Aspect-Oriented Modelling with Instantiation Cardinalities illustrated using Software Design Patterns

Sunit Bhalotia

Master of Science

School of Computer Science

McGill University

Montreal,Quebec

2014-07-08

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

DEDICATION

This thesis is dedicated to all the students and professors who have been responsible for the success of Software Engineering Lab.

ACKNOWLEDGEMENTS

I thank my supervisor, Jörg Kienzle for his excellent support and guidance.

ABSTRACT

The power of aspect-oriented modelling is that a model designer can encapsulate the structural and behavioural properties of a specific concern within an aspect model. Theoretically, if the functionality provided by such a modularized concern is needed repeatedly within a system, a model user can instantiate the aspect model multiple times within the same target model. Practically, though, it is often not clear which model elements from the aspect model the model user is allowed to instantiate multiple times, and how the model weaver is supposed to compose the structural and behavioural elements of the aspect model instances with the target model. Motivated by past research that points out the pending issues related to multiple aspect model instantiations, this master thesis proposes to extend the customization interface of aspect models with instantiation cardinalities , a novel concept that allows the model designer to unambiguously declare which model elements of an aspect model can be multi-mapped. As a result, customization ambiguities are completely avoided for the model user. Furthermore, instantiation cardinalities give the model designer fine-grained control about how many instances of each structural and behavioural element contained in an aspect model are to be created in the target model. This thesis describes the syntax and semantics of instantiation cardinalities in detail, and shows how they integrate with object-oriented concepts such as inheritance and polymorphism. The elegance of the approach is illustrated by presenting aspect-oriented design models of six well-known and widely used behavioural, structural and creational design patterns.

ABRÉGÉ

La puissance de la modélisation orientée aspect est que le concepteur d'un modèle peut regrouper toutes les propriétés structurelles et comportementales d'une préoccupation particulière. En théorie, si la fonctionnalité que fournit un modèle aspect est utilisée à plusieurs endroits dans un même système, l'utilisateur de ce modèle aspect peut instancier le modèle plusieurs fois dans le même modèle cible. Cependant, en pratique, il s'avère que ce n'est souvent pas évident pour l'utilisateur de savoir quels éléments du modèle il a le droit d'instancier plusieurs fois et comment le tisseur de modèle doit les combiner avec le modèle cible dans ce cas là. En partant des problèmes connus de multi-instanciations des modèles aspects, cette thèse de maîtrise propose d'inclure des cardinalités d'instanciation dans l'interface de personnalisation d'un modèle aspect. Ce nouveau concept permet au concepteur d'un modèle de préciser exactement quels éléments peuvent être instanciés plusieurs fois, ce qui évite à l'utilisateur de faire des erreurs de personnalisation. De plus, en spécifiant des cardinalités d'instanciation, le concepteur du modèle peut contrôler le nombre d'instances de chaque élément du modèle aspect qui seront créées dans le modèle cible. Cette thèse décrit la syntaxe et la sémantique des cardinalités d'instanciation et montre comment les intégrer avec les concepts orientés objet tels que l'héritage et le polymorphisme. L'élégance de l'approche est démontrée en présentant les modèles aspects de six "design patterns" structurels et comportementaux bien connus.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
Introduction

Modelling allows the representation of entities and the relationships between them. It allows the expression of the meaning ideas used by experts to discuss problems. A model aims to provide clarity in the interpretation of the terms and concepts used. . The clarity is important since it allows efficient and unambiguous communication of ideas between the designers of the model and those who make use of it. Due to their visual representation, models also aid in finding relationships between the concepts.

Software systems are a natural candidate for modelling and this has given rise to various modelling approaches and notations including the widely used Unified Modelling Language (UML) [31]. In Software Engineering, when domain models are used as primary software development methodology, it is called Model-Driven Engineering (MDE) [17, 36]. MDE raises the level of abstraction by allowing a modeller to visualize the relationships between classes using diagrams. For example, in a UML class diagram, a modeller can describe the properties of classes and describe the relationships between them. As long as the class diagram stays relatively small (up to 10-15 classes), such a diagram fosters the understanding of what is being modelled and allows for faster communication between team-members, which leads to improved productivity of the development team. Model-Driven Engineering is also able to generate a significant percentage of code automatically which cuts down development time, reduces the possibility of errors and facilitates debugging.

1.1 Aspect-Oriented Modelling

Aspect-orientation adds a new dimension to modularization. In aspect-oriented modelling (AOM), advanced modularization techniques make it possible for aspect models to encapsulate structural and behavioural elements related to a particular concern. This allows the model designer to reason about all properties of relevance to the concern in isolation. AOM also draws special attention to the composition of concerns, which allows the modeller to focus on the intricacies of concern interactions and conflicts.

As a result, AOM has the potential to address two of the main challenges of model-driven engineering: model reuse and model scalability [38]. Building complex models is very time consuming: models are often created from scratch, as opposed to reusing existing models. This makes modelling often more cumbersome than coding, since most programming languages nowadays offer extensive libraries facilitating code reuse. Furthermore, models of complex applications tend to grow in size, to a point where the models are not readily understood or analyzable anymore. With advanced features for separation of concerns, the model can be decomposed into concern models of reasonable size. With advanced modularization techniques, the model can be packaged in a generic way so that it can be reused in different contexts. The potentially crosscutting nature of the concern requires that the structure and functionality provided by the model can be applied several times within the same application.

Many researchers have recognized this opportunity, and worked on aspect-oriented modelling approaches for many different modelling notations. For example, AOM approaches have been proposed for UML class diagrams [13, 35], sequence diagrams [21, 20], state diagrams [12, 40, 16], protocol models [24, 25], live sequence charts [23][23], activity diagrams, the Specification and Description Language (SDL) [10, 11], the User Requirements Notation (URN) [30, 29], and more.

However, only a small number of AOM approaches have concentrated their efforts on making aspect models reusable. In the context of reuse and MDE, a distinction needs to be made between *model designers* and *model user*s. Model designers are modellers that specify a reusable model, i.e., define its structural and behavioural properties. In AOM approaches that explicitly support reuse, the model designer also defines a model interface that clearly specifies how the model is intended to be (re)used [4]. Based on this interface, the model user can compose the reusable model with the application model to be able to use its functionality.

Different aspect-oriented modelling approaches provide different means to apply a reusable aspect within a target model. Some approaches require the specification of explicit mappings [35, 9, 19], whereas others allow the use of wildcards in so-called pointcut expressions [10, 39, 16]. Most approaches, though, have focussed mainly on composing a small number of models (typically two (!)), and illustrated their approach with academic examples. Rarely have these techniques been demonstrated on systems of considerable size, which would require several aspect models to be applied repeatedly within the same application. To the best of our knowledge, none of the current AOM approaches specifies precisely how a model user should go about applying a reusable model multiple times. As a result, the model user is faced with multiple possibilities: specifying multi-mappings or multiple individual mappings, or specifying a single complex pointcut expression vs. using several pointcut expressions. This is a problem, since it has been shown in [28] that in practice, the model designer of a reusable aspect model needs fine-grained control over how many instances of each reusable model element are created in the target model when an aspect is applied.

1.2 Thesis Contributions

This thesis presents a novel concept for aspect-oriented modelling: *instantiation cardinalities*. Instantiation cardinalities augment the interface of aspect models with a specification that describes precisely how the model is to be used and composed, even when the aspect's functionality is needed repeatedly within the target model.

Specifically, this thesis makes the following contributions:

- Instantiation Cardinality Concept
  - Instantiation cardinalities are introduced as a novel concept that is part of the model interface of an aspect model. The semantics of instantiation cardinalities allow the model designer to precisely specify how the model user is supposed to map model elements from the aspect model that is being reused to model elements in the target model. Specifically, the model designer can determine if model elements from the interface of the aspect can be multi-mapped, and if so, how many times (minimally and maximally). This solves customization ambiguities for users of aspect-oriented models that have been identified in previous research.
  - By declaring and using variables within the range declarations of the instantiation cardinalities, the model designer can express dependencies between the minimum and maximum number of mappings of structural entities encapsulated in the aspect model. Based on these dependencies, the model weaver can determine how many instances of each model element found in the aspect model are to be created in the target model, even in the case where the aspect is used multiple

times. With such fine-grained control, the model designer is able to specify all the instantiation policies deemed important according to [28].

- Most aspect-oriented modelling approaches and programming languages include object-oriented features as well. The thesis describes how to integrate instantiation cardinalities with standard object-oriented concepts such as inheritance and polymorphism. In particular, the thesis describes:
    * The effects that the cardinalities of a superclass and its methods have on its subclasses and the methods that they override,
    * How automated call forwarding can be applied by the weaver in order to allow for polymorphic treatment of multi-mapped subclasses in one model, while not requiremeng uniform naming of polymorphically related operations in each individual subclass.

• Integration of Instantiation Cardinalities with Reusable Aspect Models:
While instantiation cardinalities are a general concept, they are illustrated concretely in this thesis using the *Reusable Aspect Models* notation (RAM)) [19], an aspect-oriented multi-view modelling approach for software design modelling. The thesis proposes a concrete syntax for instantiation cardinalities, and shows how they can be used to augment the customization interface of RAM models to reap the aforementioned benefits.

- Design Pattern Case Study:

  The applicability and usefulness of instantiation cardinalities and their seamless integration with object-oriented concepts is demonstrated by showing the detailed aspect-oriented design models of seven well-known and widely used design patterns: the creational design patterns *Builder* and *Abstract Factory*, the structural design patterns *Composite* and *Decorator*, as well as the behavioural design patterns *Observer*, *Template Method* and *Command*.

1.3 Thesis Outline

This master thesis is structured as follows. Chapter 2 introduces the Reusable Aspect Models approach that is used in the rest of the thesis for illustration purpose. The issues that are caused due to multiple model reuse and instantiation ambiguities are reviewed by means of an example model that specifies the structural and behavioural design of the Observer Design Pattern.Chapter 3 proposes *Instantiation Cardinalities* as a solution to the problems stated in Chapter 2. The syntax and semantics of Instantiation Cardinalities are defined, and clear rules of how they are to be used in RAM by the model designer are developed . Automated Call Forwarding is introduced to enable integration of Instantiation Cardinalities with object-oriented concepts such as polymorphism. Chapter 4 demonstrates the applicability and usefulness of the Instantiation Cardinalities approach by showing extensive example models of commonly used design patterns, and how they can be applied in practice. The modelled design patterns are selected to evenly represent creational, structural and behavioural patterns .Chapter 5 surveys related work in this field, and the last chapter concludes the thesis and presents ideas for future work.

Chapter 2
Background

As mentioned in the introduction, aspect-oriented modelling (AOM) approaches apply advanced separation of concern techniques to modelling notations with the aim of modularizing crosscutting concerns. Many approaches have been developed in the past 12 years. These approaches show variability with regards to the notation they employ, the targeted level of abstraction as well as the temporal aspect of the software development process. The most prominent ones are reviewed in chapter 5.

This chapter first introduces Reusable Aspect Models, the AOM approach used in this thesis to illustrate instantiation cardinalities, and then explains the problems that a model user faces when a concern has to be applied multiple times within the same target model.

2.1 Introduction to RAM

Reusable Aspect Models (RAM) is an aspect-oriented modelling technique targeted at agile, concern-oriented software design modelling. RAM lies within the general framework of AOM with some distinct characteristics that are explained in the following subsections.

2.1.1 Support for Multi-View Modelling

RAM is a multi-view modelling approach. Every software design concern expressed in RAM within an aspect model has three different kind of views: a *structural view*, *message views* and *state views*.

The structural view is similar to a UML class diagram. It allows the model designer to express the structural properties of a software design concern, i.e., the classes with their attributes and operations and the associations among the classes relevant for that concern. In addition to providing the standard object-oriented concepts that UML offers, i.e., subclassing and overriding, RAM also offers support for aspect-oriented techniques such as class merging and method advising as explained later.

Message views describe the behaviour of the concern being modelled. There is one message view for each public operation defined by a class in the structural view. Each message view describes the sequencing of message interchanges that occur between instances of classes of the concern when providing the functionality offered by the public operation. RAM message views are based on the UML Sequence Diagram notation, extended with aspect-oriented concepts.

. Finally, state views allow a model designer to specify an *operation invocation protocol* for instance of classes of the structural view using a simplified version of the Protocol Modelling [25, 7] notation. The main idea of this additional behavioural view is to document the intended use of the classes that an aspect defines to the model user. Furthermore, the modelling tool can use the state views to check that the model user indeed obeyed the invocation protocol specified by the model designer. This is done by using a model checker that compares the sequence diagrams and state diagrams with each other.

2.1.2 Support for Reusability

RAM (Reusable Aspect Models), as the name suggests is about aspect models that are intended for reuse. This is a very significant feature in RAM. Each model has a well-defined *model interface* [4], in which the model designer specifies how the design can be (re)used within other models. Having an explicit model interface makes it possible to apply proper information hiding principles [34] by concealing internal design details from the rest of the application. In RAM, an aspect model has two kinds of interfaces: the *usage* and the *customization interface.*

- **Usage Interface**: *The usage interface specifies the design structure and behaviour that the model provides* to the rest of the application. In other words, the usage interface presents an abstraction of the functionality encapsulated within the model to the model user. It describes *how* the application can trigger the functionality provided by the model. It is comprised of all the *public* model elements, i.e., the structural and behavioural properties that the classes within the design model expose to the outside.
- **Customization Interface**: The *customization interface* of a RAM model specifies how a generic design model needs to be adapted to be used within a specific application. To increase reusability of models, a RAM modeller is encouraged to develop models that are as general as possible. As a result, many classes and methods of a RAM model are only partially defined. For classes, for example, it is possible to define them without constructors and to only define attributes relevant to the current design concern. Likewise, methods can be defined with empty or only partial behaviour specifications. The idea of the customization interface is to clearly highlight those model elements of the design that need to be completed/composed with application-specific model elements before a generic design can be used for a specific purpose. In RAM, these model elements are called *mandatory instantiation parameters*, and are highlighted visually by prefixing the model element name with a "|", and by exposing all model elements at the top right of the RAM model similar to UML template parameters.

In object-orientation and object-oriented modelling, structural and behavioural properties encapsulated inside a class can be reused (i.e., transferred to other classes) through inheritance. However, structure and behaviour that involves multiple classes can not be reused in a simple way. In OO, a designer is confined to the reuse of individual interface/classes, but needs to redefine how messages are passed between all these classes every time, even if the interactions he needs to create are always similar. This is not the case for reuse in RAM, since RAM allows the reuse of an aspect that usually contains multiple classes as well as the relationships and behaviour defined between instances of the classes as a whole.

This is why in object-orientation design patterns have become quite popular [14]. A design pattern encapsulates the knowledge of frequently reused component interaction approaches, and describes how to modify them to the design-specific context at hand. However, the proficient use of design patterns comes through significant experience [citation]. Sometimes, more than one design pattern can be skillfully combined to achieve a desired functionality. Combining multiple design patterns is a task that tends to get complex even for experienced programmers [citation]. Since RAM allows the reuse of classes as well the interactions between them, an entire design concern can be modularized and reused as a single unit. These concerns could be design patterns, also other interactions that recur often in software designs. Typical examples of such design concerns are complex data structures or network communication infrastructure.

2.1.3 Support for Explicit Model Dependencies

The ability to reuse aspects allows the creation of complex aspect dependency chains in RAM. Higher-level aspects, i.e., aspects that encapsulate complex structure and behaviour, depend on lower-level aspects, reusing the generic structure and behaviour that they define for their own purpose. This ability to reuse models simplifies the task of creating extremely complex models bringing it within the cognitive load of the modeller. Studies have shown that when an individual undertakes a mental task (e.g. attempting to *analyze* a model or answer *questions* about a model) that exceeds their *working memory capacity*, *errors* are likely to occur [37]. As a result, recent psychology research argues that mental tasks should always be designed such that they can be processed within the limit of ones *working memory capacity* [33].

When a higher-level aspect (HL) reuses a lower-level aspect (LL), the model designer of HL (who is the model user of LL) must specify detailed *instantiation directives* that map at least all the model elements from LL that were designated by the model designer of LL as mandatory instantiation parameters to model elements in HL. This is necessary in order to allow the weaver to generate a composed model as explained in the following subsection.

2.1.4 Tool Support

In the context of MDE and in particular of AOM, tool support for the creation, analysis and composition of models is essential. The RAM tool is called TouchRAM [3], and is developed following the current state-of-the-art techniques for MDE tools: a metamodel [22] for the abstract syntax of RAM has been defined, and the realization is done within the Eclipse Modelling Framework (EMF) [1].

Contrary to standard modelling tools, though, TouchRAM has an intuitive and streamlined user interface that enables agile software design modelling. In particular, it takes advantage of multi-touch input when running on a digital surface, such as a touch-enabled table, tablet or wall display. Touch-based input not only speeds up standard modelling activities (creating, moving, and connecting model elements), but also enables faster model reuse, because it speeds up browsing though model libraries and specifying model customizations.

