# Associations in MDE: A Concern-Oriented, Reusable Solution

Céline Bensoussan(✉), Matthias Schöttle, and Jörg Kienzle

School of Computer Science, McGill University, Montreal, Canada
{Celine.Bensoussan,Matthias.Schoettle}@mail.mcgill.ca,
Joerg.Kienzle@mcgill.ca

**Abstract.** Associations play an important role in model-driven software development. This paper describes a framework that uses Concern-Oriented Reuse (CORE) to capture many different kinds of associations, their properties, behaviour, and various implementation solutions within a reusable artifact: the *Association* concern. The concern exploits aspect-oriented modelling techniques to modularize the structure and behaviour required for enforcing uniqueness, multiplicity constraints and referential integrity for bidirectional associations. Furthermore, it packages different collection implementation classes that can be used to realize associations. For each implementation class, the impact of its use on non-functional qualities, e.g., memory consumption and performance, has been determined experimentally and formalized. We show how the class diagram notation, i.e., its metamodel and visual representation, can be extended to support reusing the *Association* concern, and present enhancements to automate feature selection and customization mappings to maximally streamline the reuse process in modelling tools.

## 1 Introduction

Model-Driven Engineering (MDE) [6] is a unified conceptual framework in which software development is seen as a process of *model production*, *refinement,* and *integration*. To reduce the accidental complexity and the effort needed to move from a problem domain to a software-based solution, MDE advocates the use of different modelling formalisms, i.e., modelling languages, to represent and analyze the system from *multiple points of view*. For each level of abstraction, the modeller uses the best formalism that concisely expresses the properties of the system that are important to that level. 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 [8].

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*.


A. Wąsowski and H. Lönn (Eds.): ECMFA 2016, LNCS 9764, pp. 121–137, 2016.
DOI: 10.1007/978-3-319-42061-5_8

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,* 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 [2,4,9,11,12]. 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, such as 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.

In this paper we describe a framework for dealing with associations in the context of MDE. We show how we used Concern-Oriented Reuse (CORE) [3] to capture many different kind of associations, their properties, behaviour, and various implementation solutions within a reusable artifact: the *Association* concern. The *Association* concern encapsulates models for many association variants, and exploits aspect-oriented modelling techniques to modularize the structure and behaviour required for enforcing uniqueness, multiplicity constraints and referential integrity for bidirectional associations. Furthermore, it packages several collection implementation classes that can be used to realize associations. For each provided implementation class, the impact of its use on memory consumption and performance has been experimentally determined and formalized within the concern.

The remainder of the paper is structured as follows. Section 2 reviews the essential background on CORE. Section 3 describes how we designed the *Association* concern. Section 4 presents how to streamline the reuse of the *Association* concern within a modelling tool. Section 5 discusses related work, and the last section draws our conclusions.

## 2 Background on Concern-Oriented Reuse

CORE [3] is a new software development paradigm inspired by the ideas of multi-dimensional separation of concerns [22]. It builds on the disciplines of MDE, software product lines (SPL) [18], goal modelling [13], and advanced modularization techniques offered by aspect-orientation [15,19] to define flexible software modules that enable broad-scale model-based software reuse called *concerns.*

A CORE *concern* is a unit of reuse that groups together software artifacts (models and code, henceforth called simply models) that address a recurring domain of interest in software development. The models encapsulated within a concern capture in a *generic* way the structural properties and behaviour of all relevant variations and ways of dealing with the domain of interest at all relevant levels of abstraction. Building a concern is a non-trivial, time consuming task, done by the *concern designer*, who is an expert of the concern's domain. Deep understanding of the nature of the concern is required to be able to identify its user-relevant features, to model the common properties and differences of all features of a concern at all relevant levels of abstraction, and to express the impact of the different variants on high level stakeholder goals and qualities. This is ensured by creating requirements, design and implementation models that (i) realize the features of the concern using the most appropriate modelling notations and programming languages, and (ii) are eventually refined into executable specifications.

### 2.1 The CORE Reuse Process

The concern designer elaborates *three interfaces* [3] for a concern:

- The *Variation Interface* describes the available variations of the concern and the impact of different variants on high-level stakeholder goals, qualities, and non-functional requirements. The variations are typically represented with a *feature model* [14] that specifies the individual, user-relevant features that a concern offers, as well as their dependencies, e.g., *optional*, *alternative*, *requires*, and *excludes*. The impact of choosing a feature is specified with impact models, which are based on GRL [13].
- The realization of each variant of a concern is described as generally as possible to increase reusability. Therefore, some model elements are only *partially* specified and need to be complemented with concrete modelling elements stemming from the application models that intend to reuse the concern. These generic elements are exposed in the *Customization Interface*.
- The *Usage Interface* describes how the application can finally access the structure and behaviour provided by the concern, similar to what the set of public operations represents for a class in the object-oriented paradigm.

