aspect* in our case. In order to also store the reuse of the *Association* concern within a specific association end, a link between *AssociationEnd* and *COREModelReuse* was introduced in form of a unidirectional association *AssociationEnd* `--> 0..1` *COREModelReuse* as shown in Figure 4–3[1] .

4.3.2 Modifications to the Weaver

Drawing a line between two classes to represent an association is a DSL for specifying the intent of applying the *Association* concern. Actually applying the concern is done using the

[1] Note that the minimum cardinality is set to `0` because the *Association* concern itself, in order to prevent recursive reuse, should not reuse the *Association* concern, even if it defines associations.

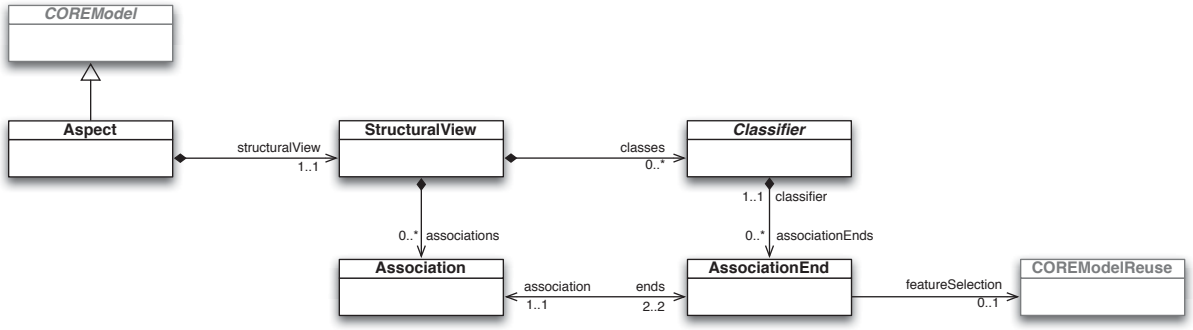


Figure 4–3: Changes to the RAM Metamodel

weaver, which combines the structure and behaviour of the current model with the structure and behaviour of the model generated from the realization models of the *Association* concern that correspond to the selected configuration. In the resulting model, we would expect to find that the line between `|Subject` and `|Observer` is gone, as it is replaced by the concrete realization of the association, e.g., one using an intermediate *ArrayList* collection.

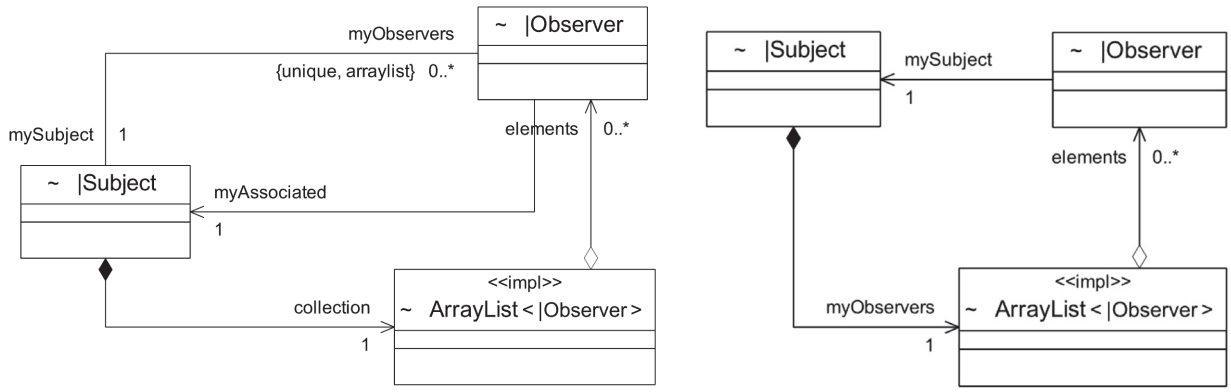
The weaver, however, does not allow the removal of model elements [25]: it can only add new model elements or merge model elements[2] . Therefore, when weaving a model that is reusing the *Association* concern, the weaver keeps the association from the model and adds another one from the realization models of the concern, leading to a woven model where the association is represented twice. This is illustrated in Figure 4–4a, where the `myObservers` end from the original model is now also realized by the two associations going from `|Subject` to `ArrayList` to `|Observer`. As a result, there are actually now two ways of navigating from

[2] A current limitation in TouchCORE also prevents merging of associations and association ends.

`|Subject` to `|Observer`. Likewise, the `mySubject` end is also duplicated, as it is now also realized by the `associated` end.

When composing two models, the weaver maintains a data structure called *weaving information*. The weaving information is a map where the key is a model element from one model, and the value is a model element (of the same type) from the other model. Essentially, it maintains a mapping from the original model element to the element it was woven into (in the resulting model). When weaving, the weaving information gets filled with classes, attributes, operations and associations according to the customization mappings provided by the user and the default rules defined for model extensions in CORE. Finally, the last step in weaving a reuse is called *post processing*. During that step, all occurrences of the *key* elements in the weaving information are merged into the *value* elements. All model elements from the input models that are not in the weaving information data structure are copied to the output model. The post processor also merges any duplicate classes it finds. To improve the weaver in order to deal properly with associations, we added an additional step called *processAssociations* at the end of this post processing phase, which is only executed if the current reuse being woven is of the *Association* concern. This step handles two different cases:

- Single references [e.g., `mySubject` end]: In the case of single references, we add an entry in the weaving information where the *myAssociated* end is the key and the end in the model is the value (`mySubject` in our example). As a result, the weaver then merges the association ends. This also ensures that all references to `associated` in the message views will be updated to `mySubject`.



(a) Before Modifying the Weaver (b) After Modifying the Weaver

Figure 4–4: Woven Model of the *Observer* reusing *Association*

- Collections [e.g., `myObservers` end]: In the case of an association end with multiplicity many, the original association end is refined, and its name replaces the *collection* name in the association realization model. For example, in the case where *Association* is reused as part of the *Observer* concern, the `myObservers` end is used to update the association coming from the concern. Its name and multiplicity are preserved as follows: the *collection* end is renamed to `myObservers` and the `elements` end is given multiplicity `0..1`.

4.3.3 Streamlining the Association Reuse

The main flow of execution in the tool follows the MVC [13] architecture style. The core components are *Views*, *Handlers* and *Controllers*. Handlers listen for events from the Views, which implement the visual representation of the model elements as part of the GUI. The handlers interpret the events, and once they determine what the modeller wants to accomplish, they call the Controllers to make command-based changes to the model.

In the case of associations, the `AssociationView` encapsulates the main visual representation of the association. It is composed of `TextViews` for the role name, e.g. `mySubject` or `myObservers`, and the multiplicity of both ends: `fromEndRoleName`, `toEndRoleName`, `fromEndMultiplicity` and `toEndMultiplicity`. Their handlers, `AssociationRoleNameHandler` and `AssociationMultiplicityHandler` call `setRoleName` and `setMultiplicity` of the `AssociationController`, respectively.

When an association is drawn from one class to another, the `createAssociation` in the `StructuralViewController` is called. Now, it triggers a `setFeatureSelection` in the `AssociationController` for each navigable end. The `setFeatureSelection` is responsible for creating a reuse for the *Association* concern, which includes the generation of mappings and the feature selection and storing the reuse in the association end.

Automated Generation of Mappings

The modeller must customize the generic model elements of the concern by mapping them to the model elements in the reusing model. As explained in Section 4.1, this can be quite tedious for the *Association* concern, as typically many of the generic operations should be renamed to reflect their more specific purpose which involves mappings of operations. For example, `get` should be renamed to `getMySubject`. This subsection describes how this customization step was fully automated.

An operation `createReuseInstantiation` was added to the `AssociationController`. It creates the `ClassifierMappings` (for `|Data`, `|Associated` and optionally for `|Key`) and the `OperationMappings` (for operations like `add`, etc.). Since operations need to be mapped to existing operations in the classifier, and since they typically do not exist when reusing the *Association* concern, they are first created. The operation `createOperationMappings` in

the controller deals with this. It copies the operations of the classes inside the concern and clones them. It renames them by appending the name of the association end before adding them to the mapped class in the reusing concern.

In Figure 4–5, a unidirectional association is drawn, therefore the concern is reused once. `|Data` is mapped to `|Subject` and `|Associated` is mapped to `|Observer`. `|Data` has a getter and a setter. We then created a getter and a setter for `|Subject` and renamed them with the role name of the association end *myObserver* to get `getMyObserver` and `setMyObserver`. Theses operations are also mapped. In Figure 4–6, a bidirectional association is drawn, hence, the concern is used twice. In the first reuse, `|Data` is mapped to `|Subject` and `|Associated` is mapped to `|Observer` and vice versa in the second reuse.

Now that the operations are named after the role name of the association end, it makes sense to ensure that this remains the case even when updates are made to those role names. To this aim, the operation `setRoleName`, invoked whenever the user updates the role name of an association end, was extended to iterate through the operations from the operation mappings in the corresponding reuse and rename them accordingly.