The most important feature of TouchRAM in the context of this thesis is the model weaver, which is capable of composing aspect models, i.e. their structural, message and state views, according to the instantiation directives specified by the model user. The weaving algorithm is out of the scope of this thesis. The interested reader is referred to [3] for more details.

TouchRAM allows the model user to perform selective weaving, i.e. the user can choose to compose the structure and behaviour of a reused (lower-level) aspect with the structure and behaviour of the reusing (higher-level) aspect. Alternatively, the user can also directly instruct TouchRAM to weave the complete model, which recursively composes all reused aspects according to the instantiation directives to yield one big woven model. Selective weaving is useful, because it allows the model user to study the detailed interaction between the higher-level and lower-level concerns to ensure that the resulting structure and behaviour is consistent with his expectations. Complete weaving is useful for model analysis or code generation purpose.

2.1.5 RAM Example

This subsection illustrates the ideas introduced in the previous subsections by presenting the RAM design of the *Observer* design pattern aspect. The *Observer* design pattern [14] is a software design pattern in which an object, called the *subject*, maintains a list of dependents, called *observers*. The functionality provided by the pattern is to make sure that, whenever the subject's state changes, all observers are notified.

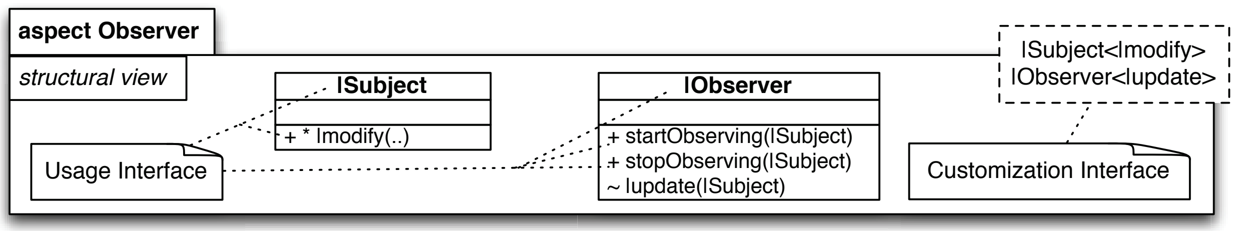


Figure 2–1: *Observer* RAM Model Interface (Customization and Usage)

As explained above, the *usage interface* of a RAM model is comprised of all the *public* model elements, i.e., the structural and behavioural properties, that the classes within the design model expose to the outside. The usage interface of the RAM design of the *Observer* design pattern is shown in Fig. 2–1. The structural view of the *Observer* RAM model specifies that there is a `|Subject` class that provides a public operation that modifies its state (`|modify`) that can be called by the rest of the application. In addition, the `|Observer` class provides two operations, namely `startObserving` and `stopObserving`, that allow the application to register/unregister an observer instance with a subject instance.

The *customization interface* of the *Observer* RAM model specifies which model elements inside the aspect are generic, i.e., which model elements need to be adapted to be used within a specific context. In our case these elements are the class `|Subject`, which needs to specify a `|modify` operation, and the class `|Observer` , which has to specify an `|update` operation. |modify has generic parameters and return type represented by "*" and ".." respectively.

Fig. 2–2 shows a possible internal design for the *Observer* aspect. The subject maintains an `ArrayList` of `Observers` referenced by `myList`. The *startObserving* and *stopObserving* message views are straightforward behavioural specifications that show how class instances collaborate to add/remove an observer instance to/from the array list of a subject.

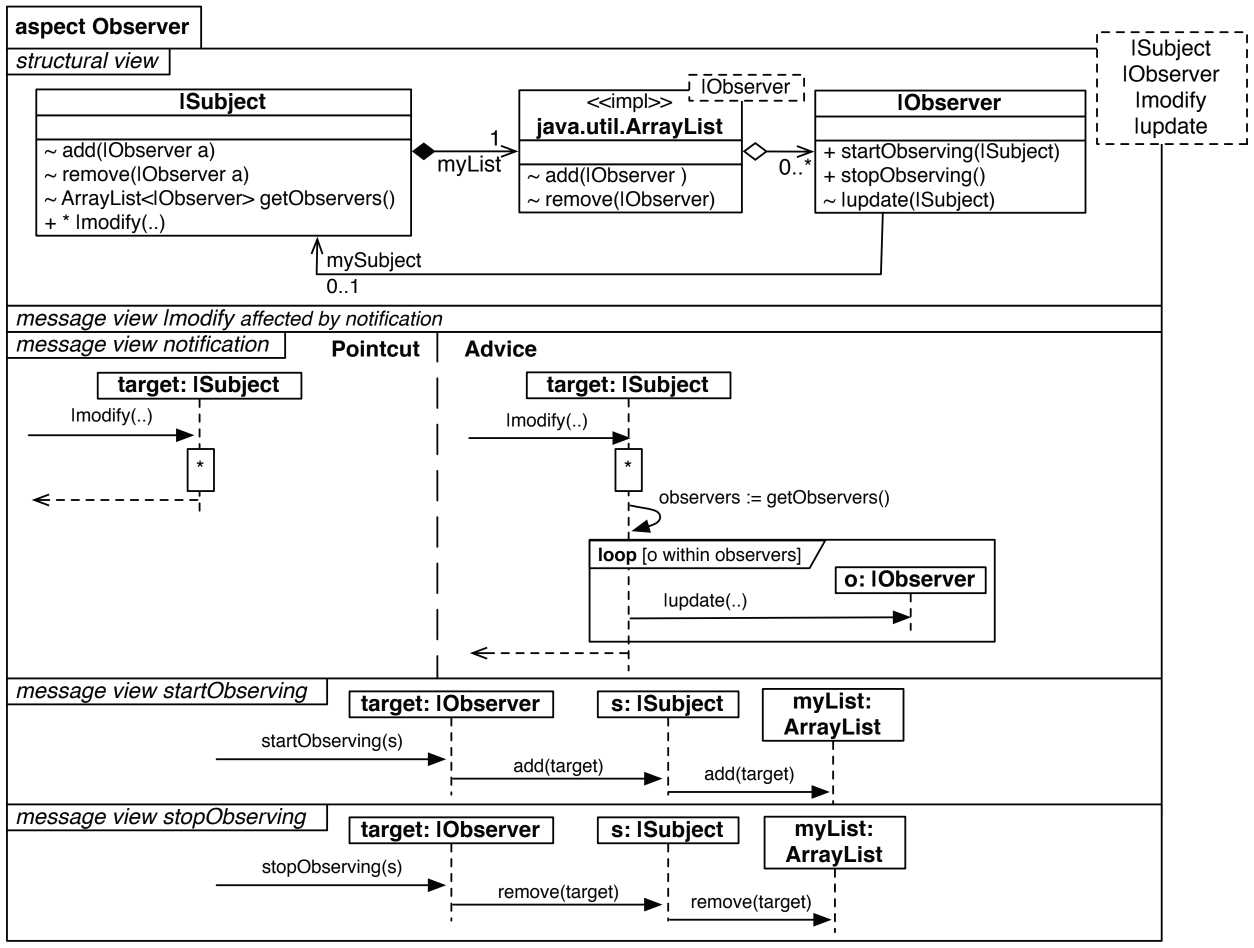


Figure 2–2: Internal Design of the *Observer* Aspect

The interesting behaviour encapsulated in the Observer aspect is specified in the *notification* message view. It uses aspect-oriented techniques to specify that the behaviour of `|modify` (represented by a white rectangle with a "*" in it) is augmented to invoke the `|update` method on all registered observer instances after the behaviour of `|modify` completed execution.

2.2 Instantiation Ambiguities

In the object-oriented world, where classes are the main modularization unit, generic designs are encapsulated within generic classes (also called template classes). The customization interface of a generic class clearly specifies what information the programmer who wishes to reuse a generic class needs to provide in order for the class to be usable. For instance, the Java class `ArrayList<E>` requires the user to specify the type of the elements that are to be stored within the array. If the user needs two different kinds of `ArrayLists` in his design, she can simply instantiate the generic class twice with different element types.

In RAM, when a modeller wants to reuse an already existing, generic RAM model within her current design, she must also use the customization interface to adapt the generic model to her specific design. This is done by providing *instantiation directives* that map every model element in the customization interface to a model element in current design model. If desired, *TouchRAM* [3], the modelling tool for the RAM approach, can compose the structure and behaviour of the two models using the instantiation directives to yield the complete software design model.

In some way, the reuse process in RAM is therefore similar to the one of generic classes in programming languages. However, in contrast to generic classes, RAM models typically encapsulate more than one design class, and the functionality provided by the aspect results from interacting instances of several different classes. Just like with classes, a modeller might want to reuse the functionality provided by an aspect once or multiple times in his design. However, since the functionality of the aspect is split over several classes, the user might need parts of the structure or functionality provided by an aspect model multiple times, but not all of it.

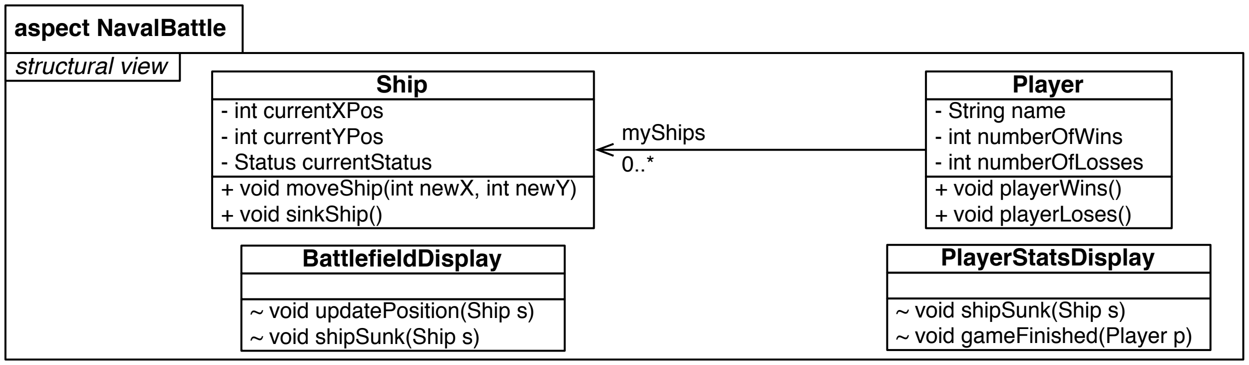


Figure 2–3: Simplified Naval Battle Base Model

Fig. 2–3 illustrates such a situation. The model shows parts of the design of a turn-based naval battle game, where players control ships that move around on a battlefield. Lets assume that there is a `BattlefieldDisplay` class that takes care of visualizing the battlefield on the screen, and there is also a `PlayerStatsDisplay` class that shows the list of all players together with statistics about their game performance, e.g., how many games they won or lost, and how many ships they sunk.

In such a design, the modeller may want to reuse the *Observer* concern shown in Fig. 2–1 to notify the display classes whenever the state of the ships or players change. An instantiation directive such as:

```
Subject → Ship
   modify → moveShip
Observer → BattlefieldDisplay
   update → shipMoved
```

would make sure that whenever a ship moves (because someone invokes the `moveShip` method on a ship), the `updatePosition` method of any `BattlefieldDisplay` instances that previously registered with the ship instance would be called.

In this situation, however, one could imagine more complex reuses of the *Observer* design that are not trivial to express. For instance, when a ship sinks (because someone invokes the `sinkShip` method on a ship), all registered `BattlefieldDisplay` instances and registered `PlayerStatsDisplay` instances should be notified by a call to their respective `shipSunk` methods. The modeller might be tempted to multi-map the *Observer* class, i.e., to write an instantiation directive such as:

```
Subject → Ship
   modify → sinkShip
Observer → BattlefieldDisplay, PlayerStatsDisplay
   update → shipSunk
```

to achieve the desired effect. Unfortunately, the implementation of the *Observer* design shown in Fig. 2–2 does not support such a multi-mapping, since the generic `java.util.ArrayList` class can only be parameterized with one type. To solve this problem, and in order to be able to reach both `BattlefieldDisplay` and `PlayerStatsDisplay` with a call to `shipSunk`, a superclass needs to be introduced and `shipSunk` must be transformed into a polymorphic call.

Without these changes, the only way to achieve the desired effect is to reuse *Observer* twice, i.e., to map `Observer` in one instantiation directive to `BattlefieldDisplay`, and to map `Observer` in the second instantiation directive to `PlayerStatsDisplay`. This will achieve the desired effect, but internally we then get two array lists, one containing `BattlefieldDisplay` instances, and the other one containing `PlayerStatsDisplay` instances. The `sinkShip` method is also advised twice, i.e., after updating the ship status, a first loop notifies all `BattlefieldDisplay` instances, and then a second loop notifies the `PlayerStatsDisplay` instances. Although this works, using two array lists (and looping through the observers in two separate loops) is not elegant, increases memory use and maybe even decreases performance.

In general, the need for fine-grained control over how many instances of a specific element defined in an aspect model should be created when the aspect model is reused multiple times within the same target model has been already highlighted in [28]. The authors define four so-called introduction policies. By default, new instances of the element are created each time the aspect model was reused (named *PerPointcut-Match* in [28]). It is also possible to specify that only a single instance is created regardless of how many times the aspect model is reused (referred to as *Global*). Finally, the authors also provide the possibility to specify new instances should be created only for a given matched set or tuple of model elements in the target model (*PerMatchedElement* or *PerMatchedRole*).

Chapter 3
Instantiation Cardinalities

This chapter introduces an extension to the customization interface of aspect-oriented models that addresses the issues introduced in section 2.2: it solves the reuse ambiguity that the model user currently experiences in RAM and similar AOM approaches. At the same time, this extension makes it possible for the model designer to have fine-grained control about how many instances of a specific model element defined in an aspect model are introduced into the target model. Section 3.1 first presents a general overview of the idea, and then provides the detailed definitions. Section 3.2 analyses the consequences of multi-mapping with respect to structural and message views. Finally, section 3.3 explains the integration of instantiation cardinalities with object-orientation.

3.1 Instantiation Cardinalities

3.1.1 Overview

We propose to augment the customization interface of a reusable unit by allowing the model designer to specify *instantiation cardinalities* for each model element. *The instantiation cardinality* of a model element *declares how many times, minimally and maximally, the model element can be mapped* to model elements of the target model within one instantiation, i.e, within one reuse. Visually, we suggest to show the instantiation cardinality in curly brackets to the right of the name of the model element using a syntax similar to what is done for UML multiplicities on association ends [31].

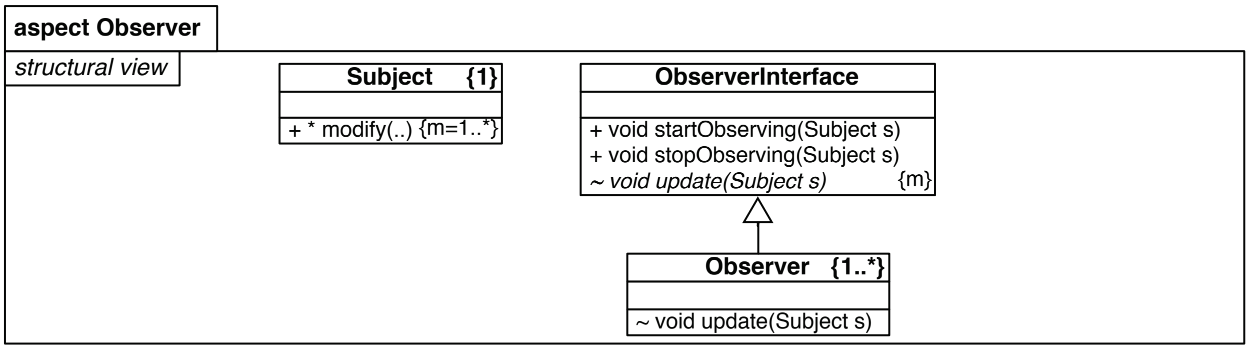


Figure 3–1: *Observer* RAM Model with Instantiation Cardinalities

Fig. 3–1 shows a design of the *Observer* pattern with instantiation cardinalities. It is meant to be used for one `Subject` and potentially several `Observers`, clearly specified by the instantiation cardinalities `{1}` for `Subject` and `{1..*}` for `Observer`. To achieve the problematic reuse mentioned in subsection 2.2 (to notify both `BattlefieldDisplay` and `PlayerStatsDisplay` when a ship is sunk), the modeller uses the following instantiation directive[1] :

```
Subject → Ship
   modify → sinkShip
ObserverInterface → DisplayInterface
   update → shipSunk
Observer<1> → BattlefieldDisplay
Observer<2> → PlayerStatsDisplay
```

With instantiation cardinalities, there is no need anymore for using the “ `|`” notation to designate mandatory instantiation parameters. Any model element that has a non-zero minimum instantiation cardinality must be mapped. To simplify the use of instantiation cardinalities

[1] The notation “`model_element<x>`” is used within an instantiation to refer to the xth instantiation of the corresponding model element.

for the model designer we also define a default cardinality for classes, i.e., `{0..1}`. For methods, the default cardinality is `{0}` since they are usually provided a definition when they are declared.[2] .