The *concern user* reuses an existing concern through three simple steps:

1. The *concern user* first *selects the set of feature(s)* (called a *configuration*) with the best impact on relevant stakeholder goals and system qualities *from the variation interface* of the concern based on impact analysis provided by the CORE tool. Using this configuration, the CORE tool then composes the models that realize the selected features to yield new models of the concern corresponding to the desired configuration.
2. Next, the *concern user adapts* the generated realization *models to the application context by mapping customization interface elements* to application-specific model elements. Again, the CORE tool helps to establish correct

mappings based on the signatures of the model elements that have to be customized, and subsequently generates customized realization models.
3. Finally, *the concern user uses the functionality exposed in the usage interface* of the customized realization models within his application models.

To demonstrate our framework, we use TouchCORE[1] [21], a multi-touch enabled, software design modelling tool that supports feature and impact models, as well as realization models expressed using class, sequence, state diagrams, and Java implementations.

## 3 Designing the Association Concern

In this section we present the design of the *Association* concern, which encapsulates all relevant variants of dealing with *unidirectional, binary associations* between two entities in MDE[2]. We start by describing the variation, customization and usage interfaces of the concern, follow up with an overview of the structural and behavioural realization models encapsulated within the concern, and finally describe the experiments that we ran to determine the impact of different association realization on memory use and performance.

### 3.1 Association Variation Interface

Coming up with a variation interface for a concern requires (i) breaking down the domain into distinct *features*, i.e., modules that provide well-defined *user-relevant* structure, functionality and/or properties, and organizing the features and their relationships in a feature model, and (ii) identifying the non-functional qualities that the realizations of the features might impact. Usually, the variation interface of a concern is not elaborated in a top-down manner. Rather, the expert domain knowledge of a concern designer typically allows her to sketch an initial variation interface, which is then refined as more insight is gained while realizing the features. Figure 1 shows the final variation interface of the *Association* concern.

**Structure:** The first mandatory sub-feature, *Structure*, differentiates between an association with a maximum multiplicity of *One* (single object) and associations with a multiplicity of more than one, i.e., *Many* (collection of objects). The feature *One* is therefore used for multiplicities of `0..1` and `1..1`. Among the associations with multiplicity *many*, there are qualified associations, where objects in the association are retrieved using a key (feature *KeyIndexed*), and *Plain* associations, which can be *Ordered* or *Unordered*. The leaf features finally encapsulate different data structures and algorithms that implement the collections with the corresponding properties, namely *ArrayList*, *LinkedList* and *Stack*

[1] http://touchcore.cs.mcgill.ca.
[2] Bidirectional associations are supported as well by using two unidirectional associations between the same elements in opposite direction.

for *Ordered* collections, *HashSet* and *TreeSet* for *Unordered* ones, and *HashMap* and *TreeMap* for *KeyIndexed*.

**Association Properties:** Associations are *Bidirectional* when they are navigable in both directions, in which case referential integrity must be enforced. For associations with multiplicity *Many*, it makes sense to decide whether the same element can be part of the association more than once or not. The optional feature *Unique* ensures that adding an object to an association is only allowed if the object is not already part of the association. Since the implementation data structures that we use for unordered collections—*HashSet* and *TreeSet*—do not support duplicate insertion of the same object (i.e., they implement *Sets* and not *Bags*), we specified the constraints that *TreeSet* requires *Unique,* and *HashSet* requires *Unique* within the feature model. Finally, the *Minimum* and *Maximum* features constrain the behaviour of *insertion*/*removal* operations to enforce minimum and maximum multiplicity constraints. They are sub-features of *Plain*, because they cannot be used in combination with qualified associations.

**Impacts:** The different variations of association implementations encapsulated inside the *Association* concern have an impact on memory use and performance. We modelled the impacts with the following goals: *Minimize Memory Footprint, Increase Insertion Performance*, *Increase Iteration Performance*, *Increase Access Performance* and *Increase Removal Performance*, as shown on the right side of Fig. 1. To determine the weights that drive the evaluation of the impacts based on a feature selection, we ran an extensive set of experiments that are described in Subsec. 3.6.