Automated Feature Selection

As explained in Section 4.2 of this chapter, our DSL provides a visual notation that the modeller can use to specify the desired association features: *Minimum*, *Maximum* and *Bidirectional*. In order to ensure the consistency between the specification provided by the modeller using the DSL and the feature selections stored in the *Association* concern reuse, the feature selection of the *Association* concern is now performed automatically, derived from the properties of the association that was drawn. It is performed without the modeller being aware of it. As soon as an association is created or modified, the

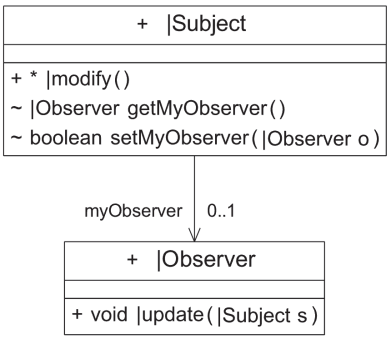


Figure 4–5: *Observer* Model with a Unidirectional Association

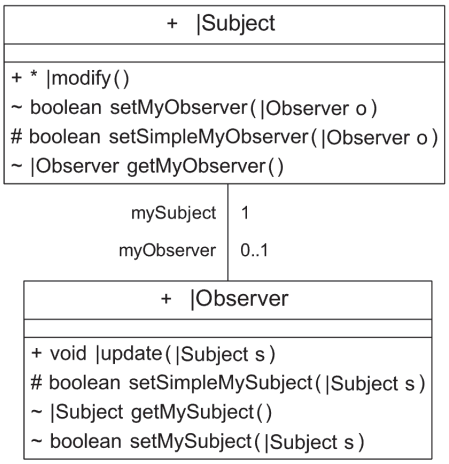


Figure 4–6: *Observer* Model with a Bidirectional Association

controller analyses the properties of the association. It checks whether it is navigable in both directions and what the multiplicity is on the navigable ends. To this aim, an operation `getSelectedFeatures` was added to the `AssociationController`, which takes the `associationEnd` and the `oppositeEnd` as parameters and creates a set of selected features as follows:

One when the upper bound of the multiplicity is `1`.

Many when the upper bound of the multiplicity is greater than `1`.

Maximum when the upper bound of the multiplicity is greater than `1` and not many (`*`).

Minimum when the lower bound of the multiplicity is `1` or greater and the upper bound is greater than `1`.

OneOpposite when the association is navigable in both directions and the upper bound on the multiplicity of the opposite end is `1`.

ManyOpposite when the association is navigable in both directions and the upper bound on the multiplicity of the opposite end is greater than `1`.

The `getSelectedFeatures` operation is not only used when creating a new association in a model, but also when updating the multiplicity or the navigability. The operations `setMultiplicity` and `switchNavigable` of the controller were extended to update the selection whenever the multiplicity or navigability of an association is modified. In Figure 4–5, the selected feature is *One* and in Figure 4–6, the selected features are *One* and *OneOpposite* for both reuses as the multiplicity is `1` on both sides.

Modifications to the Association View

In addition to the features selected automatically, the concern offers implementation classes to the modeller that he can choose from. Changes to the UI were required to allow the user to make this selection manually. Those changes reflect UML's notation for ordering and uniqueness specifications. `TextViews` were added to the `AssociationView` to handle displaying the feature selection: `fromEndFeatureSelection` and `toEndFeatureSelection`. These `TextViews` are, however, only created when the multiplicity of the association end is greater than `1`. When the multiplicity is `1`, the user should never be presented with the variation interface as he does not need to select a data structure.

When the `TextView` is created, i.e., as soon as a multiplicity greater than `1` is entered, a clickable text `{select}` is displayed as shown in Figure 4–7a. Double clicking on this text component presents the variation interface of the *Association* concern to the modeller. However, the variation interface presented is just a subset of the one shown in the previous chapter. The features that can be inferred from the model are omitted from the feature model. Specifically, the *Bidirectional* subtree, the *One*, *Minimum* and *Maximum* features

are not shown, as allowing the modeller to select and deselect those features manually could result in an inconsistency between the information specified using the DSL and the actually selected features. For example, the modeller could select *Minimum* whereas the lower bound of the multiplicity is `0`. Once a selection is made, it is displayed next to the multiplicity in curly brackets, such as `{arraylist}`, as is shown in Figure 4–7b.

Another change made to the view is the visualization of qualified associations. After a key-indexed selection is made, a box is rendered next to the class that is at the origin of the association end, which is shown in Figure 4–7c when reusing *HashMap*. The box allows the modeller to select the type of the key. Once the selection of the key is made, the `AssociationController` creates an additional customization mapping and updates the operation signatures.

4.3.4 Further UI Improvements

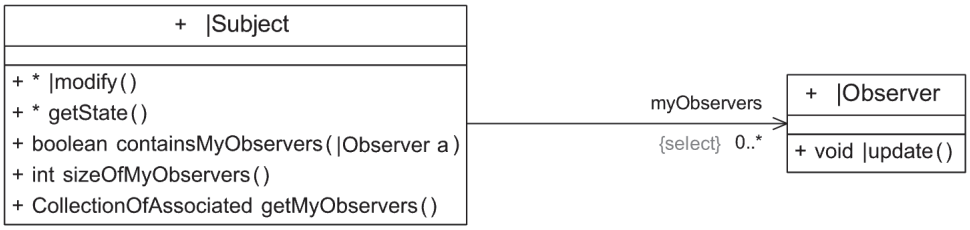
Once we started using our DSL, we realized we could further improve the user experience by reducing the visualization clutter and by restricting direct access to the concern.

Preventing Visualization Clutter

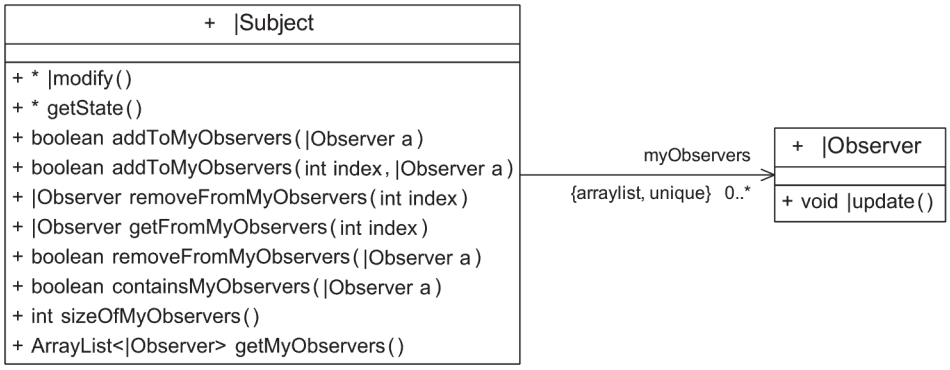
The list of operations added to a class when reusing the *Association* concern can be quite extensive, and not all the operations are always relevant to the user. The list also takes up space on screen that could be used for other purposes.

For practicability reasons, the operations are therefore added automatically when creating associations (see subsection 4.3.3) are hidden by default, but can be shown if the modeller so desires.

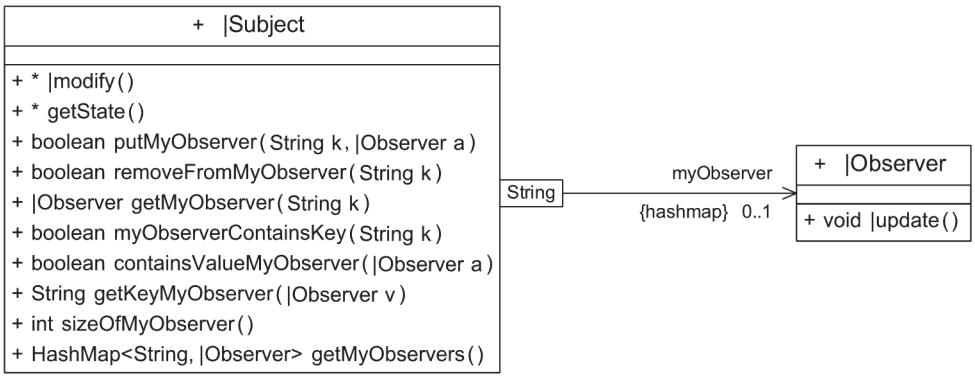
To implement this functionality, the `ClassView` has now a boolean attribute `showAllOperations` that is initialized to `false`. The `ClassView` holds a map with the `Operation` as key and



(a) Reusing *Many*



(b) Reusing *Unique* and *ArrayList*



(c) Reusing *HashMap*

Figure 4–7: Class Diagram of the *Observer* Structural View reusing the *Association* Concern

the `OperationView` as value. On adding an operation to the class, the view creates an `OperationView` and maps it to that `Operation`, it is then added to the `operationsContainer` of the `ClassView` if the `showAllOperations` attribute is `true`. Expanding and collapsing operations consists of adding and removing the `OperationView` from the `operationsContainer`.

In order to expand and collapse operations, a button was added to the class menu. When an operation is made public or used in a message view, it is always shown. Currently, it only hides operations mapped to operations from the *Association* concern, as we did not find other use cases where such a hiding feature could be useful, but the implementation was performed in such a way that it could easily be reused for other concerns, if needed.

Preventing Manual Updating of Mappings

Concern reuses within an aspect are listed in TouchCORE within a "Concern Reuses" box as shown in Figure 4–1. Since the *Association* concern is reused frequently in most models, this visualization is quickly overloaded and it is hard to understand the concern dependencies visually. Also, users should not be able to modify the mappings through this interface. We decided to hide reuses of the *Association* concern from this list. This strengthens the fact that the reuse of the *Association* concern happens without the user knowing it is happening.

Preventing Manual Reuse of the *Association* Concern

Now that we created a DSL for reusing the concern in a streamlined and safe manner, the concern should clearly be used only through this DSL to prevent incorrect use. The concern was originally part of the reusable concern library, which could lead modellers to reuse the concern manually. Therefore, the concern was moved out of the reusable concern library into a resources directory.