In order to express the situation where the number of instantiations of one model element must be equal to the number of instantiations of another model element, it is possible to define variables within the instantiation cardinality specification. For example, Fig. 3–1 states that there must be at least one `modify` method within the `Subject` class, but there can be more than one. However, for every `modify` method there should be a corresponding `update` method in the `ObserverInterface` class. By assigning the number of instantiations of the `Subject` class to the variable `m` (by specifying `{m=1..*}`), we are able to express this constraint on the `update` method of the `ObserverInterface` (by specifying its instantiation cardinality to be `{m}`).

In this case it is possible to write an instantiation directive such as:

```
Subject → Player
   modify<1> → playerWins
   modify<2> → playerLoses
ObserverInterface → DisplayInterface
   update<1> → gameCompleted
   update<2> → gameCompleted
Observer → BattlefieldDisplay
```

to specify that whenever `playerWins` *or* `playerLoses` is invoked on a `Player` instance, `gameCompleted` of the registered `PlayerStatsDisplay` instances is called.

[2] This makes sense not only because there are often classes and methods that do not need to be mapped, but also because that way classes and methods are not mandatory instantiation parameters by default. As a result, the model designer is forced to make a conscious decision when exposing model elements as mandatory instantiation parameters.

3.1.2 Instantiation Cardinality Syntax and Variables

Instantiation cardinalities can take the form of one integral number (or variable), e.g. {2}, or a range of numbers, e.g. {0..q}. For ranges (two numbers separated by two dots), the first number represents the minimum value and the second number represents the maximum value of the cardinality. For the second number, the character * can also be used which means that there is no predefined maximum value. When there is only one number (or variable) representing a cardinality, e.g. {2}, it can be equivalently represented as {2..2} in the two number format. In other words, the minimum value of {2} is 2 and the maximum value of {2} is also 2.

Whenever a variable like p appears in a cardinality, it is assumed that it is declared in the model somewhere. We have this rule for declaring variables in cardinalities:

Variable Declaration Rule: **"Each variable name that appears within an instantiation cardinality specification must be declared exactly once within the aspect model. The declaration must equate the variable to a cardinality range that may depend on other variables, if needed. Circular variable dependencies, however, are forbidden."**

The examples shown in Table 3–1 illustrate how this rule is applied in practice.

3.1.3 Single-Mapping and Multi-Mapping

As seen above, when a higher-level aspect (subsequently referred to as HL) reuses a lower-level aspect (LL), the model user must provide instantiation directives that consist of a set of mappings that relate a model element from LL to a model element in HL of the same

Example of Variable Declaration	**Comments**
{p=0..*} or {p=1..*} or {p=2..*}, etc.	Recommended way to declare a variable
{p=0..q} or {p=1..q} or {p=2..q}, etc.	Allowed. Make sure that q is declared exactly once and its declaration does not directly or indirectly depend on p. While we haven't experienced a situation where this kind of declaration might be useful, it might prove itself useful in the future.
{p=0} or {p=1} or {p=2} or {p=3}, etc.	Allowed. On declaring {p=3}, whenever any other element that has a cardinality that depends on p, e.g. {p}, it would be clear which element has an independent cardinality and which has a dependent cardinality. Also, in such a case, if a different value is needed, e.g. {p=4}, the value has to be changed at only one place.
{p=q..r}	Not recommended but allowed. Make sure that q and r are each declared exactly once and their declaration does not depend on p. In such a case it is also important that the maximum possible value of q is less than or equal to the minimum possible value of r. While we haven't experienced a situation where this kind of declaration might be useful, it might prove itself useful in the future.
{p=q..4} or {p=q..5} or {p=q..6}, etc.	Not recommended but allowed as long the declaration of q makes it clear that its maximum possible value is 4 (or 5 or 6, etc.) e.g. {q=0..3}. While we haven't experienced a situation where this kind of declaration might be useful, it might prove itself useful in the future.
{p=q}	Not allowed. Referring to one cardinality with two variable names would be confusing and not serve any useful purpose.

Table 3–1: Rules for Variable Declaration

Cardinality	**Type of Mappings Possible**
{0}, {1} and {0..1}	Single Mapping Only
{0..*}, {1..*}, {2..*}, ... {0..p}, {1..p}, {2..p}, ... {p},{p..q}	Single and Multi-Mapping

Table 3–2: Mappings and Cardinalities

type. The general rule for mappings is as follows:

Mapping Rule: **"Every model element (class or method) of a reused aspect can be mapped to a model element (of the same type) in the reusing aspect irrespective of its instantiation cardinality"**.

Mappings can be divided into two categories: *Single-Mapping* and *Multi-Mapping*.Single-mapping refers to the case where the model user provides only one mapping for a given model element in LL. Multi-mapping corresponds to the case where more than one mapping for a given model element in LL are provided. Table 3–2 provides the possible instantiation cardinalities, and what kind of mappings they allow.

When a single mapping is used for a model element, the woven model will contain that element *exactly* once. For multi-mappings, the woven view contains the element *at least* once. Since it is not possible to have the multiple model elements with the same name and type in one model, the following rule for multi-mappings is defined:

Multi-Mapping Rule: **" When a class or method has a multi-mapping cardinality, if it is mapped more than once, it must be renamed in each assignment".**

Syntactically, when a class or method gets mapped more than once, the suffix `<numeral>` is used to refer to each instance of the model element. The only place where the suffix is allowed to be used is within a mapping. The suffixes should not appear in structural or message views.

3.1.4 Class Cardinalities and Mappings

In this subsection we describe the meaning of mapping classes. When a model user of LL specifies that a class LC from LL is mapped to a class HC of HL, he has the possibility to *rename* the class, and also to *augment* the class.

Class renaming occurs when the class name HC differs from LC. Renaming of classes is important to adjust the name of the class from LL to accurately reflect its purpose in HL. For example, subsection 3.1.1 shows an instantiation directive where the Subject class of the Observer model is mapped to the Player class in NavalBattle.

Class augmentation occurs when the model user of LL, who is also the model designer of HL, specifies additional properties for HC in HL. When a class is augmented, it means that either one or more of the following four things happen to the class:

1. At least one new attribute or association is defined on HC in the structural view of HL. This is a case of *structural augmentation.*
2. At least one new method definition `mnew` is defined for HC in the structural view.It is a case of *structural augmentation.* Optionally, the modeller can also provide a message view for `mnew`, which results additionally in a behavioural augmentation.

Class Cardinality	**What can be done to the Class in a Higher-Level Aspect**
{0}	It can be mapped once (see mapping rule in subsection 3.1.3), and serves to change the name of a class to more accurately describe its purpose in HL. The class is *not allowed to be augmented.*
{1}	It *must* be mapped once (see rule in subsection 3.1.3). The class *must* be augmented.
{0..1}	It *can* be mapped once(see rule in subsection 3.1.3). It is optional to augment the class.
{1..*}	It must be mapped at least once, but can be multi-mapped. If mapped more than once, it must be renamed in each assignment. For each assignment, the class must be augmented.
{0..*}	In addition to the behaviour that is shown by {1..*}, {0..*} also allows the possibility of not mapping the class.
{p}	It must be mapped exactly p times. If p is greater than 1, it must be renamed in each assignment. For each assignment, the class must be augmented.

Table 3–3: How Class Cardinalities Affect the Model User

3. At least for one method `mold` declared in the structural view of LL for which there is no provided message view in LL, a message view is *defined* in HL. This is a case of *behavioural augmentation.*
4. A new message view is defined in HL that advises an existing method `mold` in LL. This is a case of *behavioural augmentation* as well.

Table 3–3 lists how instantiation cardinalities determine what the model user is allowed to do with the class in HL.

3.1.5 Method Cardinalities

In this subsection we describe the meaning of mapping methods. Just like for classes, when a model user of LL specifies that a method mlower (ml) of a class LC from LL is mapped to a method mhigher (mh) of class HC of HL, he has the possibility to *rename* the method, and also to *define* the method.

Method renaming occurs when the method name mh differs from ml. Renaming of methods is important to adjust the name of the method from LL to accurately reflect its purpose in HL. For example, subsection 3.1.1 shows an instantiation directive where the update method of the Observer class is mapped to the shipSunk method.

Defining a method means to provide a message view for mh in HL. This makes sense in the case where ml does not have a message view defined in LL.

Table 3–4 lists how method instantiation cardinalities determine what the model user is allowed to do with the method in HL.

3.1.6 Effects of Class Cardinalities on Cardinalities of the Contained Methods

The case where a class has a cardinality of $\{0\}$ deserves special mention, because it has an effect on the possible cardinalities of the methods it contains. In this case, only class renaming is possible. No new attributes, associations or methods can be added to the class. Furthermore, the message view for existing methods cannot be modified.

Hence, no method contained in LC should have a cardinality $\{1\}$, since $\{1\}$ implies that the message view *must* be modified. Also, multi-mapping methods is not allowed, since the rule of multi-mapping says that the message view for the methods must be modified. This means that in a class with cardinality $\{0\}$ all methods must have cardinality $\{0\}$. However, this should not be cause of concern because in our experience, the only situations where it might make sense to have a class cardinality $\{0\}$ is when it is an *implementation class*, i.e., an unmodifiable class provided by a third party or a language run-time.

Method Cardinality	**What can be done to the Method in a Higher-Level Aspect**
{0}	It *can* be mapped once (see mapping rule in subsection 3.1.3).The *existing message view* from the lower-level aspect *cannot be re-defined*.
{1}	It *must* be mapped once (see mapping rule in subsection 3.1.3). The (non-existing) *message view must be defined* in the higher-level aspect (unless it is an abstract method).
{0..1}	Not allowed. For every method, the designer must be aware whether it is intended to be defined in a higher-level aspect.
{1..*}	It must be mapped *at least once*, but can be multi-mapped if the model user chooses to do so. If mapped more than once, it *must be renamed in each mapping*. For each mapping, *the message view must be defined if it does not exist* (unless it is an abstract method).
{0..*}	In addition to the behaviour that is shown by {1..*}, {0..*} also allows the possibility of not mapping the method.
{p}	It must be mapped *exactly p times*. If p is greater than 1, it *must be renamed in each mapping*. For each assignment, *the message view must be defined if it does not exist*(unless it is an abstract method)

Table 3–4: How Method Cardinalities Affect the Model User

3.2 Consequences of Multi-Mapping

In the previous section, we introduced the concept of Instantiation Cardinalities, and precisely defined what kind of cardinalities are allowed and what they mean with respect to classes and methods. It must be stated here that these rules have simple explanations and hence, the designer does not need to actually memorize them. Also, as we shall see in the next chapter, when using instantiation cardinalities in real-world examples, they are straight-forward to use. In any case, the rules can be easily enforced by the modelling tool.

In this section, we look into the consequences of multi-mapping with respect to how many instances of each model element appears in the target model. In particular, we describe how the weaver needs to handle associations between classes, message views and method calls depending on the different cases that arise.

3.2.1 Associations Between Multi-Mapped Classes

In the presence of instantiation cardinalities, the weaver can easily determine how many instances of each model element from the reused aspect should be created in the target model. For model elements that are explicitly mapped, the number of instances is determined by the instantiation directive. Classes, operations and attributes that are not explicitly mapped are created once, except for classes that are contained in another class. In that case, the number of instances of the class is equal to the number of instances of the containing class.

Handling of relationships between classes, i.e., associations, aggregations, compositions and generalization-specialization, are more interesting. Assuming that class A and class B are related with relationship r, the different cases are handled as illustrated in Fig. 3–2 and described in the following list:

1. If the instantiation cardinality of class A is {0}, {0..1} or {1}, and the instantiation cardinality of B is {0}, {0..1} or {1}, then one single instance of r is created in the target model.
2. If the instantiation cardinality of class A is {q=1..*} or {q=0..*}, and the cardinality of B is {0}, {0..1} or {1}, then q instances of the relationship r are created in the target model.
3. If the instantiation cardinality of class A is {0}, {0..1} or {1}, and the cardinality of B is {q=1..*} or {q=0..*}, then q instances of the relationship r are created in the target model.
4. If the instantiation cardinality of class A is {q=1..*}, and the cardinality of B is {p=1..*}, then we are in a situation where the number of instances of A and B are completely independent. Hence, p*q instances of the relationship r are created in the target model.
5. The last and the most interesting case occurs if the instantiation cardinality of class A is {q=1..*} or {q=0..*}, and the cardinality of B is {q}, then we are in a situation where the number of instances of B is derived from the number of instances of A. In other words, every instance of A has its corresponding instance of B, and hence, 1 instance of the relationship r is created in the target model for each mapping of A. This gives a total of q instances.

3.2.2 Multi-mapping and Message Views

In the previous section, we looked at the effect of multi-mapping on classes and associations. In this section, we look at the effect of multi-mapping on message views. When a method calls another method, depending on the cardinalities of the methods as well as the

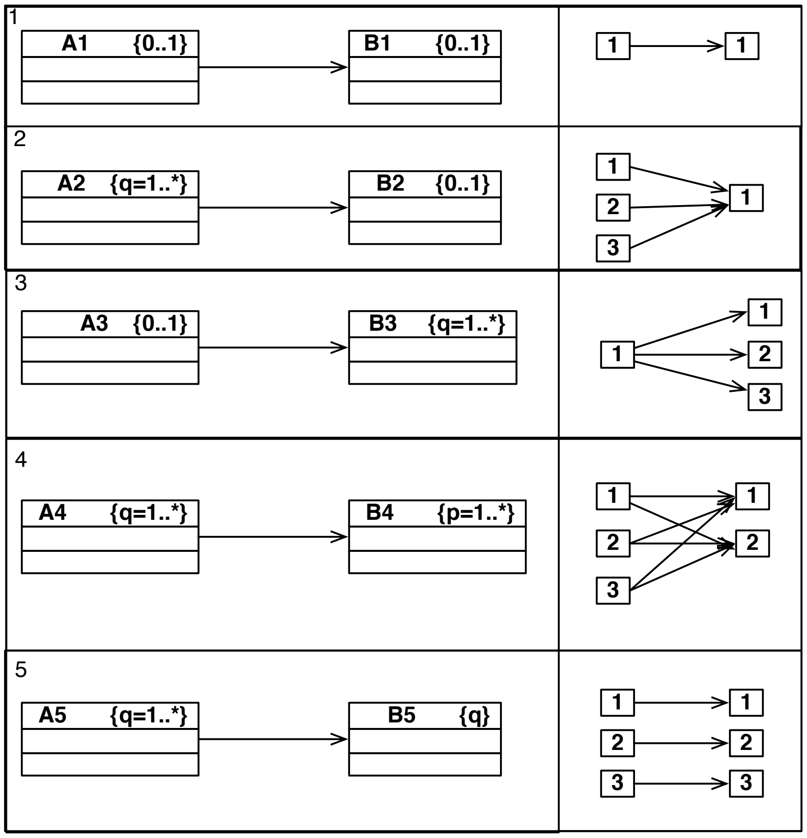


Figure 3–2: Rules for Associations

classes in which they are declared, there are multiple possibilities for the resulting woven message view. To handle the multiple cases that can arise, we define three rules and then show how they can be applied:

1. The number of message views of a method whose instantiation cardinality is p in the woven model is equal to p.
2. In the message view `m` of class `A`, any method call to a method `n` with cardinality q is replaced by a sequence of q method calls (`n`<1>, `n`<2>...`n`<q>) in the woven view, except if the cardinality of `m` or the cardinality of `A` is also q. If this is the case then only one method call to the corresponding instance of `n` appears in the woven message view.
3. In a message view `m` of class `A`, any method call to a method n that is defined within a class `B` that has cardinality q is replaced by a sequence of q method calls (`B`<1>.`n`, `B`<2>.n.. `B`<q>.`n`) in the woven view, except if the cardinality of `m` or the cardinality of `A` is also q. If this is the case, then only one method call to the corresponding instance of `B.n` appears in the woven message view.

Fig. 3–4 shows the application of the above rules in some cases. The application of the rules is straightforward in the different cases. The above three rules can also be applied to `nestedOp` and `sameOp` in Fig. 3–3 without any special considerations.

3.3 Integrating Instantiation Cardinalities with Object-Orientation

In the previous section we discussed how the weaver handles associations and message views with instantiation cardinalities. In this section we explore how subclassing and inherited methods integrate with cardinalities.