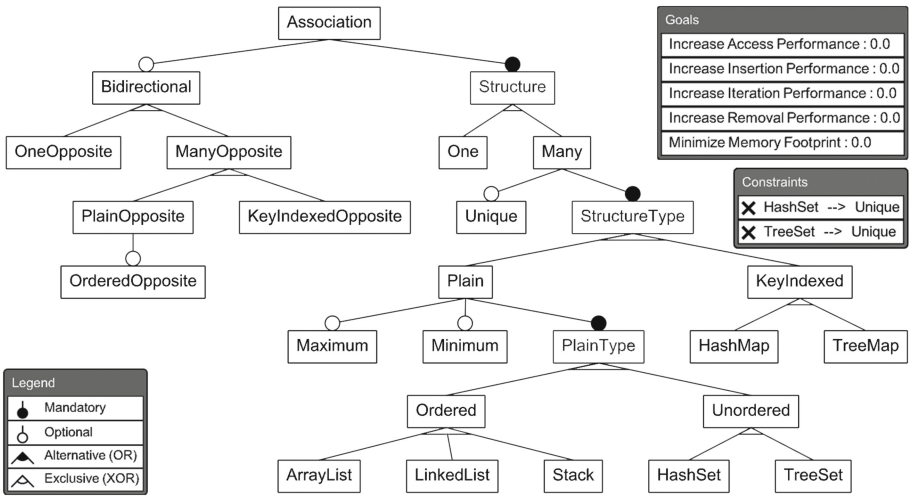


**Fig. 1.** Screenshot of variation interface of the *Association* concern

### 3.2 Customization Interface

The *customization interface* of a concern exposes the model elements that define only partial structure/behaviour. They need to be adapted by the concern user to the reuse context by mapping them to concrete model elements in the application model. To easily identify model elements that have to be customized by a concern user, the names of these *public partial* model elements are prefixed with a vertical bar ("|").

In a directed association, partial structural elements are the class of origin, i.e., the class that holds the association end, and the destination class. We named the class of origin `|Data` and the destination class `|Associated` as shown in Fig. 2 on the right. For qualified associations, the customization interface includes an additional partial `|Key` class as shown in Fig. 2 on the left.


**Fig. 2.** Customization interface for feature *KeyIndexed* (left) and others (right)

### 3.3 Usage Interface

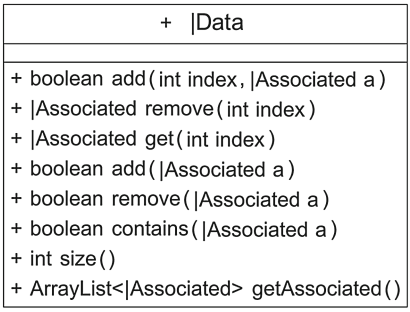


**Fig. 3.** Usage Int. for *ArrayList*

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 the `|Data` class and its *public operations*. The features of the concern do not have a common *usage interface,* as the operations of `|Data` vary with the properties of the collection. When `|Data` holds a single object reference (feature *One*), the *usage interface* consists of a *getter* and a *setter* operation. When `|Data` holds a collection (feature *Many*), it provides operations to *add* and *remove* elements. For ordered associations (feature *Ordered*), additional operations to *add* and *remove* at a specific index are provided. For example, the usage interface for the feature *ArrayList* is shown in Fig. 3. For qualified associations, the *add* and *remove* operations take as an additional parameter a key.

Since `|Data` is part of the customization interface, it is mapped by the user to the class holding the association. As a result, the operations belonging to the usage interface of `|Data` are added to the mapped class, ready to be used. The operations, though, are not part of the customization interface, i.e., they *do not have to be* mapped. However, the user *may want to rename* the operations for better usability, for example, rename `add` to `addUser`.

### 3.4 Structural Realization of Associations

In CORE, each feature is associated with realization models that describe its structural and behavioural properties at different levels of abstraction using different modelling formalisms. When a concern user makes a feature selection, the CORE tool incrementally composes all realization models associated with the selected features to create user-tailored realization models. In this subsection, we describe class diagrams encoding different structural variations of the *Association* concern.

The realization model of the root feature of the concern simply defines the two classes `|Data` and `|Associated` that we already introduced above. The realization model of the feature *One*, which is used when the upper multiplicity bound of an association end is *1*, declares a reference `myAssociated` pointing from `|Data` to `|Associated`. It also defines a *getter* and a *setter* operation for this reference. On the other hand, the realization model for the feature *Many*, which is used when the upper bound of the association is greater than *1*, defines a `¦CollectionOfAssociated` class that is contained in the class `|Data`. It is marked as *concern partial* with a discontinuous vertical bar (“¦”), which means that it is incomplete just like model elements that are part of the customization interface of the concern. However, it has to be completed *within the concern*, i.e., by other realization models. The realization model of *Many* also defines the operations *contains*, *size* and *getAssociated*.