Chapter 5
Benchmarks and Goal Models

Deciding on a concrete data structure for implementing an association is not always trivial. In order to provide guidance to the modeller who is reusing the *Association* concern, we conducted a series of experiments to determine the impact that the different realizations with Java collections have on performance and memore use.

Section 5.1 of this chapter presents the experimental setup and methodology for developing our impact models. Then, we describe in different sections how we built goal models for *Increase Insertion Performance*, *Increase Iteration Performance*, *Increase Access Performance*, *Increase Removal Performance* and *Minimize Memory Usage*. Section 5.7 compares the benchmarks performed on other platforms. The last section discusses the benefits and drawbacks of impact models.

5.1 Experimental Setup and Methodology

5.1.1 Experimental Setup

We ran our experiments on a machine with a 2,4 GHz Intel Core i5 processor and 16GB 1600 MHz DDR 3 memory. The machine was running Mac OS X 10.9.5. The Java SE Runtime (v. 1.8.0_20-b26) was configured with 384MB heap space. We created a jar file that contained the code for the experiment and ran it from the command line.

5.1.2 Methodology

Georges [17] compares Java performance evaluation methodologies in a paper where he affirms:

> "*Java performance is far from being trivial to benchmark because it is affected by various factors such as the Java application, its input, the virtual machine, the garbage collector, the heap size, etc. In addition, non-determinism at run-time causes the execution time of a Java program to differ from run to run.*"

We therefore designed our experiments to mitigate some of the problems that Georges refers to. Nevertheless, we are aware that the values obtained by our experiments do not encode an absolute truth.

The model used for the experiment was the simplest possible model, i.e., a model with a directed association `myB` with multiplicity `0..*` between classes `A` and `B`.

To measure the impact on performance, we completed benchmarks that were inspired by a performance evaluation performed by Ahuja [4]. We created n instances of `B` ($n = 10$ *(small),* $n = 100$ *(medium),* $n = 1{,}000$ (large), $n = 10{,}000$ (extra-large)), and added them to the collection `myB`. We performed the following steps for each performance benchmark:

1. We ran experiments with associations of different orders of magnitude (#elements = n) and recorded the time t it took to execute each operation `op` n times within a loop. For example, we calculated the time it took to add *10* elements in a list and not the time it took for each addition. Some small overhead might be due to the for loop, but the overhead is reasonably small.
2. We ran each sequence 60 times and discarded the first 10 runs to avoid including time used for code loading/initialization in our measurements.
3. From the set of 50 values, we calculated the *average* ($\bar{t}$) and the *median* ($\tilde{t}$).
4. We built two charts: a bar chart and a line chart to visualize the data . For both charts, the x-axis represents the size of the collection and the y-axis, using a logarithmic scale,

represents the median execution time obtained from the 50 measurements. In the bar chart, the vertical black line on each bar represents the 10th and 90th percentile range.

5. From the median, we then determined the relative values to be used in the impacts models. The lower the median value, the better the performance. Hence, since in goal models higher contributions are better than lower contributions, we use the negated median values as contributions to the *Increase Performance* goal.

For each performance benchmark, we provide the Java code that iterates and performs the `op` n times, as well as a table with $\bar{t}$, $\tilde{t}$ and a chart representative of the table for each collectionWe show only one impact model for increasing the performance (*Increasing Insertion Performance*) that groups 2 sub-goals, one for small data and one for big data.

5.2 Insertion

The insertion experiment was done by passing an array of `B`s and that were all added to the collection through a `for` loop as shown in Algorithm 7 by calling the operation `add`[1] . The results for the experiment that focusses on insertion performance are shown in Table 5–1.

When looking at the trend line of the median performance depicted in Figure 5–2, we observe that `ArrayList` and `LinkedList` consistently perform well, whether the size of the collection is big or small. For small collections, `LinkedList` slightly outperforms `ArrayList`, whereas `ArrayList` is slightly better than `LinkedList` for big collections. `TreeSet` and

[1] For ordered collections, `add` effectively appends the element passed as a parameter to the end of the collection. For lists, the insertion at a certain index through `add(index, object)` could also be tested and the results captured in a separate goal model. However, we did not run benchmarks for this case as the operation is not available for all collections.

`TreeMap` clearly perform worst when the size is small. As the size increases, `TreeSet` is still the least performant, but `TreeMap` slowly outperforms `HashSet`.

The impact model for *Increase Insertion Performance* is presented in Figure 5–3. There are 2 sub-goals: one for small sizes that combines sizes of *10* and *100* and one for big sizes that combines sizes of *1,000* and *10,000* as the trends for each group are similar. Each sub-goal contributes evenly to the parent goal. To calculate the contribution values used in the impact model, we used an approximation of the relative values of the sum of the medians within each subgroup. For example, in the collection of sizes 10 and 100 for `ArrayList` the median is 2.55µs and 15.4µs, which sums to 17.95µs. For `LinkedList`, 2.2µs and 10.4µs sum up to 12.6µs. $\frac{17.95}{12.6} \approx 3:2$. Therefore, the weight on `ArrayList` is -3 and the weight on `LinkedList` is -2 for collections with a smaller size.

Algorithm 7 Algorithm for Timing Inserting n Elements

```
public static long addToCollection(
    ArrayList<B> collection, int n, B[] arrayOfBs) {
  long before = System.nanoTime();
  for (int i = 0; i < n; i++) {
    collection.add(arrayofBs[i]);
  }
  long after = System.nanoTime();
  return after - before;
}
```

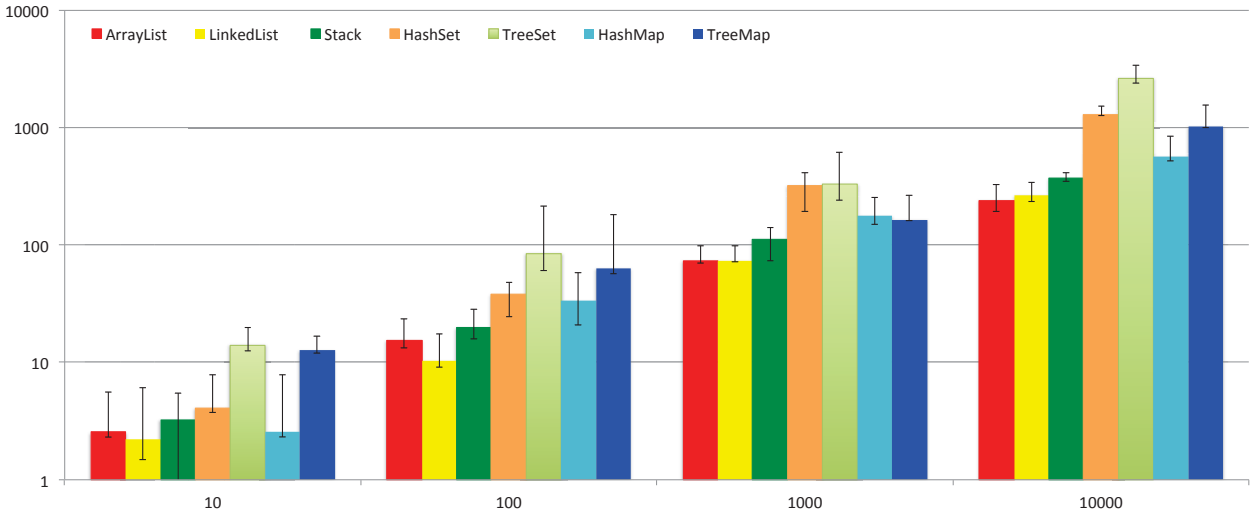


Figure 5–1: Insertion Performance showing Median in µs, as well as 10th and 90th Percentile Range

n	10		100		1,000		10,000	
	$\bar{t}$	$\tilde{t}$	$\bar{t}$	$\tilde{t}$	$\bar{t}$	$\tilde{t}$	$\bar{t}$	$\tilde{t}$
ArrayList	3.73	2.55	17.1	15.4	73	73	238	239
LinkedList	42.27	2.2	60.5	10.4	122	73	273	265
Stack	44.55	3.26	71.5	19.8	140	111	377	367
HashSet	39.41	4.09	71.5	37.8	293	319	1355	1289
TreeSet	15.78	13.95	151.2	84.2	362	330	2806	2619
HashMap	4.5	2.54	38.2	33.6	192	176	632	560
TreeMap	13.71	12.81	130.7	63.5	233	164	1147	1029

Table 5–1: Insertion Performance Results in µs

5.3 Iteration

The iteration was done through a `foreach` loop that used the underlying iterator as shown in Algorithm 8. For this case, the loop does not represent an overhead as it is part of

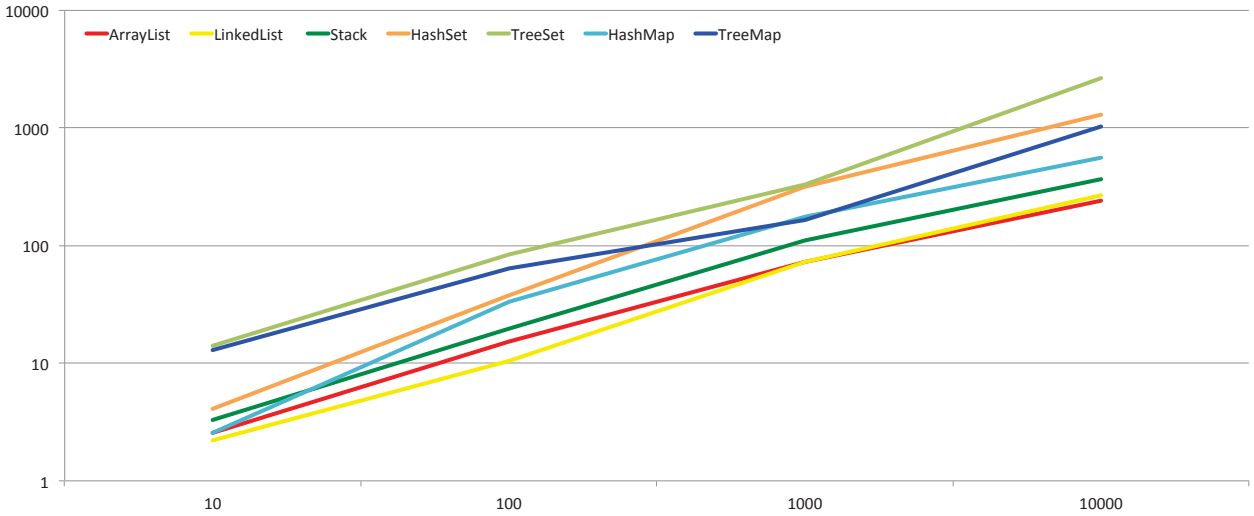


Figure 5–2: Median Insertion Performance in µs with Trend Line

the iteration process we want to measure. A summary of the measured performance results is presented in Table 5–2. The median performance together with the 10th and 90th percentile are shown in Figure 5–4, and median trend is shown in Figure 5–5.