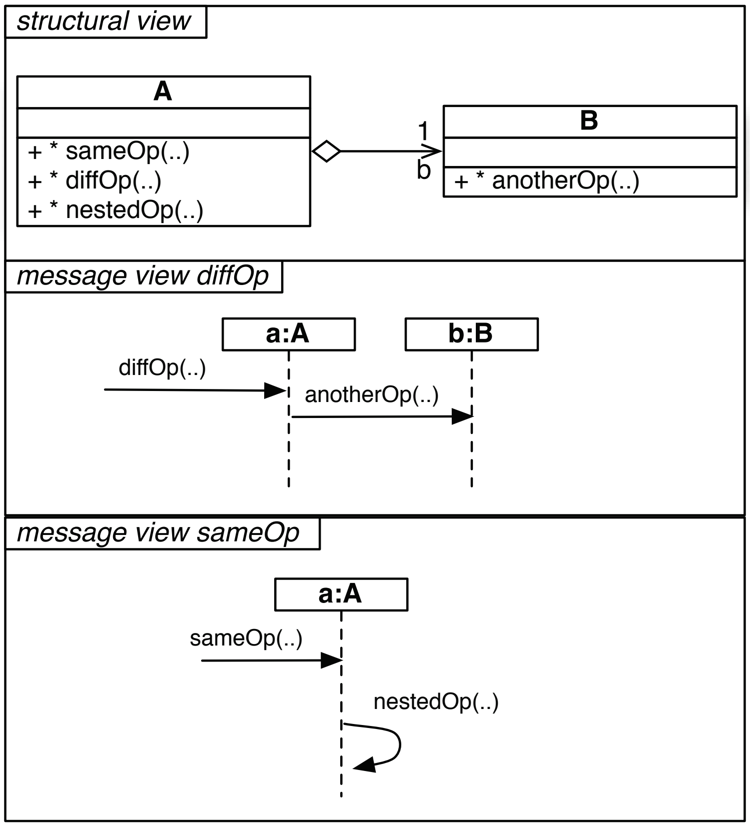


Figure 3–3: Multi-mapping nested methods

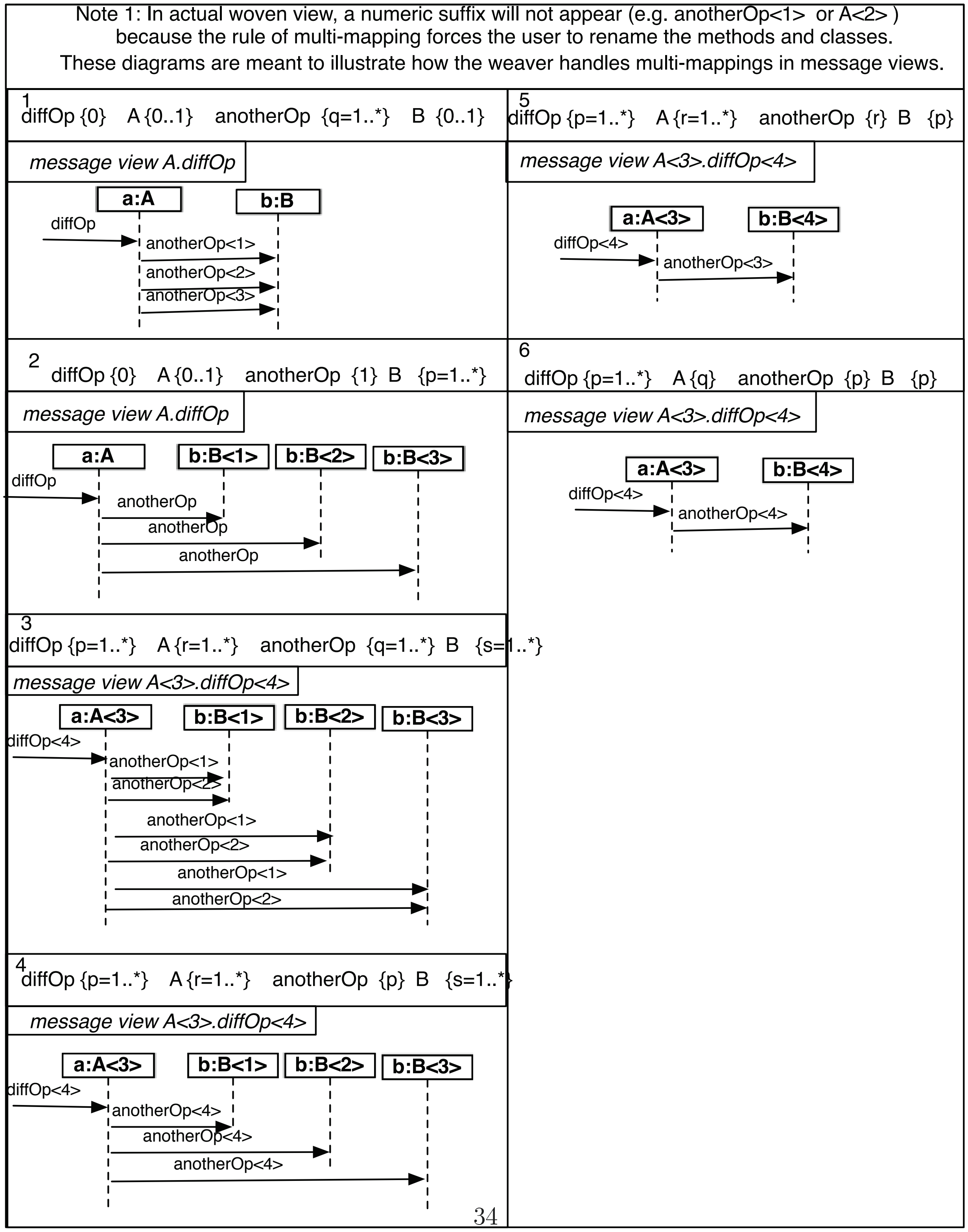

Note 1: In actual woven view, a numeric suffix will not appear (e.g. anotherOp<1> or A<2>)
because the rule of multi-mapping forces the user to rename the methods and classes.
These diagrams are meant to illustrate how the weaver handles multi-mappings in message views.
1
diffOp {0} A {0..1} anotherOp {q=1..*} B {0..1}
message view A.diffOp
a:A
b:B
diffOp
anotherOp<1>
anotherOp<2>
anotherOp<3>
2
diffOp {0} A {0..1} anotherOp {1} B {p=1..*}
message view A.diffOp
a:A
b:B<1>
b:B<2>
b:B<3>
diffOp
anotherOp
anotherOp
anotherOp
3
diffOp {p=1..*} A {r=1..*} anotherOp {q=1..*} B {s=1..*}
message view A<3>.diffOp<4>
a:A<3>
b:B<1>
b:B<2>
b:B<3>
diffOp<4>
anotherOp<1>
anotherOp<2>
anotherOp<1>
anotherOp<2>
anotherOp<1>
anotherOp<2>
4
diffOp {p=1..*} A {r=1..*} anotherOp {p} B {s=1..*}
message view A<3>.diffOp<4>
a:A<3>
b:B<1>
b:B<2>
b:B<3>
diffOp<4>
anotherOp<4>
anotherOp<4>
anotherOp<4>
5
diffOp {p=1..*} A {r=1..*} anotherOp {r} B {p}
message view A<3>.diffOp<4>
a:A<3>
b:B<4>
diffOp<4>
anotherOp<3>
6
diffOp {p=1..*} A {q} anotherOp {p} B {p}
message view A<3>.diffOp<4>
a:A<3>
b:B<4>
diffOp<4>
anotherOp<4>

Figure 3–4: Example Weaving of Message Views with Multi-Mapping

3.3.1 Subclassing

Generalization-Specialization, or inheritance as it is often called in object-orientation, defines a “is a” relationship between classes. It therefore makes sense that the instantiation cardinality of a superclass A has an effect on the cardinality of its subclasses. We define the following rule:

Rule: Multi-Mapping of Subclasses: “**When a subclass is multi-mapped, the cardinality of the subclass represents the number of times it can be mapped with respect to its superclass**.”

Fig. 3–5 illustrates two cases. In case I, class B *must* be mapped `p` times for *each mapping of class A*. If there is a situation like case II, class C must be mapped q times for each mapping of class B. Class B in turn must be mapped `p` times for each mapping of class A. The reason behind this is that while actually using instantiation cardinalities we found that it allows for the most natural usage.

3.3.2 Inherited Methods

When a method is inherited from a superclass, there can be multiple cases. In Fig. 3–6case 1, `B.operation()` has a cardinality `{1}` and it is inherited from `A.operation()` which has a cardinality`{1..*}`. This means that `B.operation()`*must* be mapped exactly once for *each* mapping of `A.operation()`. To make the example more concrete, consider the mappings:

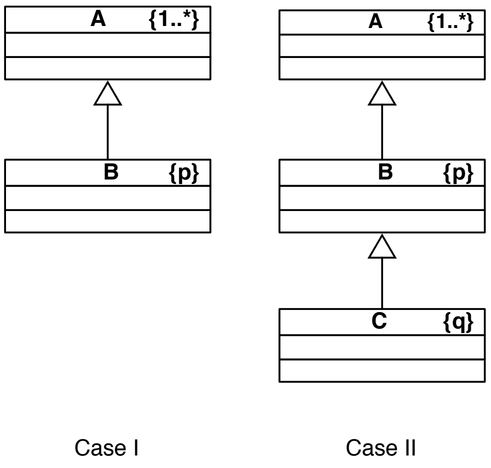


Figure 3–5: Multi-mapping Subclasses

```
A —→ Media
   operation<1> → playMedia
   operation<2> → stopMedia
B → Song
   operation<1> → playSong
   operation<2> → stopSong
```

Since A is the superclass, the multi-mappings of `operation` serve the additional purpose of determining the *number of times* `A.operation()` is actually mapped. The Song class that is mapped to B will become a subclass of Media in the woven view as B is a subclass of A. Consequently, both the methods `playMedia` and `stopMedia` will be inherited by the Song class. Since the designer has specified a cardinality $\{1\}$ for `B.operation()`, it means that `B.operation()` must be mapped and a message view must be provided for each `B.operation()`. The cardinality of $\{1\}$ is somewhat similar to the idea of method overriding in OO languages. While it serves a similar purpose, `playSong` does not *override* the message

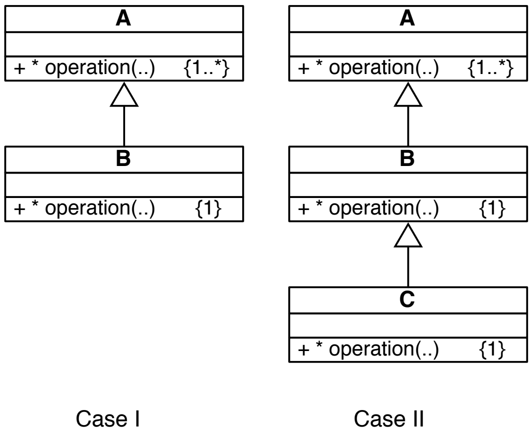


Figure 3–6: Multi-mapping Inherited Methods

view of `playMedia`. Instead, `playMedia` now simply calls `playSong` using *Automated Call Forwarding* described in the following section(3.3.3).

If `B.operation()` has the cardinality {0}, it is not allowed to be defined in a higher-level aspect. However, the user can still *rename* it to a different method name.

Fig. 3–6 Case II has an additional subclass C. `C.operation()` has the cardinality {1}. Hence, like `B.operation()`, `C.operation()` must be mapped the same number of times as `A.operation()` is mapped. The cardinality for inherited methods Fig. works slightly differently from the cardinality of subclasses as shown in Fig. 3–5. In fact, an inherited method can have a cardinality of {0} or {1} only, which leads to the following rule:

Rule: Multi-Mapping for Inherited Methods:**"If a method is to be multi-mapped, this is always indicated in the class where the method is declared. Inherited methods are not allowed to be multi-mapped."**

In Fig. 3–6, if one or more of the classes A, B or C had a cardinality greater than 1, it would not affect the above rule in any manner. The rules for multi-mapping of subclasses and inherited methods can mutually co-exist without any special considerations.

3.3.3 Polymorphism using Automated Call Forwarding

The rules of object-orientation dictate that in a subclass, the name for a method that overrides a method defined in a superclass must remain the same. In AOM, where sub- and superclasses may happen to be defined in separate aspect models, this constraint hinders true separation of concerns. It requires a designer to chose the method names in one concern based on name definitions of another concern.

To remedy this situation, we allow overridden methods in subclasses to optionally be mapped to methods that do not necessarily have the same name as the superclass method (or any of the method names in the sibling classes). The only constraint is that the method's parameter number and types must match.

If the model user specifies such a mapping, then the model weaver automatically inserts an additional method with the name defined in the superclass, that directly forwards all calls to the mapped method. As a result, it is possible in the aspect model that defines the superclass to make a call that polymorphically dispatches to a differently named method of a subclass defined in a different aspect model.

This feature is not only convenient, it becomes essential when a high-level aspect reuses several generic aspect models or existing implementation classes. For instance, in the *Navalbattle* example discussed earlier, if the player statistics are kept on a remote web server, then one might want to reuse the *Observer* aspect model defined in Fig. 3–1 as follows:

```
Subject → Ship
   modify → sinkShip
Observer<1> → BattlefieldDisplay
   update → refreshWindow
Observer<2> → PlayerStatsDisplay
   update → sendStatsToServer
```

Chapter 4
Design Patterns Revisited

Chapter 3 introduced instantiation cardinalities by means of the *Observer* behavioural design pattern example. This chapter presents an in-depth case study in which instantiation cardinalities are applied to create detailed reusable aspect models of six additional design patterns [14]. First, the structural design patterns *Composite* [14] and *Decorator* are shown in section 4.1. To complement the *Observer* design pattern, section 4.2 presents the design of the behavioural design patterns *Template Method* and *Command.* Finally, the design models of the creational design patterns *Builder* and *Abstract Factory* [14] are shown in section 4.3.

The intent of this extensive case study is to highlight:

- how instantiation cardinalities provide a precise specification of the customization interface of a model that unambiguously determines how the model user is supposed to map the generic model elements from the reused model to a target model,
- how instantiation cardinalities integrate with object-oriented concepts, and finally,
- how the weaver can exploit instantiation cardinalities to determine the exact number of times each model element from the reusable model needs to appear in the target model.

4.1 Structural Design Patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures using inheritance and object composition [14]. In this section, we discuss two structural patterns: *Composite* and *Decorator*.

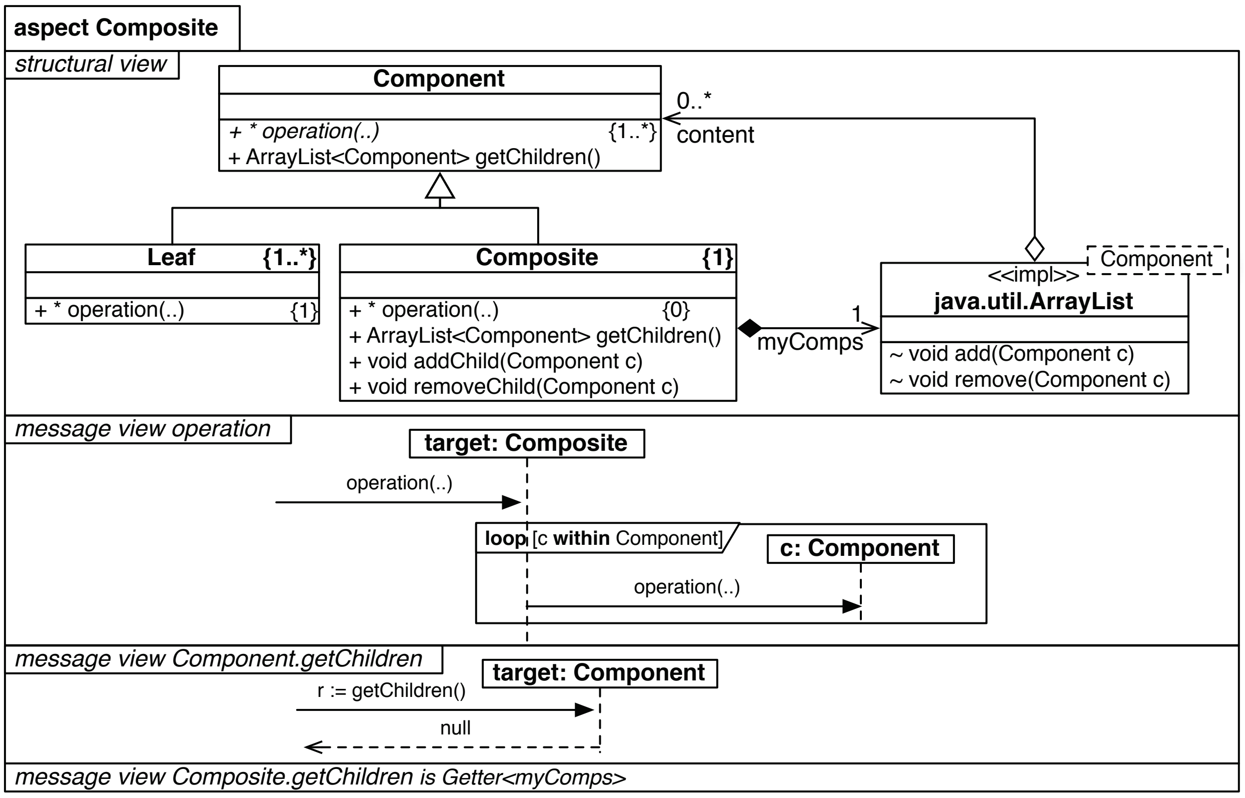


Figure 4–1: *Composite* RAM Model

4.1.1 Composite

The *Composite* design pattern is a well-known structural design pattern that allows individual objects and collections of objects to be treated uniformly [14]. Operations are defined in a common interface, and invoking such an operation on a collection of objects results in applying the operation to each element in the collection.