The structure is further refined by the realization model of feature *Plain*, which defines operations to *add* and *remove* elements to/from the `¦CollectionOfAssociated` class. Continuing, the realization model of *Ordered* adds operations to *add*, *remove* and *get elements* at a certain index. Finally, the realization model for features representing concrete implementation data structures map the `¦CollectionOfAssociated` class to a concrete Java class, e.g., `ArrayList`.

### 3.5 Behavioural Realization

We modelled the behaviour of operations using sequence diagrams. Figure 4 shows the `add` operation defined in *Plain*, which calls `add` of the contained collection.

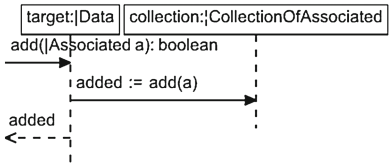


**Fig. 4.** Base behaviour of `add`

Some features of the *Association* concern may affect the behaviour of other features. For example, the feature *Unique* affects 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.

CORE uses aspect-oriented techniques to augment the behaviour of other realization models. For example, Fig. 5 shows how *Maximum* extends the behaviour of Fig. 4 to verify that the maximum has not been reached before executing the original behaviour of `add` (represented by a white box containing a "*").

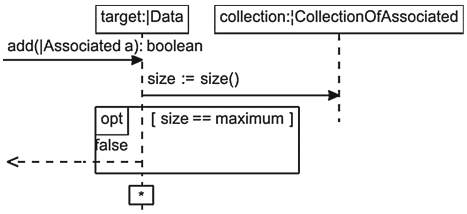


**Fig. 5.** Aspect sequence diagram *Maximum*

Additional complexity stems from the fact that there are some behavioural feature interactions inside the *Association* concern that need to be taken care of. For example, the behaviour of the feature *Bidirectional* requires that before a new object is associated with a current object, the object might first need to be removed from other associations, and the current object has to be added to the opposite association of the new object, and only then the new object can be added to the association of the current object. However, the operations that need to be called to deal with the opposite end of the association depend on the multiplicity constraints on the opposite end. In certain cases, *setter* operations should be invoked, in other cases, *add/remove* operations. These different behaviours had to be specified in so-called feature interaction resolution realization models, which are linked to the features they deal with, so that the CORE tool can apply them automatically when needed.

For example, Fig. 6 shows the feature interaction resolution model for *Plain* and *OneOpposite*, which ensures that for bidirectional `0..1 <-> 0..*` associations a new `|Associated` object `a` is only then added to the collection in the `target` object `|Data`, if `target` was successfully set as the opposite associated object of `a`. To ensure that this resolution is combined in the correct order with the behavioural modification that realizes *Maximum,* as shown previously, an additional feature interaction model has to be defined that first applies *Plain/OneOpposite*, and then *Maximum*.

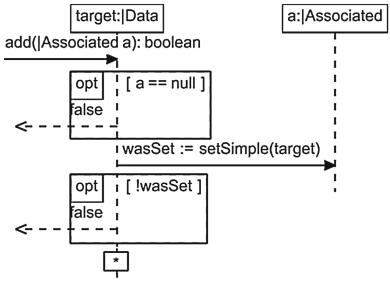


**Fig. 6.** Interaction Res. *Plain/OneOpposite*

### 3.6 Determining the Impacts of Association Realizations

In order to provide the modeller with guidance on which of the association features to choose, we conducted a series of experiments to determine the impact that the different realizations have on memory use and performance.

**Experimental Setup:** We ran our experiments on a machine with a 2,4 GHz Intel Core i5 processor and 16 GB 1600 MHz DDR 3 memory. The machine was running Mac OS X 10.9.5. The Java SE Runtime (v1.8.0_20-b26) was configured with 384 MB heap space. The model that was 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`.

**Impact on Memory Use:** To determine the amount of memory used by the different realizations, we created $n$ instances of `B` ($n$ = 10 (small), $n$ = 100 (medium), $n$ = 1,000 (large) and $n$ = 10,000 (extra-large), and added them to the association between `A` and `B` by successively calling `a.addMyB(b`$_i$`)`. We used the Heap Walker of JProfiler [7] to determine the amount of memory used by the collection implementation class realizing the association. The results are shown on the left side of Fig. 7.