Overall, iterating over an `ArrayList` or a `LinkedList` performs consistently well, and performance increases considerably when the collection grows to *10'000* elements. `TreeMap` and `HashMap` hav initially very bad performance, but perform better with big collections. Iterating over a `Stack` and `TreeSet` performs similarly, ranging from ok for small collections to bad for big collections. Based on the trends, we clustered the impact model for *Increase Iteration Performance* into 3 subgoals, one for small collections (≈10), one for mid-size collections (≈ 100), and one for big collections (1000+).

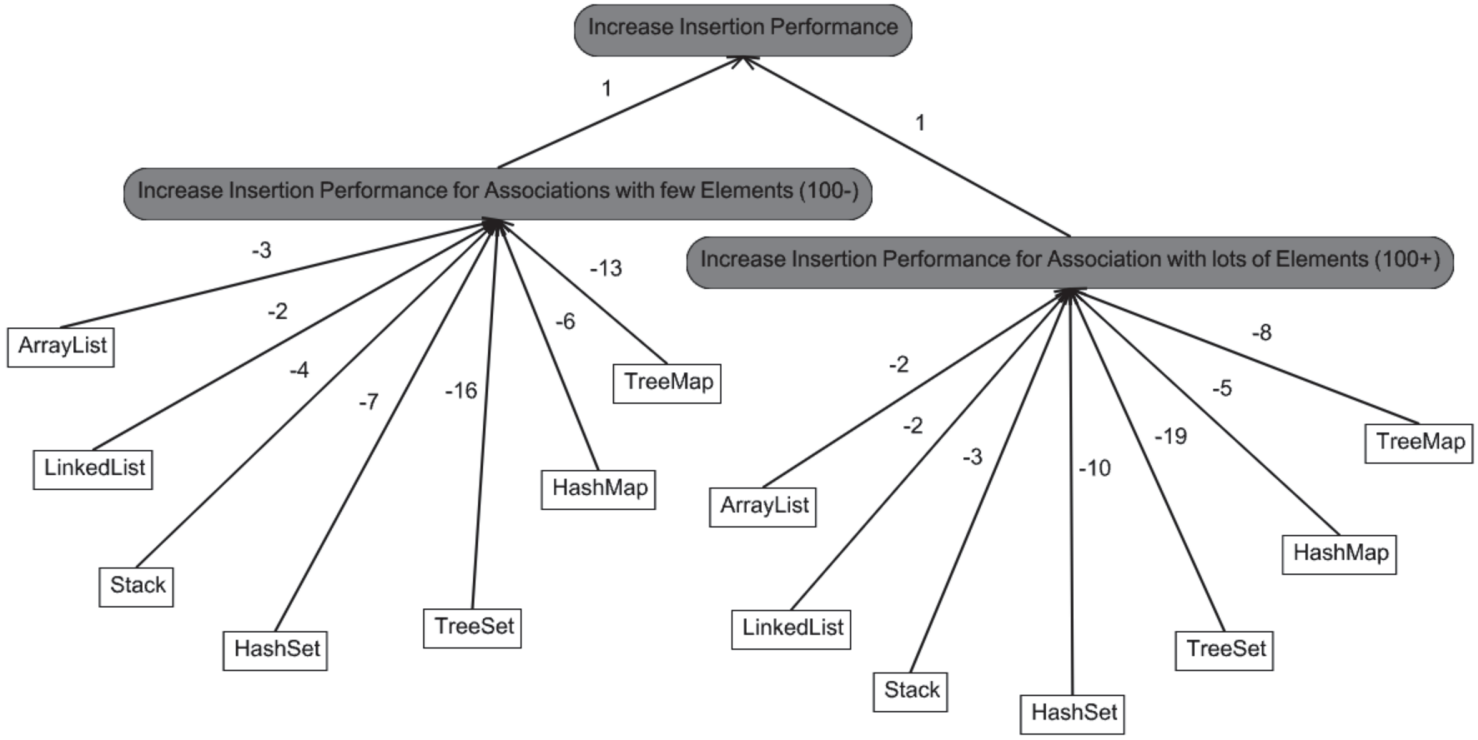


Figure 5–3: *Increase Insertion Performance* Impact Model

Algorithm 8 Algorithm for Timing Iterating over n Elements

```
public static long iterateCollection(ArrayList<B> collection) {
  long before = System.nanoTime();
  for (B b : collection) {
    helper = b;
  }
  long after = System.nanoTime();
  return after - before;
}
```

n	10		100		1'000		10'000	
	$\bar{t}$	$\tilde{t}$	$\bar{t}$	$\tilde{t}$	$\bar{t}$	$\tilde{t}$	$\bar{t}$	$\tilde{t}$
ArrayList	3.99	3.6	49.2	11.2	94	57	41	25
LinkedList	5.3	4.2	10.6	8.8	62	57	50	42
Stack	5.31	4.33	13.8	12.7	84	77	267	263
HashSet	3.74	2.92	12.4	11	86	76	195	166
TreeSet	4.29	3.52	22	17.3	93	98	268	229
HashMap	6.07	4.95	29.6	25.8	118	102	171	93
TreeMap	6.23	5.92	36.8	47	81	70	180	118

Table 5–2: Iteration Performance Results in µs

5.4 Random Access

Access performance is possible for lists (with an index) and for maps (with a key). It is however not a feature of sets.

The algorithm for random access performance is shown in Algorithm 9. Now, the overhead also includes getting a random number. To ensure the compiler will not get rid of the access if the value accessed is never used, we pass an array and fill it with the random values which we then use to remove randomly. `Random` gives a uniform distribution of random numbers, therefore we know that for lists, the objects were accessed uniformly. We

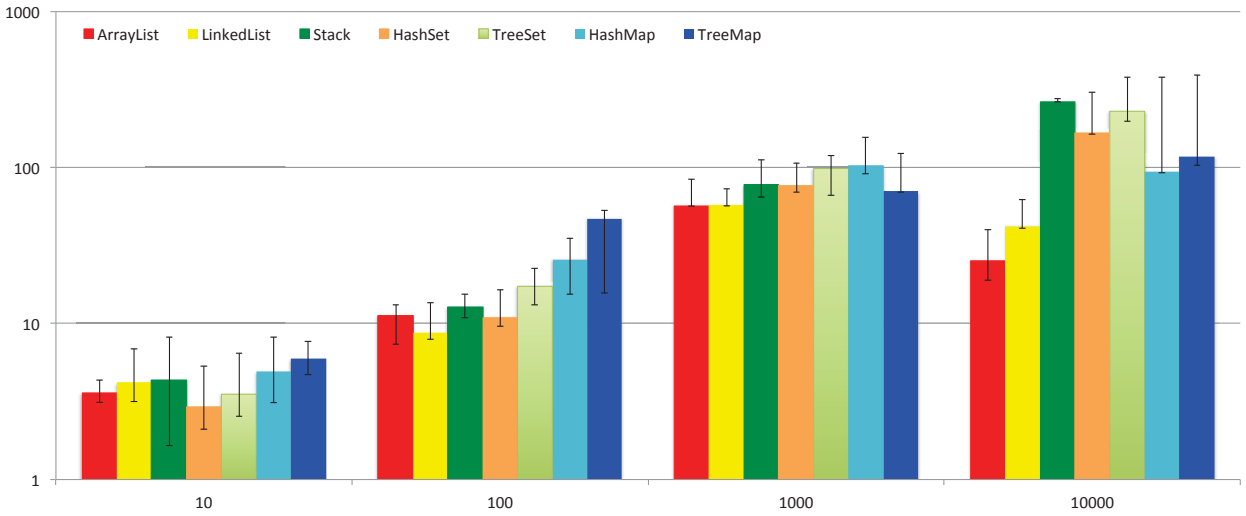


Figure 5–4: Iteration Performance showing Median in µs, as well as 10th and 90th Percentile Range