Fig. 4–1 shows that the RAM structural view of the *Composite* pattern is similar to the classic OO UML diagram found in [14]. Instantiation cardinalities have been added to each class that clearly show how the classes are intended to be mapped. While there need to be `{1..*}` `Leaf` classes, there has to be exactly `{1}` `Composite` class. The common `Component`

```
Component → Media
   operation<1> → playMedia
   operation<2> → stopMedia
Leaf<1> → Song
   operation<1> → playSong
   operation<2> → stopSong
Leaf<2> → Video
   operation<1> → playVideo
   operation<2> → stopVideo
Composite → PlayList
   operation<1> → playPlayList
   operation<2> → stopPlayList
```

Figure 4–2: Example Instantiation of *Composite*

interface is optional to map, but at least one `operation` must be specified `{1..*}`. Mapping it multiple times allows the model user to expose multiple leaf operations. In the message view for `Composite.operation`, we define the behaviour that loops through all the children and calls `operation` on each child. Note that, we need to and are allowed to define only one message view for this method, irrespective of the number of times `operation` is going to be mapped in a higher-level aspect.

For example, suppose a higher-level aspect *Jukebox* reuses the *Composite* aspect as follows:

The mappings in the *Jukebox* aspect also nicely illustrates the advantage of automated call forwarding. The designer of *Jukebox* is not bound to use identical method names for the common operations defined in the different leaf classes. This allows for a great amount of flexibility while modelling. For instance, a user might have started creating the *Jukebox* aspect with the `Song` and `Video` classes together with the `playSong` and `playVideo` operations. Only later, when designing `PlayList`, she realizes that the *Composite* pattern is useful in this context. Thanks to automated call forwarding, she can simply map the methods to `operation` defined in *Composite* without the need to modify any existing method

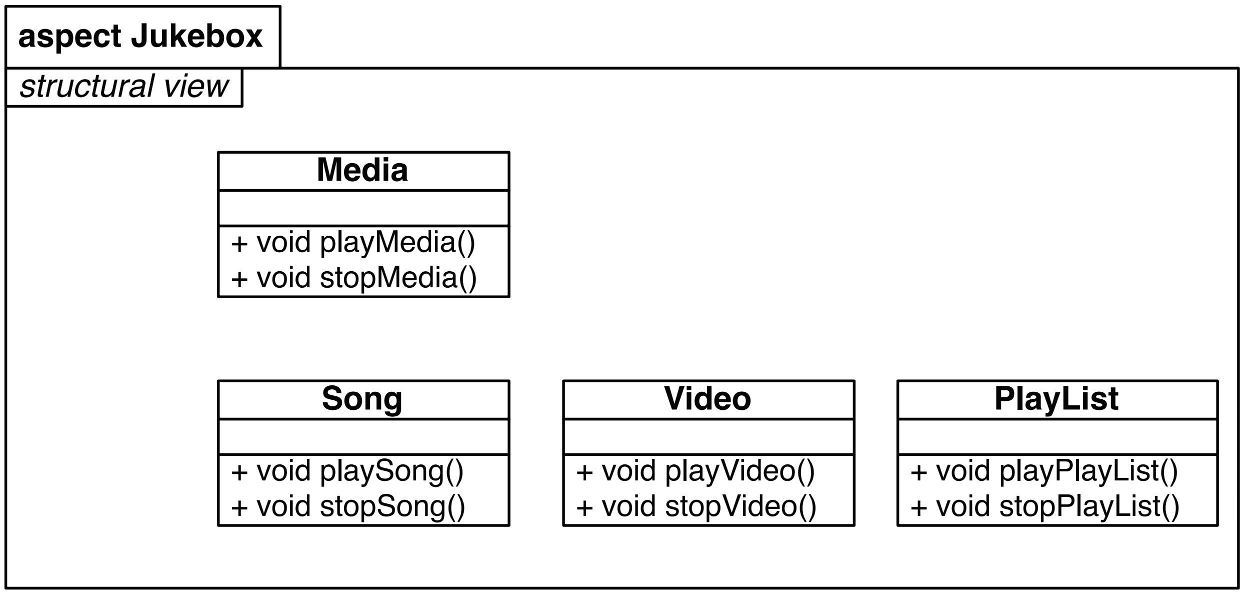


Figure 4–3: Example Use of *Composite*

names. When the two aspects are woven together by the *TouchRAM* tool, the weaver creates `playMedia` methods in `Song` and `Video` that forward calls to `playSong` and `playVideo`, respectively. That way, the polymorphism exploited in the `Composite.operation` message view is maintained.

4.1.2 Decorator

Decorator is a structural design pattern that makes it possible to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. Let us consider an example (taken from [2]). Suppose we have a pizza store that sells multiple types of pizza. One approach could be to have a `Pizza` class and a separate subclass for each type of pizza. This approach works fine when there are a limited varieties of pizza to offer. If we want to now extend the varieties of pizza by letting the customer create custom pizza, i.e., the customer decides on a base and then selects the

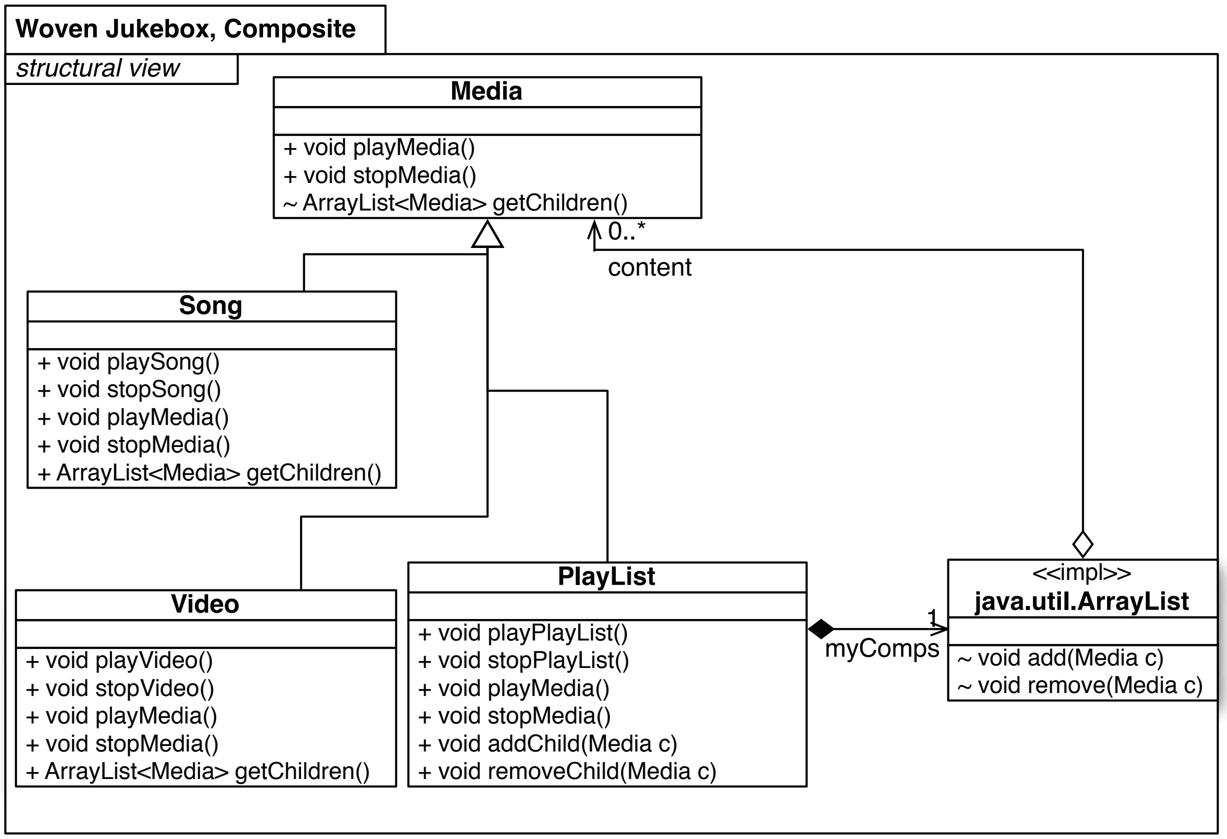


Figure 4–4: Woven Model using *Composite*

toppings, the number of possible pizzas would grow exponentially with each topping. The limitations of subclassing now start to become clear. If there is change in the price of one component, the price of the component would need to be changed individually in all the classes. The *Decorator* pattern works elegantly in this kind of situation. It allows the selection of the *ConcreteComponent* (pizza base) and the different *ConcreteDecorator* (toppings) at runtime.

Fig. 4–5 shows the *Decorator* RAM model with instantiation cardinalities. The mappings in the higher-level aspect *PizzaStore* that reuse the *Decorator* aspect are as follows :

It should be noted that mapping `operation` in the `Topping` class is optional and a user is not allowed to redefine its message view since the cardinality of `operation` is $\{0\}$. The message view `assignComponent` is an advice that affects the behaviour of constructors of a `Decorator`. Constructors are represented by the keyword `create` in RAM and are treated exactly like methods with respect to cardinalities. The cardinality of $\{1..*\}$ implies that multiple constructors can be defined for the class that maps to *Decorator.* The advice for `assignComponent` states that whenever `create` is called, after the definition for `create` as defined in the higher-level aspect has been executed, the `component` reference in the *Decorator* class should be assigned to the `Component` object `c` that must be passed to `create` as a parameter.

Fig. 4–8 shows the woven model that combines *PizzaStore* and *Decorator.* The message views of *Topping* are shown. The message views of other methods utilize a straightforward application of Automated Call Forwarding and have not been shown for space reasons.

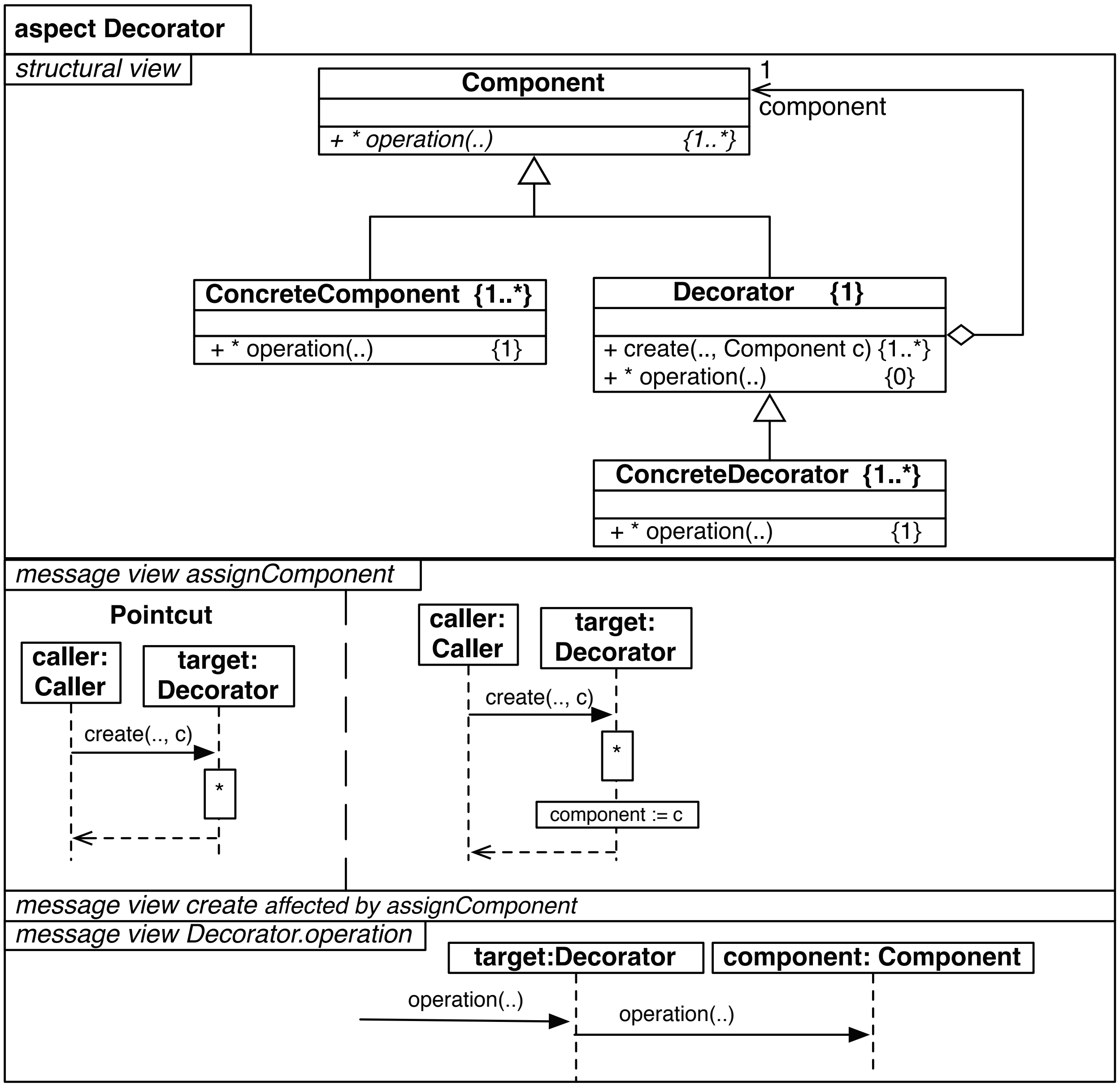


Figure 4–5: *Decorator* RAM model

```
Component→Pizza
   operation<1>→getPizzaDescription
   operation<2>→getPizzaCost
ConcreteComponent<1>→ThinCrust
   operation<1>→getThinCrustDescription
   operation<2>→getThinCrustCost
ConcreteComponent<2>→StuffedCrust
   operation<1>→getStuffedCrustDescription
   operation<2>→getStuffedCrustCost
Decorator→Topping
   operation<1>→getToppingDescription
ConcreteDecorator<1>→Mushrooms
   operation<1>→getMushroomsDescription
   operation<2>→getMushroomsCost
ConcreteDecorator<2>→Onions
   operation<1>→getOnionsDescription
   operation<2>→getOnionsCost
ConcreteDecorator<3>→Chicken
   operation<1>→getChickenDescription
   operation<2>→getChickenCost
ConcreteDecorator<4>→Pineapple
   operation<1>→getPineappleDescription
   operation<2>→getPineappleCost
```