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

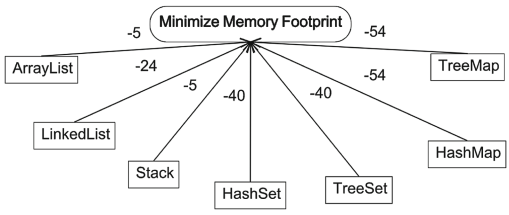


**Fig. 7.** Memory usage in bytes and corresponding impact model

The relative measured memory use is approximately consistent across different orders of magnitude of number of elements. We therefore used the measured number of KB for 1000 elements directly as negative contribution values in the corresponding CORE impact model (see the right side of Fig. 7). This means that *ArrayList* and *Stack* (with contribution *-5*) are from a memory use point of view the best choice, whereas *HashMap* and *TreeMap* (with contribution *-54*) are the worst choice, i.e., they use approximately 10 times more memory.

**Impact on Performance:** To measure the impact on performance, we used an approach similar to the one described in Ahuja [1]. Again, we ran experiments with associations of different orders of magnitude (#elements = $n$), and measured the time $t$ it took to execute each operation `op` $n$ times from within a loop. Measuring Java performance is not trivial, because of various factors involving

the virtual machine, the garbage collector, actual heap size at runtime and associated non-determinism [10]. To minimize external influences, we refrained from measuring the first runs to avoid accounting for time spent loading/initializing code, and then collected measurements of 50 runs. From those runs we calculated the *median* as well as the *10th and 90th percentile* to minimize effects of the garbage collector.

The performance measurements for *adding/appending n* objects to an association are shown in Fig. 8[3]. Some implementations perform consistently well, e.g., *ArrayList* and *LinkedList*, and others consistently bad, e.g., *TreeSet*. However, the relative performance of some varies depending on the order of magnitude of the number of elements in the association. For example, *HashSet* and *HashMap* perform well for a small number of elements, but then performance worsens for larger associations. We therefore decided to create separate impact models for each order of magnitude using the median values from the experiments as negative weights for the impact models.

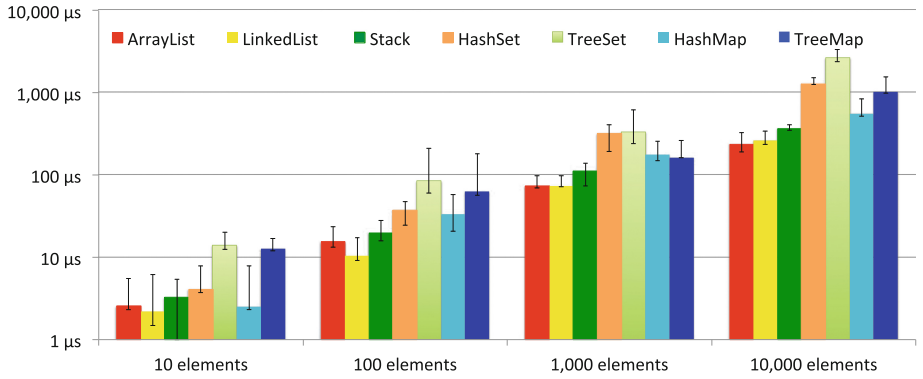


**Fig. 8.** Insertion performance of different collection implementations (Color figure online)

**Discussion:** Impact models in CORE are currently exclusively specified using the goal modelling notation [13]. 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 to more quantifiable qualities, e.g., *cost* and *number of messages exchanged*. Unfortunately, impact models as they are defined currently 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,

[3] The results for the other operations, i.e., access performance, iteration, and removal, are not shown for space reasons. They are available in [5], which also describes additional experiments that we ran to compare performance on different Java execution platforms.

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 [17], but how to exploit these in the context of CORE is out of the scope of this paper.

### 3.7 Association Concern Design Summary

In the end, the *Association* concern we designed encapsulates 26 features, specifies 5 impacts, contains 10 class diagrams, and 25 sequence diagrams (3 of which are feature interaction resolution models). The feature model allows for 225 possible selections, from which the TouchCORE tool can create 225 different user-tailored realization models by combining the corresponding realization models in different ways to suit the exact needs of the concern user.

## 4 Using the Association Concern

The complexity of associations (different variations and implementation classes, impacts, behaviour ensuring maximum, minimum, uniqueness, and bidirectionality,and additional behaviour addressing feature interactions) is now encapsulated behind the variation, customization and usage interface of the *Association* concern and ready to be reused.