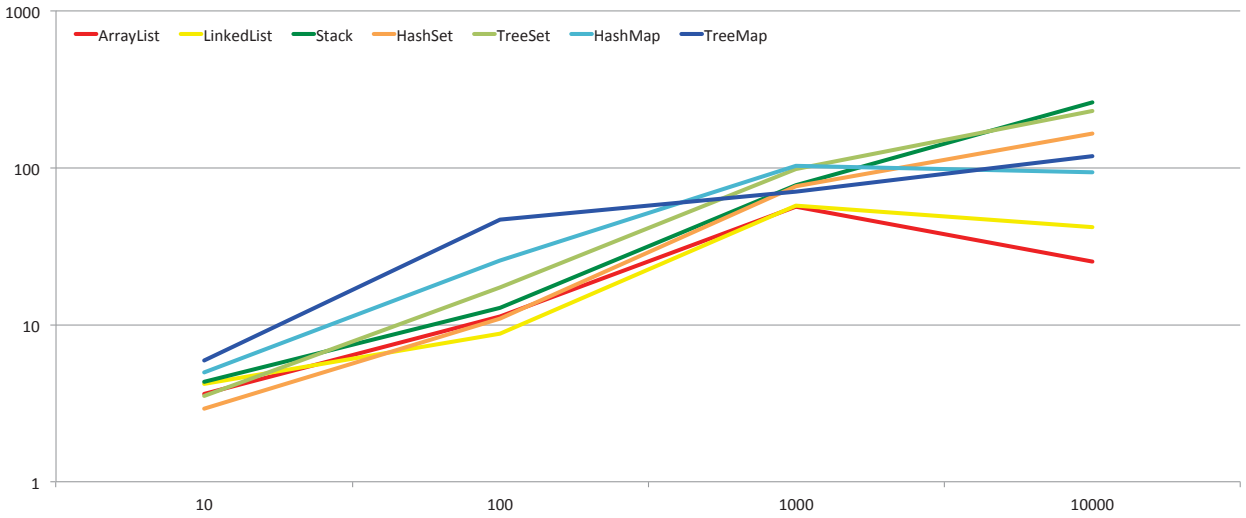


Figure 5–5: Median Iteration Performance in µs with Trend Line

observe through Figure 5–6 and Figure 5–7 that `ArrayList` and `Stack` are the best option throughout all sizes, closely followed by `HashMap`. `LinkedList` is very costly for big collections, and `TreeMap` is also not great.Based on the trends, we clustered the impact model for *Increase Access Performance* into 3 subgoals, one for small collections (≈10), one for mid-size collections (≈ 100), and one for big collections (1000+).

Algorithm 9 Algorithm for Timing Randomly Accessing n Elements

```
public static long accessElementsInCollection
    (ArrayList<X> collection, int seed, int n, B[] randomBs) {
  Random random = new Random(seed);
  long before = System.nanoTime();
  for (int i = 0; i < n; i++) {
    randomBs[i] = collection.get(random.nextInt(n));
  }
  long after = System.nanoTime();
  return after - before;
}
```

n	10		100		1'000		10'000	
	$\bar{t}$	$\tilde{t}$	$\bar{t}$	$\tilde{t}$	$\bar{t}$	$\tilde{t}$	$\bar{t}$	$\tilde{t}$
ArrayList	7.5	5,46	17.8	15,8	70	66	230	207
LinkedList	12.52	10,54	22.2	19,4	523	492	48389	47715
Stack	7.08	7,2	16	14,9	96	77	319	305
HashMap	6,06	5,76	21.2	21,7	124	115	472	352
TreeMap	7,21	6,14	38.4	34,1	213	198	2028	1787

Table 5–3: Access Performance Results in µs

5.5 Removal

In order to remove randomly, we pass the array of random `B`s that was generated in the previous section when randomly accessing. In Algorithm 10, we can see that there is an

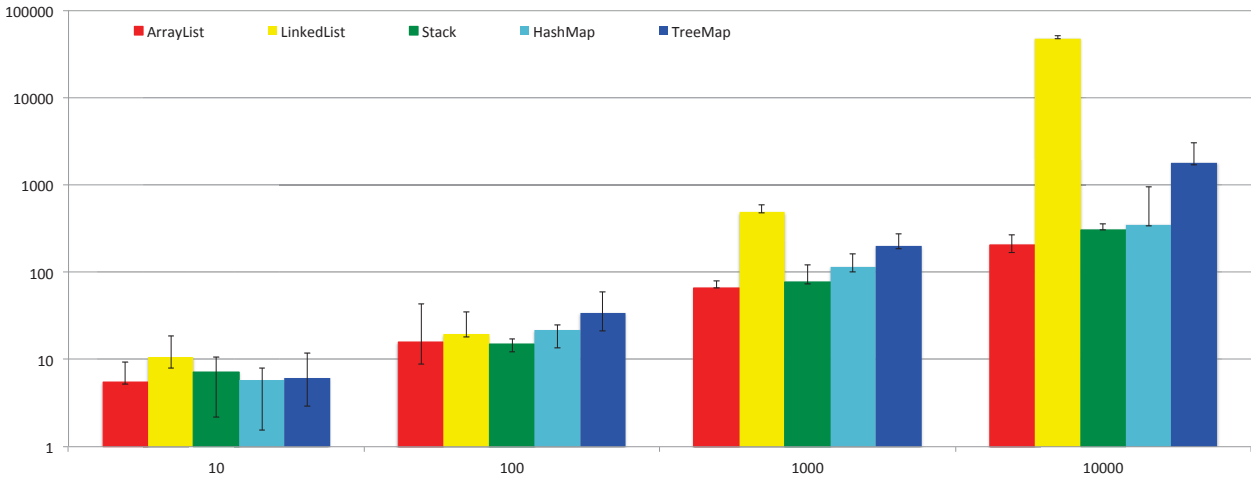


Figure 5–6: Median Random Access Performance sin µs, as well as 10th and 90th Percentile Range

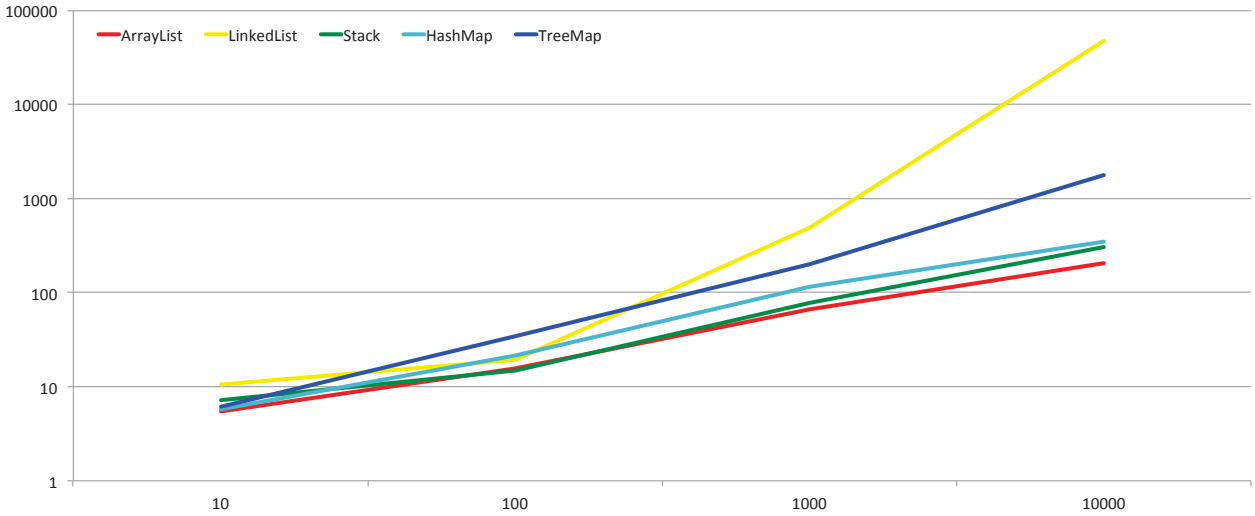


Figure 5–7: Median Random Access Performance in µs with Trend Line

extra overhead due to accessing a B in the array, but it is a straightforward way to empty the collection randomly. The performance results are shown in Table 5–4.

We observe through Figure 5–8 and Figure 5–9 that removing for `TreeMap` and `TreeSet` is less performant for smaller collections, but removing from ordered collections (`ArrayList`, `LinkedList`, `Stack`) is very costly for bigger collections. `HashMap` and `HashSet` perform mediumly well for small collections, and best for big collections. Based on the trends, we clustered the impact model for *Increase Access Performance* into 3 subgoals, one for small collections (≈10), one for mid-size collections (≈ 100), and one for big collections (1000+).

Similar to the adding, we could have also determined the performance of removing with an index, but we decided to not run benchmarks for this situation because it is only applicable to ordered collections.