Figure 4–6: Example Instantiation of *Decorator*

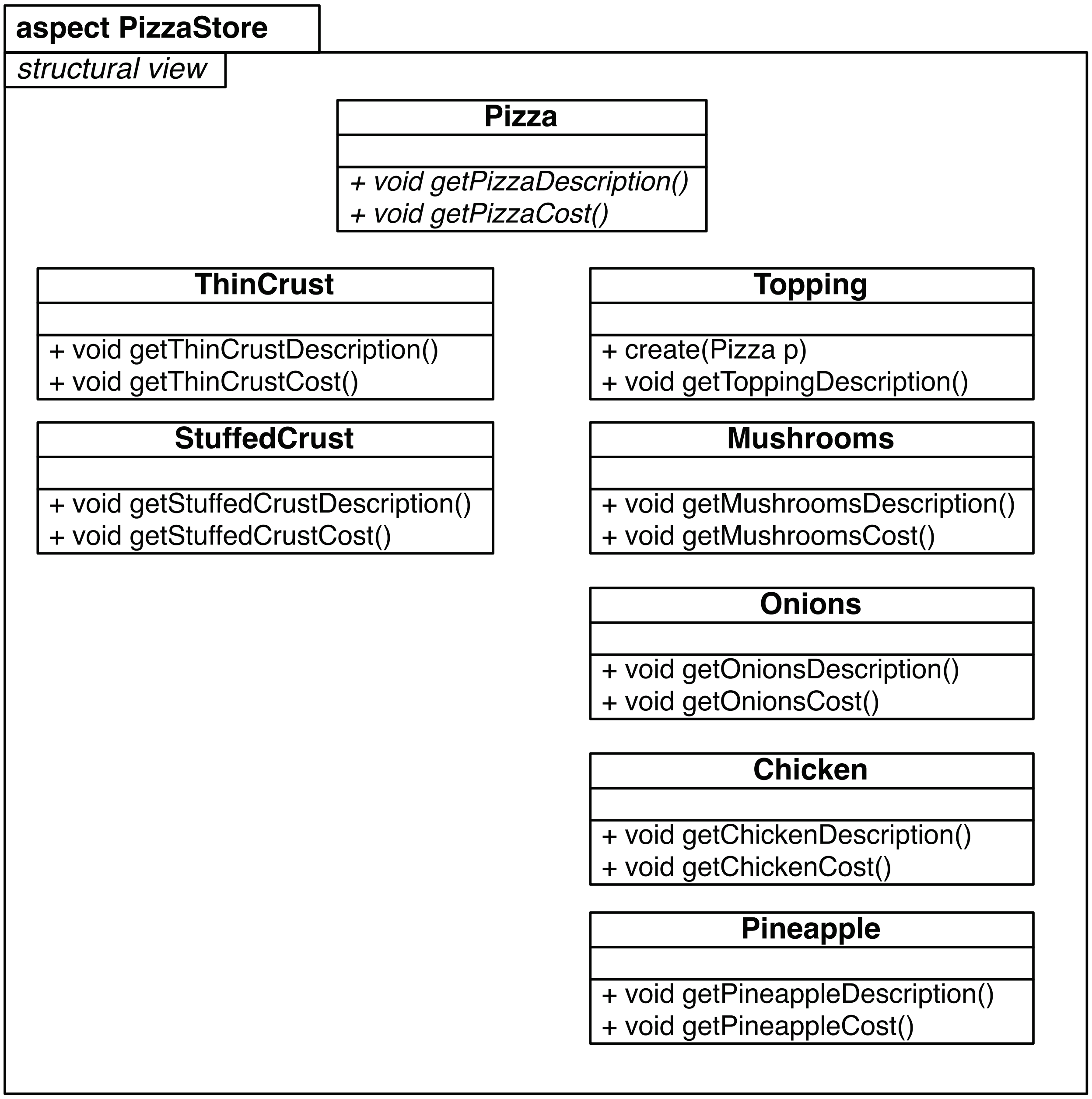

aspect PizzaStore
structural view
Pizza
+ void getPizzaDescription()
+ void getPizzaCost()
ThinCrust
+ void getThinCrustDescription()
+ void getThinCrustCost()
StuffedCrust
+ void getStuffedCrustDescription()
+ void getStuffedCrustCost()
Topping
+ create(Pizza p)
+ void getToppingDescription()
Mushrooms
+ void getMushroomsDescription()
+ void getMushroomsCost()
Onions
+ void getOnionsDescription()
+ void getOnionsCost()
Chicken
+ void getChickenDescription()
+ void getChickenCost()
Pineapple
+ void getPineappleDescription()
+ void getPineappleCost()

Figure 4–7: Example Use of *Decorator*

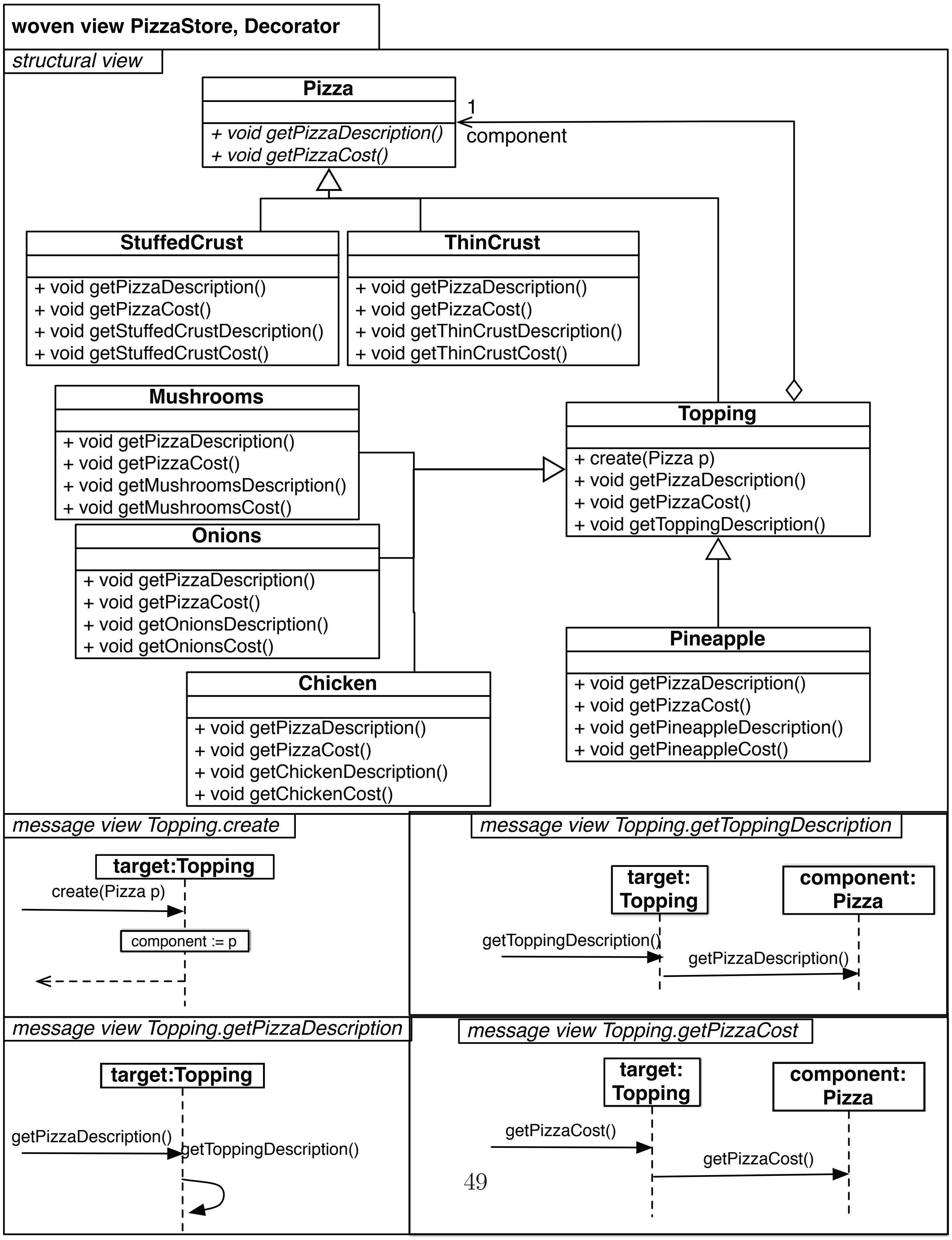

woven view PizzaStore, Decorator
structural view
Pizza
+ void getPizzaDescription()
+ void getPizzaCost()
1
component
StuffedCrust
+ void getPizzaDescription()
+ void getPizzaCost()
+ void getStuffedCrustDescription()
+ void getStuffedCrustCost()
ThinCrust
+ void getPizzaDescription()
+ void getPizzaCost()
+ void getThinCrustDescription()
+ void getThinCrustCost()
Mushrooms
+ void getPizzaDescription()
+ void getPizzaCost()
+ void getMushroomsDescription()
+ void getMushroomsCost()
Topping
+ create(Pizza p)
+ void getPizzaDescription()
+ void getPizzaCost()
+ void getToppingDescription()
Onions
+ void getPizzaDescription()
+ void getPizzaCost()
+ void getOnionsDescription()
+ void getOnionsCost()
Pineapple
+ void getPizzaDescription()
+ void getPizzaCost()
+ void getPineappleDescription()
+ void getPineappleCost()
Chicken
+ void getPizzaDescription()
+ void getPizzaCost()
+ void getChickenDescription()
+ void getChickenCost()
message view Topping.create
target:Topping
create(Pizza p)
component := p
message view Topping.getToppingDescription
target: Topping
component: Pizza
getToppingDescription()
getPizzaDescription()
message view Topping.getPizzaDescription
target:Topping
getPizzaDescription()
getToppingDescription()
message view Topping.getPizzaCost
target: Topping
component: Pizza
getPizzaCost()
getPizzaCost()

Figure 4–8: Woven model using *Decorator*

4.2 Behavioural Design Patterns

Behavioural patterns describe the patterns of communication between objects and classes with respect to algorithms and assignment of responsibilities [14]. Observer is a common behavioural design pattern and has been discussed in section 3.1. In this section we discuss two more behavioural patterns: *Template Method* and *Command*.

4.2.1 Template Method

Template Method is a behavioural design pattern. The intent is to define the skeleton of an algorithm in an operation, deferring the details of some of its steps to subclasses. *Template Method* lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. [14] Let us consider an example (taken from [2]). Suppose, there is a class `Sandwich` and it declares a method `makeSandwich`. `makeSandwich` in turn calls multiple methods like `addMeat`, `addCheese` and `addCondiments`. The user wants to create a subclass for each of the different types of `Sandwich`. The method `makeSandwich` follows the exact same steps in each of the sandwiches. However, the type of meat (or no meat), type of cheese (or no cheese) and the type of condiments depend on the actual sandwich. Hence, the definitions of the methods `addMeat`, `addCheese` and `addCondiments` are delegated to the individual subclasses. However, the `makeSandwich` method is defined in the superclass and it should not be redefined in the subclasses. In this example, `makeSandwich` would be the template method.

Fig. 4–9 shows the RAM model for the *Template Method* pattern. It also exhibits the power of instantiation cardinalities. The fact that the `templateMethod` must not be overridden is shown by using the cardinality of $\{0\}$ in `Subclass` for `templateMethod`. In an OO language it would be typically represented by using the final keyword for `templateMethod` in `SuperClass`. Also, the fact that `primitiveOperation` *must* be defined in `Subclass` is shown by the cardinality of $\{1\}$. Since, `primitiveOperation` is abstract, it would have been true anyway, but this is a case where the cardinality clearly shows the user exactly what needs to be done. In the message view, a simple note is used to indicate that all methods mapped to `primitiveOperation` must be called inside the `templateMethod`. However, it is also possible that the method mapped to `templateMethod` does other things as well. In our example, `makeSandwich` might also have an *if* statement to check if `addMeat` should be executed. As of now, the RAM metamodel does not support the definition of such constraints for message views. This is work for future consideration (section 6.1)

Here are the mappings for the aspect *SandwichMaker* (Fig . 4–11):

4.2.2 Command

Command is a well-known behavioural design pattern. It encapsulates a request of execution of a single or multiple actions in an object. In other words, a command object reifies one or multiple method executions. When desired, the command object can be asked to execute itself, which results in calling the appropriate method(s) on a receiver object.

To better understand the *Command* pattern, let us consider an example. Suppose we have a smart room environment which automatically switches on the light whenever someone enters the room and switches off the light when the person leaves the room. The fact that a person has entered the room is encapsulated in a `Command` object that exists independently of the `Light` object. This object has a generic `execute` method which simply calls the `action` method in the `Receiver` (`Light` in this case). It can be seen how the Command pattern allows the designer to separate the action to be performed and the object that actually performs it. This separation is very useful since it can be used to create command queues,

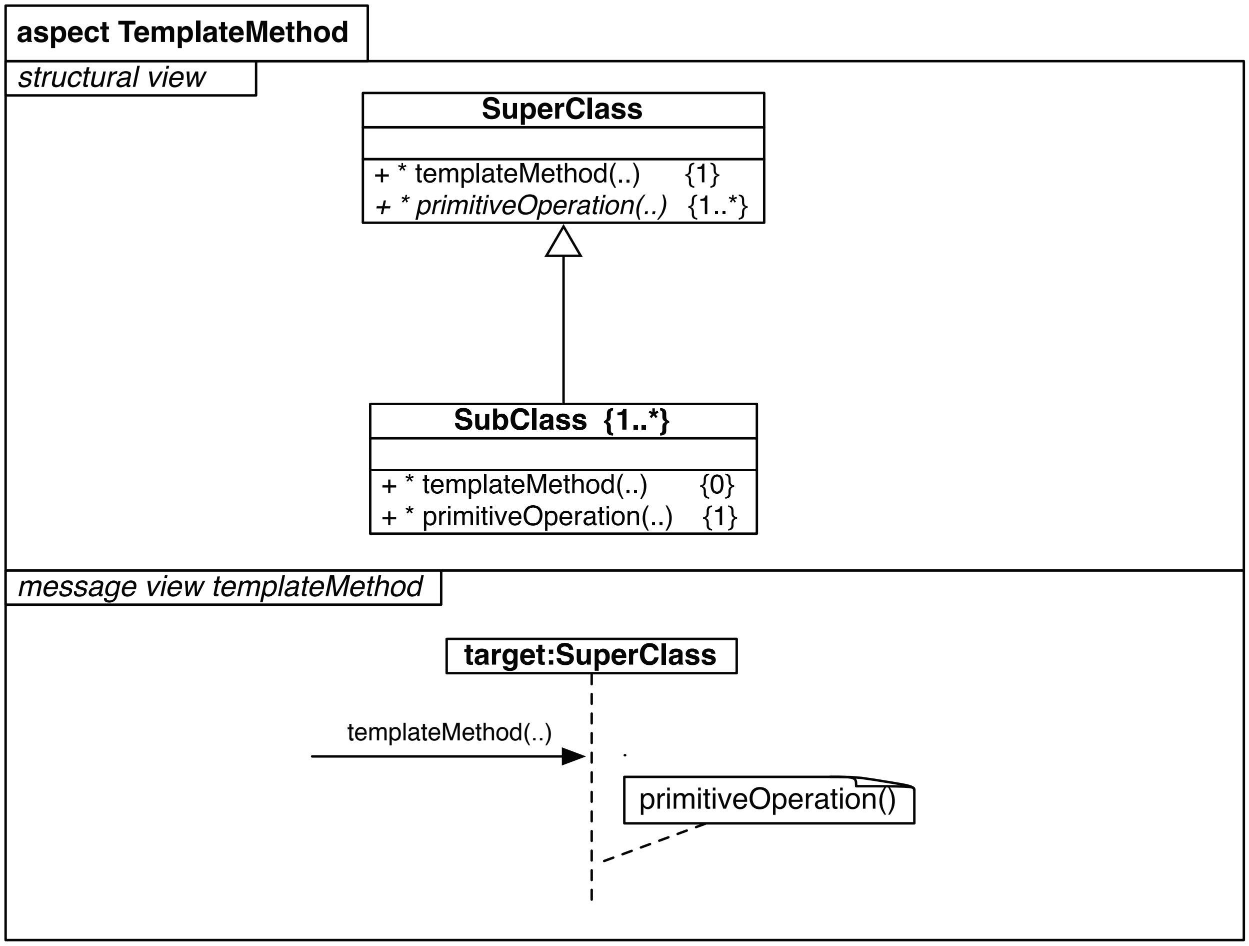

aspect TemplateMethod
structural view
SuperClass
+ * templateMethod(..) {1}
+ * primitiveOperation(..) {1..*}
SubClass {1..*}
+ * templateMethod(..) {0}
+ * primitiveOperation(..) {1}
message view templateMethod
target:SuperClass
templateMethod(..)
primitiveOperation()

Figure 4–9: *TemplateMethod* RAM Model

```
SuperClass → Sandwich
   templateMethod → makeSandwich
   primitiveOperation<1> → addMeat
   primitiveOperation<2> → addCheese
   primitiveOperation<3> → addDressingSauce
SubClass<1> → BeefSandwich
   templateMethod → makeBeefSandwich
   primitiveOperation<1> → addBeef
   primitiveOperation<2> → addCheddarCheese
   primitiveOperation<3> → addMayonnaise
SubClass<2> → TurkeySandwich
   templateMethod → makeTurkeySandwich
   primitiveOperation<1> → addTurkey
   primitiveOperation<2> → addSwissCheese
   primitiveOperation<3> → addHoneyMustard
```

Figure 4–10: Example Instantiation of *TemplateMethod*

command managers that facilitate undo and redo, to pass commands over a network, etc. The discussion of all possible uses of the command pattern are beyond the scope of this work. We show here how instantiation cardinalities allow the model designer to clearly communicate the intent of the design pattern to the model user.

Command with Single Receiver

he RAM model of *Command* with a single `Receiver` is shown in Fig. 4–14. The message view for `execute` calls the `action` method in the `Receiver` object associated with the `Command` object. Since, the designer wants the `execute` method of `ConcreteCommand` to not be modified in a higher-level aspect it is assigned a cardinality {0}. Another point of interest is that `ConcreteCommand` has cardinality {p=1..*}, and action has cardinality {p}. This ensures that there exists a corresponding action for each `Command` class in the `Receiver`.