The standard CORE reuse process, outlined in Subsect. 2.1 and implemented in the TouchCORE tool, is general, i.e., it is applicable when reusing *any concern*. It can therefore also be used for reusing the *Association* concern. Unfortunately, due to its general nature, the process is unnecessarily tedious for *Association*. In TouchCORE, it involves the following effort for the modeller:

1. The modeller first needs to indicate the desire to reuse *Association*. This involves searching through the reusable concern library to find the *Association* concern, which typically involves navigating down the folder hierarchy.
2. When the *Association* variation interface is displayed, the modeller must make a selection of the desired variant. The feature model of *Association* is large, in particular because of the features that deal with ensuring the correct behaviour for bidirectional associations. It takes cognitive effort to visually browse through it and make the desired selection.
3. When the *customization interface* is displayed, the modeller has to manually establish the mappings of the source and destination classes of the association: `|Data` and `|Associated` have to be mapped, as well as `|Key` in case of qualified associations. The mappings of the operations are not required, but in case the modeller desires to rename the generic names of operations to more specific names, e.g., *add* to *addUser*, mappings have to be specified for each operation that is to be renamed.

Finally, a bidirectional association requires reusing the *Association* concern *twice*. This not only constitutes a duplication of effort, but it is also a potential

source of inconsistencies. In order to avoid errors, the modeller must make sure to select the right sub-feature of *Bidirectional* that correctly represents the type of the opposite association (one, many, ordered, or key-indexed).

In light of these usability issues, we devised a domain-specific language (DSL) inspired by the concrete syntax for associations defined in UML to streamline the reuse of the *Association* concern for modellers. The DSL minimizes the effort involved and eliminates any risk of mis(re)use. We then integrated this DSL into the TouchCORE tool in order to facilitate the reuse of the *Association* concern while modelling with class and sequence diagrams.

### 4.1 DSL for Applying the Association Concern

UML already defines a visual notation for associations [16]. A line that connects two classes represents an association, arrowheads on the association ends depict navigability, and inclusive intervals of non-negative integers on the association ends specify a lower bound and a (possibly infinite) upper bound for multiplicities. The default properties for associations in UML are *unique* and *unordered*. It is possible to specify deviations from the default by annotating the association ends with textual constraints, i.e., {*ordered*} and/or {*nonunique*}. For qualified associations, the UML syntax dictates that the type of the model element used for lookup is specified in a rectangular box at the border of the originating class.

Since the graphical notation in UML already covers our features *Bidirectional*, *Minimum*, *Maximum*, *Unique*, *Ordered*, *Unordered*, and *KeyIndexed*, we simply defined additional textual constraints to allow the modeller to specify the concrete implementation classes, i.e., *ArrayList*, *LinkedList*, *Stack*, *HashSet*, *TreeSet*, *HashMap* and *TreeMap*. This list is automatically extended whenever additional implementation classes are added to the *Association* concern.

### 4.2 Modifications to the Class Diagram Metamodel

In the CORE metamodel [20], the *COREReuse* class represents reuses. From a *COREReuse* one can get to the *COREConfiguration*, i.e., the set of selected features of the reuse. The *CORECompositionSpecification*, i.e., the set of customization mappings can be retrieved through the *COREModelReuse*, which specifies the compositions of a reuse for a particular model. To use the *Association* concern consistently, every navigable association end has to have a corresponding model reuse of the *Association* concern. Hence, a directed association between *AssociationEnd*, i.e., the class that represents association ends in the class diagram metamodel, to *COREModelReuse* is needed. The backend of TouchCORE was updated to create a *COREReuse* and *COREModelReuse* (for the design model) whenever an association end between two classes becomes navigable.

### 4.3 Automated and Consistent Feature Selections

TouchCORE was adapted in such a way that whenever the modeller manipulates the graphical representation of an association, e.g., by changing the multiplicity

or navigability, the selected features of the reuse of *Association* are *updated automatically* as follows:

- When the upper multiplicity bound is `1`, *One* is selected, otherwise *Many*.
- When the upper multiplicity bound is greater than `1` and not many (`*`), *Maximum* is selected.
- When the lower bound is `1` or greater and the upper bound is greater than `1`, *Minimum* is selected.
- When the association is navigable in both directions and the upper bound on the multiplicity of the opposite end is `1`, *OneOpposite* is selected.
- When the association is navigable in both directions and the upper bound on the multiplicity of the opposite end is greater than `1`, *ManyOpposite* is selected.

Additionally, the GUI of TouchCORE was extended to display textual constraints, e.g., {*ordered*} or {*ArrayList*} on association ends. If the modeller clicks on the textual constraint, they are presented with a simplified variation interface of the *Association* concern. All automatically selected features are not shown, so the modeller can maximally focus on exploring the impact of the available implementation classes and to eventually select the most appropriate one.