Algorithm 10 Algorithm for Timing the Removal of n Elements from the Collection

```
public static long removeFromList(
    ArrayList<B> collection, int n, B[] randomBs) {
  long before = System.nanoTime();
  for (int i = 0; i < n; i++) {
    collection.remove(randomBs[i]);
  }
  long after = System.nanoTime();
  return after - before;
}
```

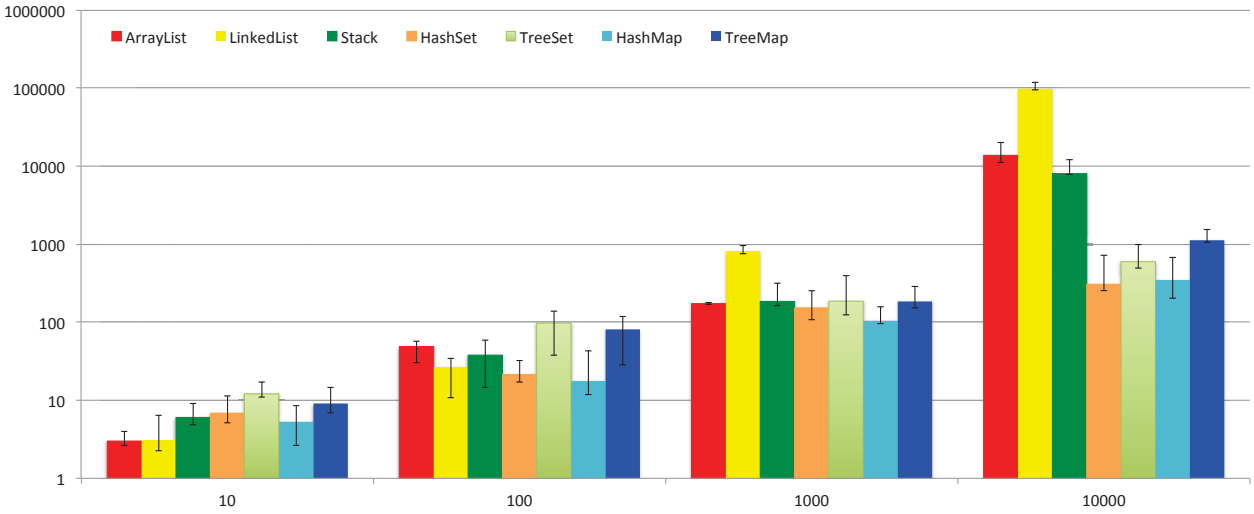


Figure 5–8: Median Removal Performance in µs, as well as 10th and 90th Percentile Range

n	10		100		1'000		10'000	
	$\bar{t}$	$\tilde{t}$	$\bar{t}$	$\tilde{t}$	$\bar{t}$	$\tilde{t}$	$\bar{t}$	$\tilde{t}$
ArrayList	3.24	3.04	44.8	48.1	181	174	14954	13568
LinkedList	3.95	3.06	23.2	26.9	838	812	101582	97444
Stack	6.37	6.1	36.7	38.2	225	183	9065	7951
HashSet	7.78	6.8	25.2	21.2	215	153	417	302
TreeSet	13.1	12.05	136.6	97.5	290	186	713	597
HashMap	41.25	5.28	68.9	17.6	149	106	357	351
TreeMap	10.62	9.1	70	80	199	186	1335	1111

Table 5–4: Removal Performance Results in µs

5.6 Memory Footprint

We also wanted to compare memory usage of the presented collections. To determine the size of the collection in the memory of the virtual machine we used the *Heap Walker* of *JProfiler* [32]. The retained size of the collection includes the size of the elements contained

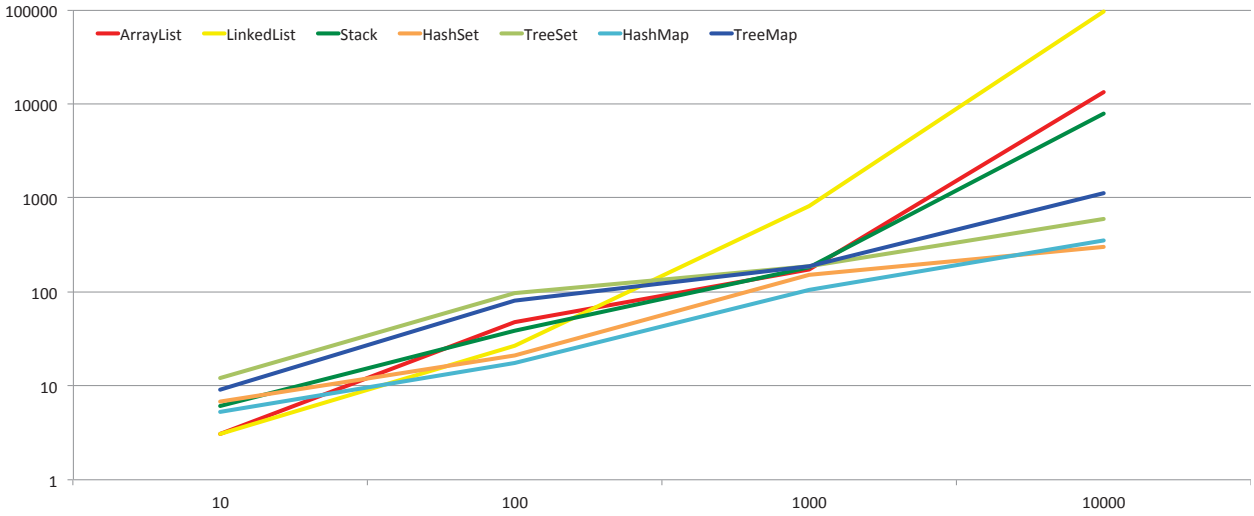


Figure 5–9: Median Removal Performance in µs with Trend Line

in the collection, which in our case is 16 bytes per element. We therefore substracted 16 bytes * n to obtain the actual amount of memeory that the collection data structure uses.

Table 5–5 shows the memory usage of the collections for six different sizes. We observe the values are proportional to n. Thus, we captured the results of the experiment in one single *Minimizing Memory Footprint* goal model, using the values in the column where n is *1'000*, which is shown in Figure 5–10.

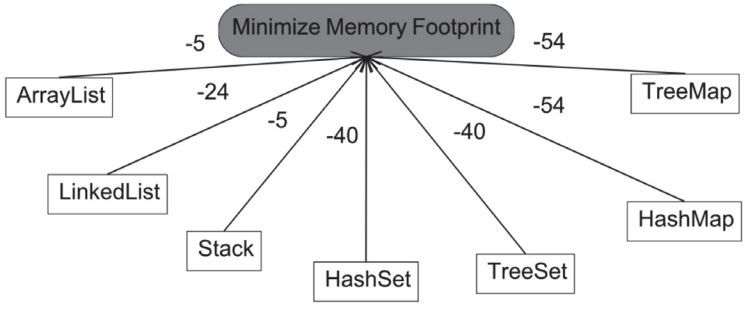


Figure 5–10: *Minimize Memory Footprint* Impact Model

n	10	100	1,000	10,000	100,000	1,000,000
ArrayList	80	480	5k	56k	426k	4861k
LinkedList	272	2432	24k	240k	2400k	24000k
Stack	88	688	5k	41k	655k	5242k
HashSet	464	4304	40k	385k	4248k	40388k
TreeSet	464	4064	40k	400k	4000k	40000k
HashMap	448	4288	54k	543	5846k	56386k
TreeMap	448	4048	54k	558	5598k	55998k

Table 5–5: Memory Usage in bytes

5.7 Determining the Performance Impact of the Underlying Platform

In order to determine how dependent our results are on the underlying platform, we ran the `ArrayList` experiments for Insertion, Iteration, Access and Removal also on a different machine. The machine was equipped with a 2,5 GHz 6-core Intel Xeon processor and 8GB 2500 MHz DDR 3 memory. The machine was running 3.16.0-29-generic GNU/Linux Ubuntu. The Java SE Runtime (v. 1.8.0_60-b27) was also configured with 384MB heap space. We used the same jar file, executed from the command line.

In the 4 graphs of Figure 5–11, we observe that the median performance for all 4 experiments varies slightly between MacOS and Linux. Also, the 10th and 90th percentile range differs between the two platforms, MacOS being more consistent for Insertion and Access, while Linux tends to be more consistent for Iteration and Removal. However, the median trends are very similar between the two platforms.

From these 4 experiments on Linux we conclude that for now our impact models do not need to take into consideration the platform, since the weights in our impact models are anyhow just approximations of the trends. Furthermore, the impact models currently used

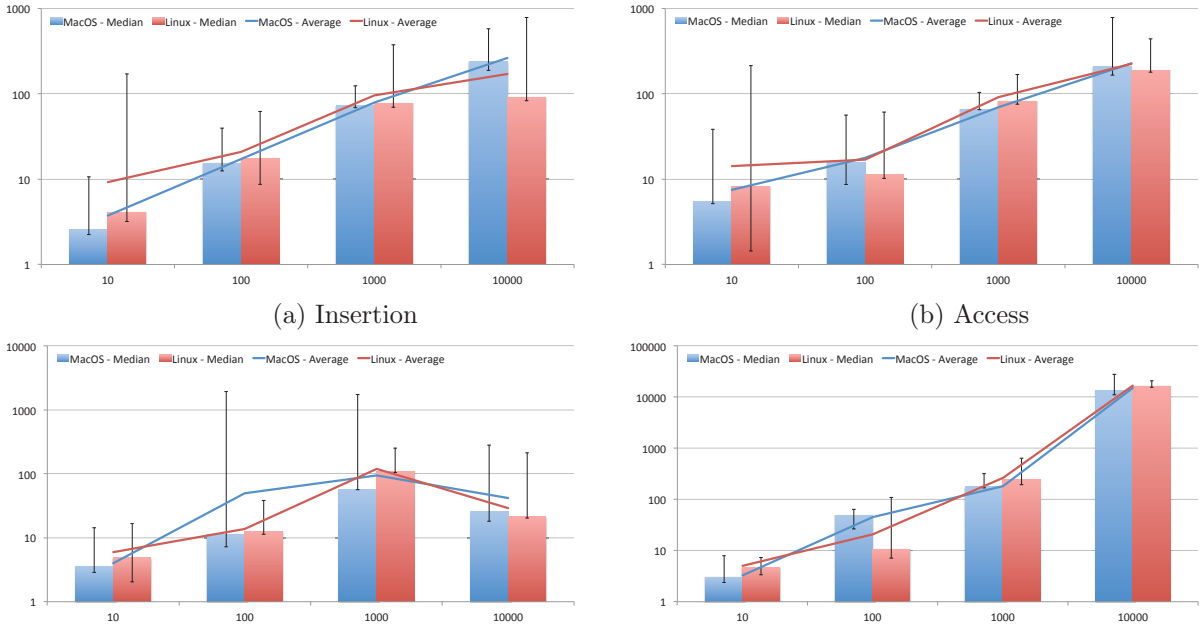


Figure 5–11: Comparing Performance Results for `ArrayList` on Mac OS and Linux

in TouchCORE cannot be parameterized, which would be necessary to adjust the impacts depending on a user-provided platform parameter. In the future, if the impact models of TouchCORE are extended to support parameterization, one could perform more extensive experiments to determine the platform dependencies, but this is out of the scope of this thesis.