Fig. 4–15 shows a higher-level aspect *SmartLight* that reuses the aspect *CommandSingleReceiver*. The mappings reinforce the utility of RAM, since the user simply needs to think

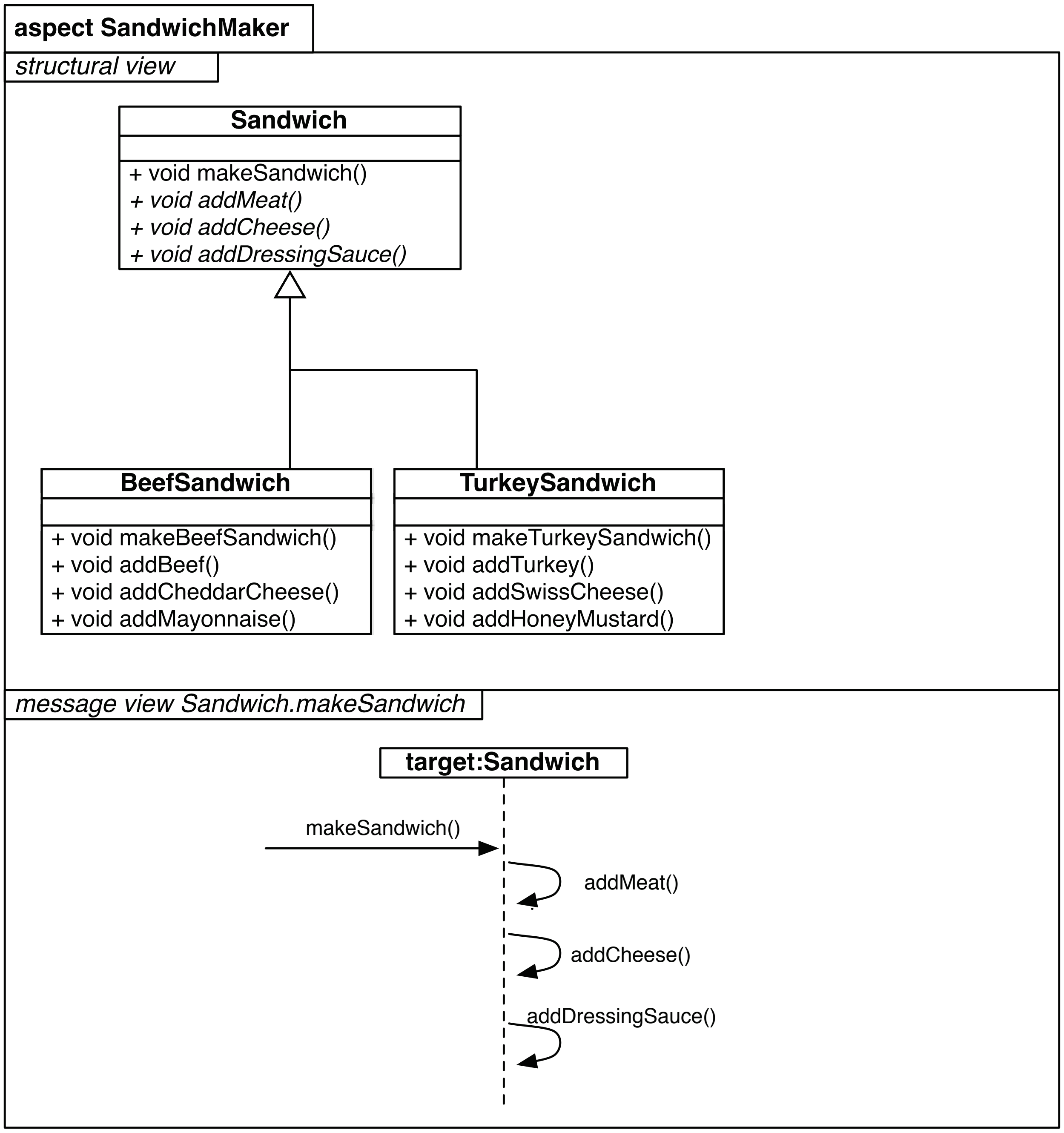


Figure 4–11: Example use of *TemplateMethod*

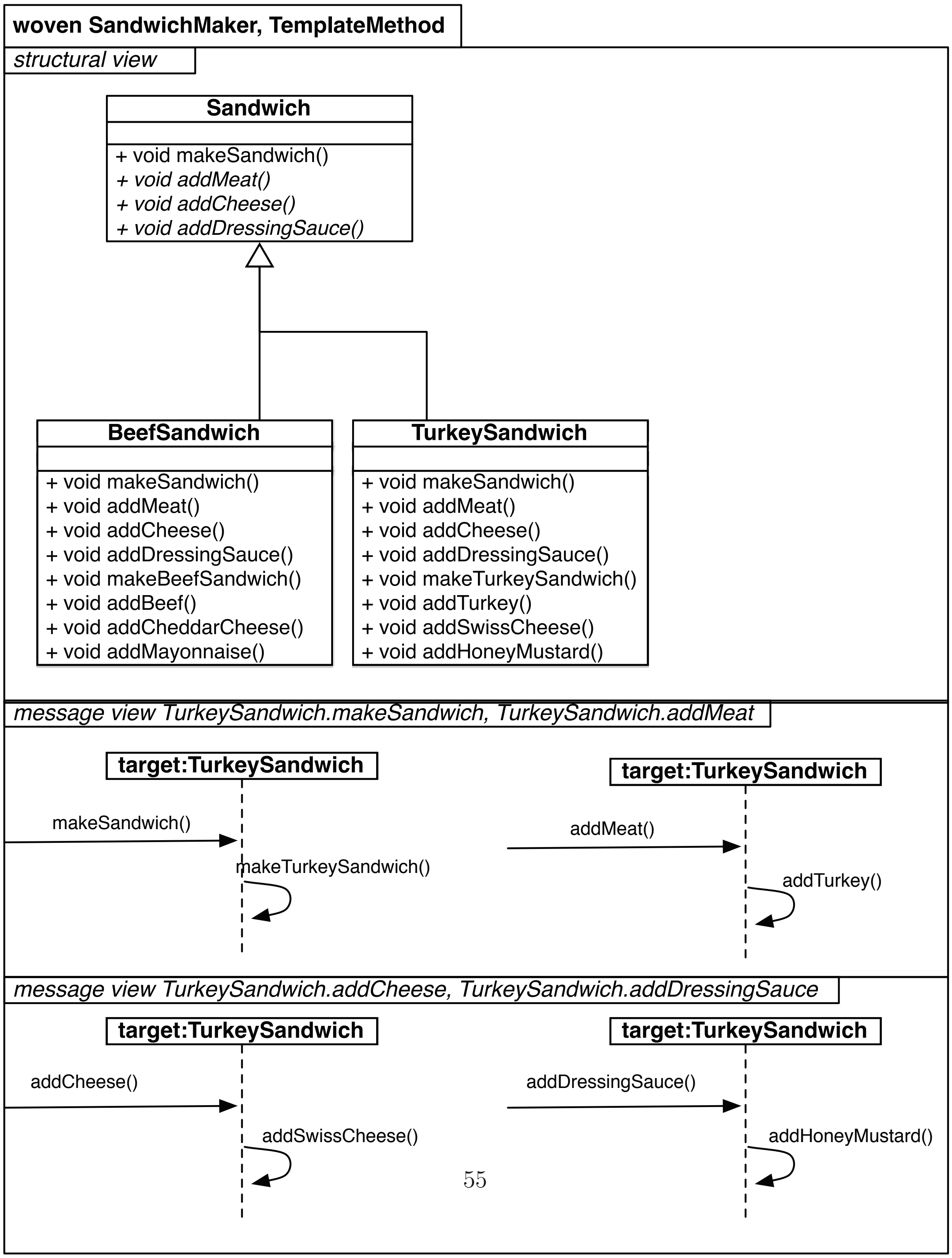


Figure 4–12: Woven Model using *TemplateMethod*

```
Receiver → Light
   action<1> → switchOn
   action<2> → switchOff
ConcreteCommand<1> → EnterRoom
ConcreteCommand<2> → LeaveRoom
```

Figure 4–13: Example Instantiation of *Command* with Single Receiver

in terms of the higher-level aspect only. The user is free to assign different method names for different actions, e.g., `switchOn` and `switchOff`.

The mappings for the aspect *SmartLight* are shown in Fig. 4–13.

By applying the rules for calling multi-mapped methods, the weaver modifies each execute message view to call the corresponding action in the receiver, i.e., `EnterRoom.execute` calls `switchOn`, whereas `ExitRoom.execute` calls `switchOff.`

Command with Multiple Receivers

When there are two or more receivers for each command, the Command pattern can be extended as shown in Fig. 4–18. Each mapping of `ConcreteCommand` contains reference to its own list which stores the multiple receiver objects. The receiver objects can belong to different classes which must be mapped to `ConcreteReceiver`. Each mapping of `ConcreteReceiver` must contain a corresponding action for each mapping of `ConcreteCommand.` An example higher-level aspect reusing the aspect CommandMultipleReceivers is shown in Fig. 4–19. The corresponding mappings are shown in Fig. 4–17.

In Fig. 4–20,the message view for `Light.onEntering`, `Blinds.onEntering` and `Radio.onEntering` show the application of Automated Call Forwarding.

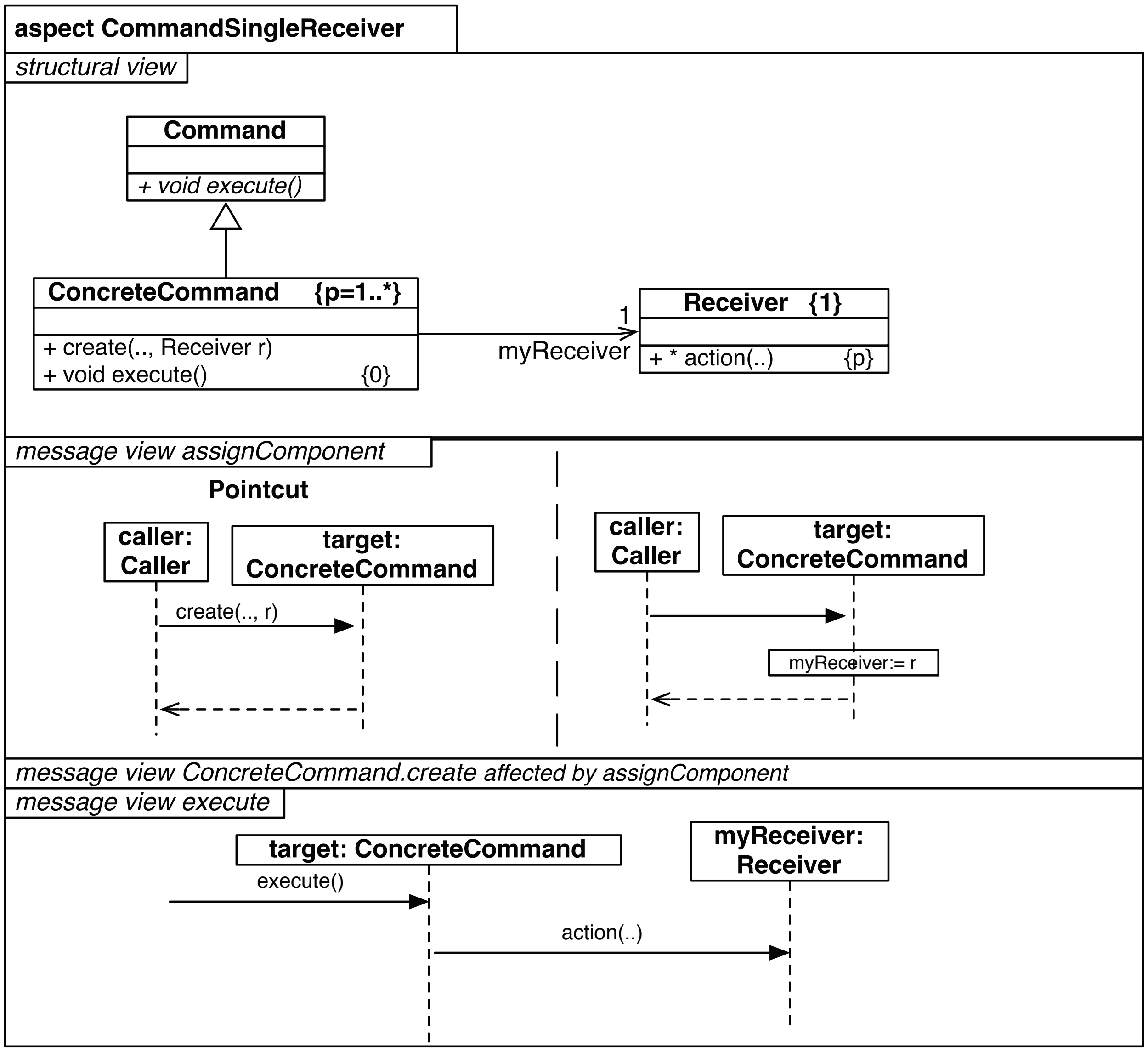


Figure 4–14: *Command* RAM Model with a Single Receiver

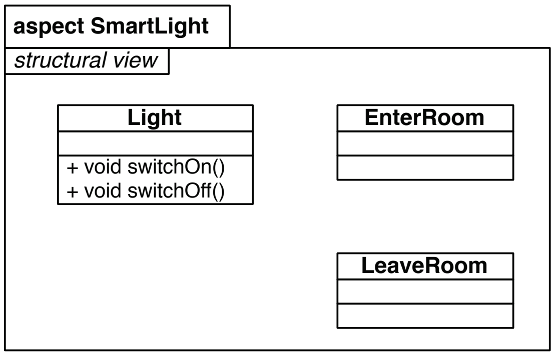


Figure 4–15: Example Use of *Command* with Single Receiver

4.3 Creational Design Patterns

Creational design patterns abstract the creation of class instances by making a system independent of creation, composition and representation of its objects [14]. In this section we discuss two creational patterns: *Builder* and *Abstract Factory*.

4.3.1 Builder

Builder is a creational design pattern that allows the creation of complex objects that must be created in the same order or using a specific algorithm [14]. Let us consider an example that illustrates the *Builder* pattern (taken from [2]). Suppose, we want to create a `Robot` object. Here, `Robot` is the `Product` that need to be built. This `Robot` object has multiple parts like Head, Arms, Legs, etc. In the *Builder* pattern, the details of how to make these `Robot` constituents is delegated to a `Builder` class, e.g. `OldStyleRobotBuilder`. Since there can be multiple types of `RobotBuilders`, the different methods that are used to make the `Robot` are declared in a `RobotBuilder` interface and the concrete builders implement this interface. Also, the rules for actual creation of the `Robot` are in a `Director` class, e.g. `RobotDirector`. `RobotDirector` calls the methods defined in the `RobotBuilder` that are needed to build the robot. This decouples the method definition of making individual parts

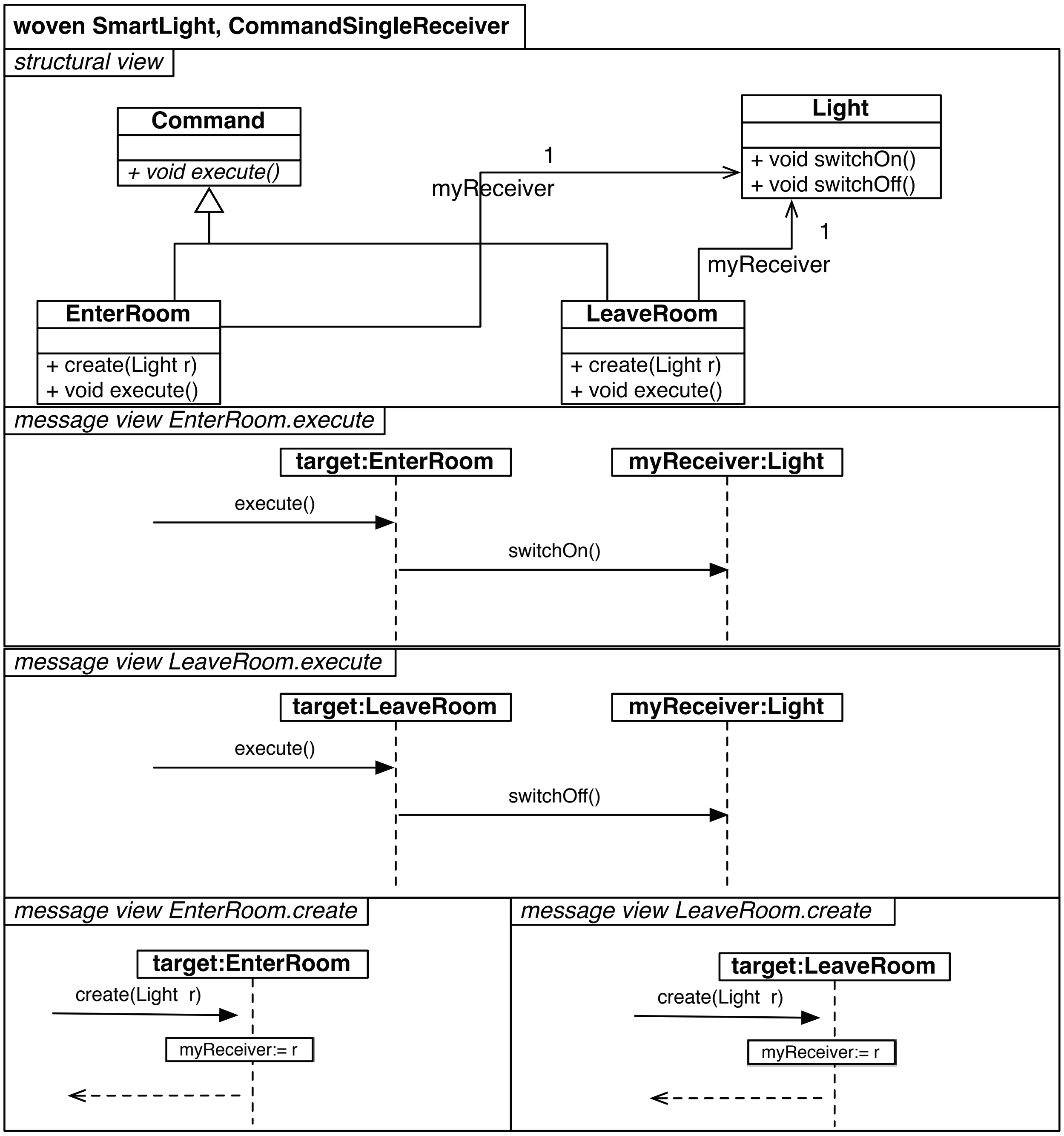


Figure 4–16: Woven Model using *Command* with Single Receiver

```
Receiver → RoomContents
   action<1> → onEntering
   action<2> → onLeaving
ConcreteReceiver<1> → Light
   action<1> → switchOn
   action<2> → switchOff
ConcreteReceiver<2> → Blinds
   action<1> → lowerBlinds
   action<2> → raiseBlinds
ConcreteReceiver<3> → Radio
   action<1> → turnOn
   action<2> → turnOff
```