### 4.4 Generation of Mappings and Operation Renaming

When a modeller draws a directed, navigable association from class `Source` to class `Destination`, the customization mappings for the *Association* concern are automatically created. `|Data` is mapped to `Source`, and `|Associated` to `Destination`. For qualified associations, TouchCORE displays a rectangular box at the association end that allows the modeller to specify the qualifier type. Based on the modeller's input, the corresponding mapping for `|Key` is created.

Additionally, for every operation that is in the usage interface of `|Data`, a mapping is created that renames the operation by appending the name of the association end specified by the modeller. For instance, for a directed association from class `User` to class `Account` with multiplicity `0..*` named `myAccounts`, the `add` operation would be renamed to `addToMyAccounts`.

## 5 Related Work

To our knowledge, the concern-orientated reuse paradigm is currently the only modelling approach that supports the encapsulation of different structural and behavioural designs and implementations and their impacts within one reusable model. As a result, most modelling tools provide only basic, "UML-like" support for modelling with associations. However, there is a substantial amount of related work on code generation optimized for and dedicated to associations.

### 5.1 Existing Code Generation Approaches for Associations

Harrison describes a technique for generating Java implementation code from UML diagrams [12]. The authors suggest generating an interface for dealing with the behaviour of associations (creating, deletion) in a manner transparent to the user. They propose 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 support for different collection implementation data structures. A similar approach is adopted by Gessenharter [11], who proposes that associations be implemented as classes. To implement an association between $A$ and $B$, a class $AB$ is created which holds a list of $AB$ links. Both class $A$ and $B$ have an `addB` and `addA` operation, respectively, that call a static method in $AB$ to establish a new link.

Génova presents some principles for mapping UML associations to Java code [9]. They demonstrate that it is unreasonable to ensure the minimum multiplicity constraint at any moment on a mandatory association end as it reduces usability. Therefore, they make the user responsible for initializing the system to a consistent state, and for maintaining it. Akehurst introduces Java code generation patterns from UML models with dedicated support for associations [2].

### 5.2 MouseTrap

Motorola has developed its own automatic code generation tool suite called *Mousetrap* [23]. The *Mousetrap* tool suite takes as input design models using SDL, UML, ASN.1, and ISL (a proprietary protocol language) and produces highly optimized C 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.

Section 5.4 of [23] on Abstract Data Types (ADT) is most related to our work. In their approach, code generation for associations involves the selection of a concrete implementation of an ADT. Where most code generators simply pick a default implementation, theirs analyzes the behaviour of the model and determines the specific ADT 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 linked lists have superior iteration performance due the lack of repeated indexing, a fact that our own benchmark measurements confirmed.

### 5.3 UMPLE

UMPLE (UML Programming Language) is a textual design modelling tool supporting class diagrams and state diagrams [4]. It has a powerful code generator that handles multiplicity constraints and referential integrity for associations just like we do.

From a user's point of view, the main differences between UMPLE and TouchCORE with the *Association* concern is that UMPLE 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. As a result, UMPLE does not provide the property *unique,* and all generated association implementations are *ordered* (since they all translate to a list in the code). However, UMPLE does provide sorted associations, and allows the modeller to specify the attribute that is to be used for sorting.

### 5.4 Discussion

One could argue that an advantage of the code generator approach over the CORE approach is that it clearly separates design decisions, which are made at the model level, from implementation decisions, which are made by the code generator (or by a platform expert that configures code generation options before running the code generator). However, this is not the case here, as the CORE reuse process allows a modeller to make partial selections. For example, it is acceptable for a designer to choose the feature *Ordered*, and *defer the decision* of which mandatory child feature from the XOR group—*ArrayList, LinkedList* or *Stack*—should be used in the realization. This decision can be made at a later point, potentially by a different developer, e.g., a platform expert. Ideally, the decision could even be automated based on some user-defined optimization criteria. Currently, though, the developer has to perform his own tradeoff analysis and opt for faster execution time or decreased memory usage depending on his preference. In the near future we are planning to build an automated reasoning system into the TouchCORE tool that exploits the impact information from the concern's variation interface to perform automated optimization of non-functional requirements according to the developer's priorities.

In the end, the main difference between addressing associations at the modelling level as done in CORE compared to dealing with associations during code generation is that if one desires to change the way that associations are handled or to support new association implementations, the latter approach requires understanding and modifying the code generator. In contrast, in our CORE approach the modeller can simply update the structural and/or behavioural realization models of existing features of the *Association* concern, or add new features and new realization models, if needed. There is no need to modify the code generation, nor modify any code in the TouchCORE tool.