5.8 Discussion

Impact models in CORE are currently exclusively specified using the goal modelling notation [23]. Goal models work well in the context of CORE, because they allow vague, hard-to-measure system qualities to be evaluated, e.g., *user convenience* or *security*. In addition, more quantifiable qualities can be specified, e.g., *cost* and *number of messages*

exchanged. Unfortunately, impact models can not be parameterized with dynamic information from the reuse context. As a result, our impact models can not be used for predicting the actual memory use or the actual performance of the final application. Rather, they are intended to help the modeller make design decisions by quantifying the impacts that one selection has over another *relatively* speaking. There exist dedicated performance modelling languages that offer advanced performance simulation and prediction capabilities [26], but a discussion on how to combine these with CORE is out of the scope of this thesis.

Chapter 6
Related Work

To our knowledge, the concern-orientation reuse paradigm is currently the only modelling approach that supports the encapsulation of different structural and behavioural designs and implementations within one reusable model. As far as we know, this is the only work that tried to encapsulate the variations of associations. Many research projects are oriented towards improving code generators when it comes to associations. It is one of the concepts that is particularly lacking in some tools, they fail to capture the variations that affect the semantics of an association [5].

In TouchCORE, no work had to be done on the code generator to support associations, because all the behaviour is modelled. Therefore, if in the future the design of the *Association* concern needs to be updated in order to support new features of Association (e.g., to make them *thread safe*), no understanding or modifications of the code generator are required. The concern designer just needs to open the concern in TouchCORE and update the features, class diagrams and message views.

We explored how they discuss and handle the difficulties in translating multiplicities and navigability, and in some cases visibility, to code. However, this thesis does not deal with visibility as the tool currently only supports private association ends.

Harrison (2000) describes a new method for generating Java implementation code from UML diagrams [21]. He suggests generating an interface for dealing with the behaviour of associations (creating, deletion) in a manner transparent to the user. He proposes the

creation of an interface and its implementation for each association end. The interface extends both the destination class and the association class, if one was modelled. It ensures referential integrity and multiplicity constraints, but does not provide any implementation class.

Génova (2003) presents some principles to mapping UML associations into Java code [16]. They demonstrate that it is unreasonable to keep the minimum multiplicity constraint at any moment on a mandatory association end as it reduces usability. They give an example of a mandatory bidirectional association: A `1-1` B where $\texttt{a}_1$ is association with $\texttt{b}_1$ and $\texttt{a}_2$ is associated with $\texttt{b}_2$. When wanting to change the state to have an association between $\texttt{a}_1$ and $\texttt{b}_2$, it is impossible to do so using primitive operations without avoiding a wrong state. Therefore, they also let the user be responsible for initializing the system to a consistent state, and maintaining it. Akehurst (2006) introduces some code generation patterns for bidirectional and multiplicity constraints that favours the production of Java based implementations from UML models [5]. Akehurst describes the situation of having a qualifier at both ends while UML does not give much information apart from saying that this situation rarely occurs. As Harrison's, Gessenharter's paper presented at MODELS 2008 [18] proposes that each association be implemented as a classes as he states that a non-public association end cannot be implemented as a reference in the class holding the end. For an association between A and B, he creates a class AB and class A hold a list of AB links Both class A and B have an `addA` and `addB` respectively that call a static method in AB to create a link.

Overall, papers discussing code generators for associations deal fairly well with UML constraints, i.e, maximum, minimum and bidirectional. However, they do not support qualified association as we do in the concern through the feature KeyIndexed. Also, they do not discuss concrete data structures to optimize a system.

We then describe in more details two other code generators that we quite pertinent for our work. The tool Mousetrap used in industry at Motorola is described in Section 6.1 and UMPLE, a tool developed at the University of Ottawa, Canada, is described in Section 6.2. The DSL used in TouchCORE to visually represent associations is an extension of the notation defined by UML, which was already mentioned in Section 4.2.1.

6.1 Mousetrap at Motorola

Motorola has developed its own automatic code generation tool suite called Mousetrap. The Mousetrap tool suite takes as input SDL, UML, ASN.1, and ISL (a proprietary protocol language) and produces highly optimizing code customized for a product platform and a set of performance constraints. Mousetrap is a rule-based code transformation system driven by a vast programming knowledge base.

Weigert and members of the Mousetrap project present the constraints in industrial-strengths systems that drive their model-to-code transformation steps [35]. They describe the transformations rules and the process itself that involves translating the models to generic constructs they call "Core", which is followed by optimization and then generation of application code in C to be used for their network products. Their design models do not specify any platform detail until they provide the platform specific interface to the code generator. Therefore, the design models are easily reusable.

Section 5.4 of [35] on Abstract Data Types (ADT) is the most related to our work. In their approach, code generation for associations involves the selection of a concrete implementation of an abstract data type. Where most code generators simply pick a default one all the time, the authors propose to analyze the behaviour of the model and determine the specific ADT selection that leads to a better tradeoff between memory usage and performance. For example, if the collection is often being iterated over, the system would favour a linked list as we saw through benchmarks that a linked list is more performant when it comes to iteration.

The solution proposed in this thesis currently does not deal with automatic selection of implementation classes. However, it allows the user to pick a concrete implementation from a given set and provides the necessary information to enable tradeoff analysis based on impacts. Based on this information, the user can decide to opt for faster execution time and increased memory usage depending on his preference. An automated reasoning system is possible in TouchCORE to optimize non-functional requirements by analyzing behaviours and impacts. This, however, is out of scope of this thesis and could be handled in future work.

6.2 UMPLE

UMPLE is a modelling tool developed by the team of Prof. Timothy Lethbridge of the University of Ottawa, Canada. It provides a textual syntax for low-level design modelling with class diagrams and state diagrams, as well as code generation. It derives its visual notation from UML, as does TouchCORE. UMPLE supports code generation towards many programming languages, i.e., Java, C++, PHP and Ruby.

What is interesting about UMPLE is that the code generator handles constraints on associations that are not taken care of in other UML modelling tools [10], such as, ArgoUML [1],

StarUML [3], BOUML [2], Rational Software Architect [22]. UMPLE's code generator ensures multiplicity constraints and referential integrity. Also, it automatically provides a rather complete API to the classes holding an attribute that represents an association: `set`, `get`, `numberOf`, `has`, `indexOf`, `add`, `remove`, `addAt`, and `delete`.

To analyze the differences of how UMPLE and TouchCORE handle associations, we created a simple model in both tools, where a class `User` is associated with a class `Account` by means of a bidirectional, 0..1 association. Figure 6–1 shows the visual representation of the two classes in UMPLE, as well as the textual UMPLE notation, and Figure 6–2 depicts the same model in TouchCORE. To compare UMPLE's generated code with TouchCORE's, we generated the corresponding Java code in both tools

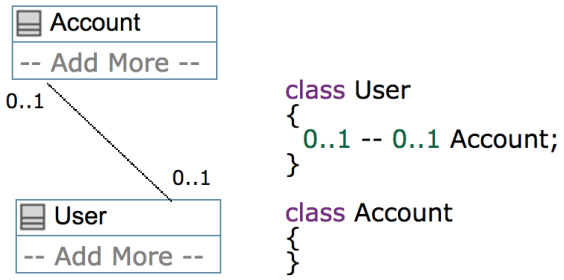


Figure 6–1: UMPLE Bidirectional Association and its Textual Syntax

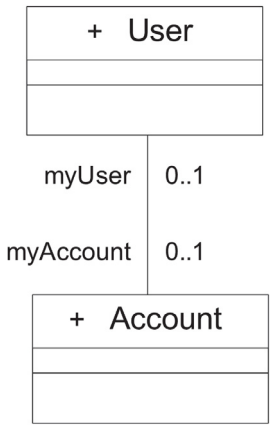


Figure 6–2: TouchCORE Bidirectional Association

Algorithm 11, shows the code for the `setUser` operation in UMPLE. The algorithm takes care of ensuring referential integrity. It first sets its user through `user = null` or `user = aNewUser` before calling `setAccount` on the current user and/or the new user. That way,

when `setAccount` is being called, it verifies that the state of the account was already updated and prevents a loop.

We decided to handle referential integrity in TouchCORE a little differently than UMPLE does. We used helper functions: `setSimple`, `addSimple` and `removeSimple`. As shown in the generated code in Algorithm 12 from the class diagram in Figure 6–2, `setMyUser` initiates the set and calls `setSimpleMyAccount`, not `setMyAccount`. By breaking it down to a `set` and a `setSimple`, both operations are shorter than UMPLE's set and a lot clearer to understand. This is desirable in our concern because the user is presented with the behaviour and should be able to easily figure out what is happening whereas UMPLE's generated code should not be looked at by the user.The set operation, both in the case of UMPLE and TouchCORE, returns a `boolean`, even if a set on a 0..1– 0..1 association never fails and always returns true. This is the case because in different types of associations, the `set` might return false. For example, if the set needs to insert an element on the opposite end that has already reached it's maximum. Since the signature of the operation should remain the same in all cases for consistency, a boolean is always returned.