Finally, while the code generators discussed in this section address the maximum, minimum, uniqueness and bidirectionality properties of associations just as well as we do, they typically do not support qualified associations as we do through the feature *KeyIndexed*. Finally, with the exception of *Mousetrap*, they do not take into account the non-functional impacts of different concrete data structure implementations.

## 6 Conclusion

In this paper we described a framework for dealing with associations in the context of MDE. We designed a reusable CORE concern named *Association* that encapsulates design models for different association variants, and exploits aspect-oriented modelling techniques to modularize the structure and behaviour required for enforcing uniqueness, multiplicity constraints, and referential integrity for bidirectional associations. Furthermore, it supports the use of different collection implementation classes used to implement associations and documents their impacts on memory consumption and performance. We showed how class diagrams, i.e., the metamodel and visual notation used in the TouchCORE tool, can be extended to support reusing the *Association* concern, and presented enhancements to automate feature selection and customization mappings to maximally streamline the reuse process.

## References

1. Ahuja, K.V.: Technical Whitepaper: Performance Evaluation | Java Collections Framework (2008). http://scrtchpad.files.wordpress.com/2008/10/java-collections-performance-evaluation.pdf
2. Akehurst, D., Howells, G., McDonald-Maier, K.: Implementing associations: UML 2.0 to Java 5. Softw. Syst. Model. **6**(1), 3–35 (2006)
3. Alam, O., Kienzle, J., Mussbacher, G.: Concern-oriented software design. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 604–621. Springer, Heidelberg (2013)
4. Badreddin, O., Forward, A., Lethbridge, T.C.: Improving code generation for associations: enforcing multiplicity constraints and ensuring referential integrity. In: Lee, R. (ed.) SERA 2013. SCI, vol. 496, pp. 129–149. Springer, Heidelberg (2013)
5. Bensoussan, C.: Associations in MDE: A Concern-Oriented, Reusable Solution. M.Sc. Thesis, School of Computer Science, McGill University, March 2016
6. Schmidt, D.C.: Model-driven engineering. IEEE Comput. **39**, 41–47 (2006)
7. EJ Technologies: JProfiler. https://www.ej-technologies.com/products/jprofiler/overview.html
8. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Future of Software Engineering, pp. 37–54. IEEE (2007)
9. Génova, G., del Castillo, C.R., Llorens, J.: Mapping UML associations into Java code. J. Object Technol. **2**(5), 135–162 (2003)
10. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous java performance evaluation. SIGPLAN Not. **42**(10), 57–76 (2007)
11. Gessenharter, D.: Mapping the UML2 Semantics of associations to a java code generation model. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 813–827. Springer, Heidelberg (2008)
12. Harrison, W., Barton, C.: Mapping UML designs to Java. In: OOPSLA, pp. 178–188. ACM Press (2000)
13. International Telecommunication Union (ITU-T): Recommendation Z.151: User Requirements Notation (URN) - Language Definition, October 2012
14. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report. CMU/SEI-90-TR-21, SEI, CMU, November 1990

15. Kienzle, J. (ed.): Transactions on Aspect-Oriented Development VII, Special Issue on a Common Case Study for Aspect-Oriented Modeling. Springer, Heidelberg (2010)
16. Object Management Group: Unified Modeling Language (UML) Superstructure, v. 2.5, pp. 32–35, March 2015
17. Object Management Group (OMG): UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, June 2011
18. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, New York (2005)
19. Filman, R., Elrad, T., Clarke, S., Akşit, M.: Aspect-Oriented Software Development. Addison-Wesley, Reading (2004)
20. Schöttle, M., Alam, O., Kienzle, J., Mussbacher, G.: On the modularization provided by concern-oriented reuse. In: Modularity in Modelling Workshop - MOMO 2016, MODULARITY Companion 2016, pp. 184–189. ACM (2016)
21. Schöttle, M., Thimmegowda, N., Alam, O., Kienzle, J., Mussbacher, G.: Feature modelling and traceability for concern-driven software development with TouchCORE. In: Companion Proceedings of MODULARITY, pp. 11–14. ACM (2015)
22. Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S.M.: N Degrees of separation: multi-dimensional separation of concerns. In: International Conference on Software Engineering - ICSE, pp. 107–119. IEEE (1999)
23. Weigert, T., Weil, F., van den Berg, A., Dietz, P., Marth, K.: Automated code generation for industrial-strength systems. In: COMPSAC 2008, pp. 464–472 (2008)