Finally, UMPLE does not support qualified associations (feature KeyIndexed in TouchCORE), and always translates a many association to a fixed implementation data structure (ArrayList in Java, a Vector in C++, an array in ruby) without determining the best fit or letting the user decide. TouchCORE provides different implementations as well as impact evaluation. UMPLE does not provide the property unique and all associations are ordered since they all translate to a list in the code. It does provide sorted associations and allows to specify the attribute for which to sort the objects by.

On the other hand, UMPLE supports code generation to Java, C++, Python and Ruby whereas TouchCORE only supports Java.

Algorithm 11 UMPLE Generated Code for Operation setAccount

```
public boolean setUser(User aNewUser) {
  boolean wasSet = false;
  if (aNewUser == null) {
    User existingUser = user;
    user = null;

    if (existingUser != null && existingUser.getAccount() != null) {
      existingUser.setAccount(null);
    }
    wasSet = true;
    return wasSet;
  }

  User currentUser = getUser();
  if (currentUser != null && !currentUser.equals(aNewUser)) {
    currentUser.setAccount(null);
  }

  user = aNewUser;
  Account existingAccount = aNewUser.getAccount();

  if (!equals(existingAccount)) {
    aNewUser.setAccount(this);
  }
  wasSet = true;
  return wasSet;
}
```

Algorithm 12 TouchCORE Generated Code for Operation setMyUser

```
public boolean setMyUser(User newUser) {
  User oldAssociated;
  oldAssociated = getMyUser();
  if (oldAssociated != null) {
    oldAssociated.setSimpleMyAccount(null);
  }
  a.setSimpleMyAccount(this);
  this.myUser = newUser;
  return true;
}

protected boolean setSimpleMyUser(User newUser) {
  User oldAssociated;
  if (newObject != null) {
    oldAssociated = getMyUser();
    if (oldAssociated != null) {
      oldAssociated.setSimpleMyAccount(null);
    }
  }
  this.myUser = newUser;
  return true;
}
```

Chapter 7
Conclusion And Future Work

This thesis presented a way of dealing with associations in model-driven engineering with an innovative technique that is concern-oriented reuse. Concern-orientated reuse aims at reducing development time for modellers by allowing them to reuse other models called *concerns* that encapsulate and solve problems related to a particular domain. Associations, because they are widely used in software design and have many variants, appeared like a great candidate to address within a concern.

We designed many relevant variations of associations and association implementations, and encapsulated them within an *Association* concern. We elaborated a *variation interface* for the concern that lists the encapsulated variants in a comprehensible way to any user who would wish to use associations within his models. Experiments were run to establish the memory use and performance of different collection classes used to implement associations, and documented in impact models. While in UML the visualization of an association in a class diagram is simple, some structure and behaviours are implied from the modelled properties, e.g., multiplicity and navigability. This structure was encapsulated in class diagrams, and the behaviours ensuring multiplicity constraints and referential integrity were specified using sequence diagrams and encapsulated within the concern. Complex behaviour resulting from feature interactions was dealt with using conflict resolution models.

In addition to building the *Association* concern, this thesis also proposes techniques to streamline its reuse. A DSL was defined extending the UML notation of associations in order

to reuse the concern in a fast and concise manner within class diagrams. The user still needs to be presented with the variation interface when making a selection in order to choose the collection to use. However, feature selections are essentially done automatically based on the properties of the association. Furthermore, when the association is created between two classes, all the customization mappings required to properly reuse the *Association* concern are created automatically.

This DSL, the automated feature selection and creation of mappings was implemented in the TouchCORE tool. Changes had to be done to the metamodel and to the weaver. The association visualization was extended to support the new elements: the multiplicity element displaying the chosen association variant, such as `{ordered}`, and the selection of the qualifier in the case of a qualified association.

Future Work

Even though our concern contains several concrete implementation classes, the concern could encapsulate more, such as heaps or graphs. Since the currently supported implementation classes are from the Java standard library, there are currently a limited amount of options. Additional implementation classes from other Java libraries could be used, e.g., the Apache Commons Java Collections [11] or Google's Guava Collections Library [19]. Apart from storing elements in a collection in memory, users might want to use a database to persist associations. It would be possible to add a *Database* feature to the Association concern with sub-features of different types of databases. Moreover, we could add other features that would represent currently unsupported properties of association implementations, such as *Thread-safe*.

Further research could focus on adding support for automatic selection of the implementation class to TouchCORE. Making a complete selection for all associations used within a model is time consuming for the user. It is also the kind of information a user might not want to decide on while modelling. As mentioned in the automated code generation paper [35], selecting a concrete implementation class automatically from how it is used in the model could be a great step for optimization. For example, by analyzing the behaviour of the application, we could determine whether the collection is often iterated over, and then select a data structure that better supports iteration to improve the non-functional qualities of the code. As an alternative to automated optimization based on application behaviour, users could also provide a priority ranking of the different non-functional goals according to what matters to them most, and the tool would then determine the feature selections that would maximize these goals.

References

[1] ArgoUML. http://argouml.tigris.org/.

[2] BOUML. http://www.bouml.fr/.

[3] StarUML. http://staruml.io/.

[4] Kapil Viren Ahuja Ahuja. Performance Evaluation | Java Collections Framework. https://scrtchpad.files.wordpress.com/2008/10/java-collections-performance-evaluation.pdf, 2008.

[5] D. Akehurst, G. Howells, and K. McDonald-Maier. Implementing associations: Uml 2.0 to java 5. *Software & Systems Modeling*, 6(1):3–35, 2006.

[6] Wisam Al Abed and Jörg Kienzle. Aspect-Oriented Modeling and Information Hiding. In *14th Aspect-Oriented Modeling Workshop, Denver, CO, USA*, pages 1–6, 10 2009.

[7] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. *Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 – October 4, 2013. Proceedings*, chapter Concern-Oriented Software Design, pages 604–621. 2013.

[8] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Concern-Driven Software Development. Technical Report SOCS-TR-2015.1, McGill University, Montreal, Canada, January 2015.

[9] 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 Demo and Poster Session co-located with ACM/IEEE 18th International Conference on MoDELS 2015*. CEUR-WS.org, 2015.

[10] Omar Badreddin, Andrew Forward, and Timothy C. Lethbridge. Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity. In Roger Lee, editor, *Software Engineering Research, Management and Applications*, volume 496 of *Studies in Computational Intelligence*, pages 129–149. 2014.

[11] Apache Commons. Commons Collections. `https://commons.apache.org/proper/commons-collections/`.

[12] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39:41–47, 2006.

[13] Robert Eckstein. Java SE Application Design With MVC. March 2007.

[14] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering*, pages 37–54. IEEE, 2007.

[15] 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.

[16] Gonzalo Génova, Carlos Ruiz del Castillo, and Juan Llorens. Mapping UML Associations into Java Code. *The Journal of Object Technology*, 2(5):135–162, October 2003.

[17] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, October 2007.

[18] Dominik Gessenharter. *Model Driven Engineering Languages and Systems: 11th International Conference, MoDELS 2008. Proceedings*, chapter Mapping the UML2 Semantics of Associations to a Java Code Generation Model, pages 813–827. 2008.

[19] Google. Guava. `https://github.com/google/guava`.

[20] Object Management Group. Unified Modeling Language (UML). In *Superstructure, Version 2.5*, pages 32–35. March 2015.

[21] William Harrison and Charles Barton. Mapping UML designs to Java. In *proceedings of the 15 th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 178–188. ACM Press, 2000.

[22] IBM. Rational Software Architect. `http://www-03.ibm.com/software/products/fr/ratisoftarch`.

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

[24] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[25] Jörg Kienzle, Wisam Al Abed, Franck Fleurey, Jean-Marc Jézéquel, and Jacques Klein. *Transactions on Aspect-Oriented Software Development VII: A Common Case Study for Aspect-Oriented Modeling*, chapter Aspect-Oriented Design with Reusable Aspect Models, pages 272–320. 2010.

[26] Object Management Group (OMG). UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, June 2011.

[27] Matthias Schöttle. Aspect-Oriented Behavior Modeling In Practice. Master's thesis, Department of Computer Science, Karlsruhe University of Applied Sciences, September 2012. Conducted at the School of Computer Science, McGill University, Montreal, Canada.

[28] 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.

[29] Matthias Schöttle, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. On the Modularization Provided by Concern-Oriented Reuse, 2016. To appear in Proceedings of MODULARITY Companion 2016 - Workshop on Modularity in Modelling (MOMO 2016) co-located with the 15th International Conference on Modularity (MODULARITY 2016), March 2016.

[30] 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 '14, pages 23–28. ACM, 2014.

[31] 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, pages 11–14, New York, NY, USA, 2015. ACM.

[32] EJ Technologies. JProfiler. `https://www.ej-technologies.com/products/jprofiler/overview.html`.

[33] 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 MoDELS 2014, Valencia, Spain, October 1st and 2nd, 2014*. CEUR-WS.org, 2014.

[34] McGill University. TouchCORE. `http://touchcore.cs.mcgill.ca/`.

[35] T. Weigert, F. Weil, A. van den Berg, P. Dietz, and K. Marth. Automated Code Generation for Industrial-Strength Systems. In *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pages 464–472, July 2008.