Figure 4–17: Example Instantiation of *Command* with Multiple Receivers

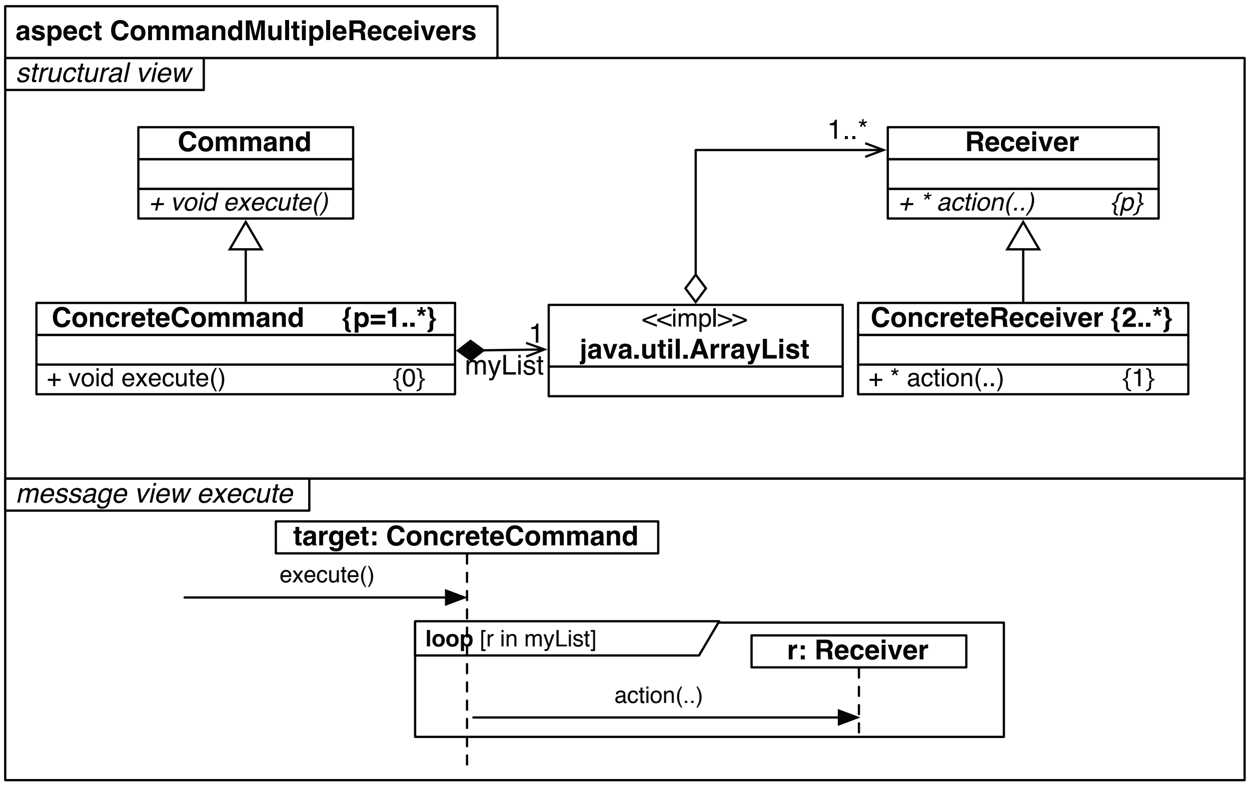


Figure 4–18: *Command* RAM Model with Multiple Receivers

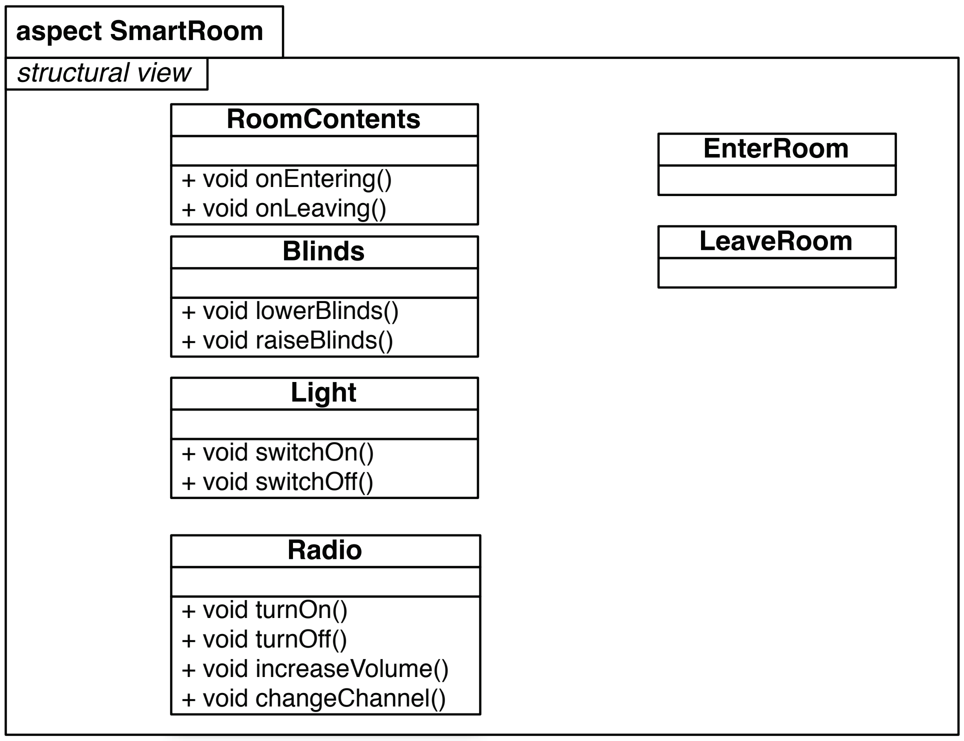


Figure 4–19: Example use of *Command* with Multiple Receivers

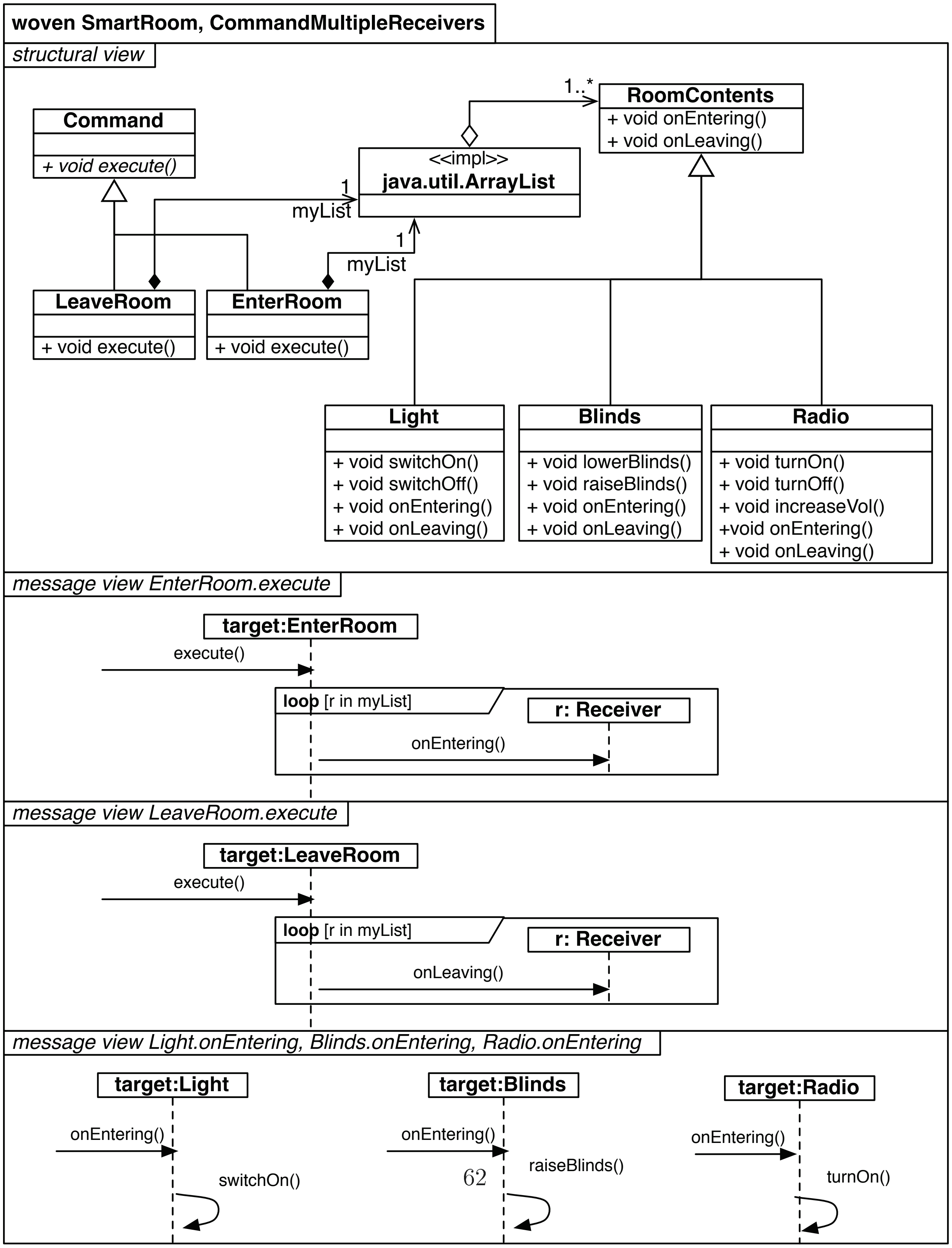


Figure 4–20: Woven Model using *Command* with Multiple Receivers

from the director that calls those methods in a specific order. This allows the constituent parts of a product to be dynamically passed to the director that combines these parts to create the final product.

Fig. 4–21 shows the RAM model for the *Builder* design pattern.

Fig. 4–23 shows an example use of the *Builder* pattern. The corresponding mappings are shown in Fig. 4–22

4.3.2 Abstract Factory

Abstract Factory is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes [14].

The pattern can be best described by an example. Consider two vehicle factories: `Toyota` and `Honda`. Each factory produces three types of vehicles: `Car`, `Motorcycle` and `Truck`. `Toyota` produces exactly one vehicle of each type: `ToyotaCar`, `ToyotaMotorcycle` and `ToyotaTruck`. The same is true for `Honda`.

Abstract Factory allows a modeller to instantiate a factory when the application is initialized (`VehicleFactory fact = new Toyota()`). Subsequently, whenever a specific type of vehicle is needed, it can be instantiated (`Car newcar = fact.createCar()`) without having to know if the application uses `ToyotaCars` or `HondaCars`. This decouples the creation of the products from the specific factory that actually produces them.

Figs. 4–25 and 4–26 highlight the difference between a standard *Abstract Factory* UML diagram (taken from [14]) and the *Abstract Factory* RAM model. The advantages of using instantiation cardinalities are obvious:

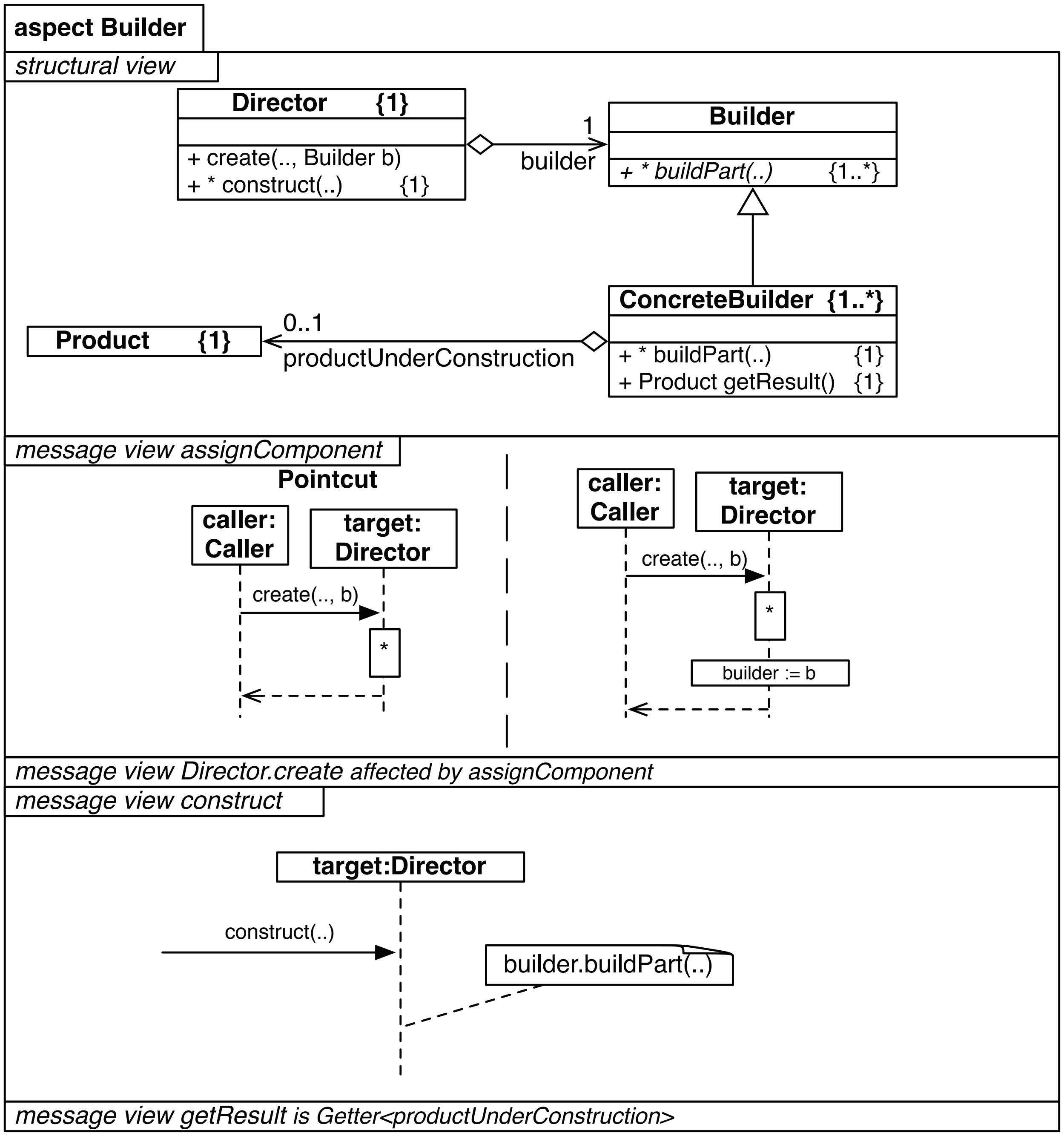


Figure 4–21: *Builder* RAM Model

```
Builder → RobotBuilder
   buildPart<1> → makeArms
   buildPart<2> → makeLegs
   buildPart<3> → makeHead
ConcreteBuilder<1> → OldStyleRobotBuilder
   buildPart<1> → makeBlowTorchArms
   buildPart<2> → makeRollerSkatesLegs
   buildPart<3> → makeTinHead
   getResult → getRobot
Director → RobotDirector
   construct → makeRobot
Product → Robot
```

Figure 4–22: Example Instantiations of *Builder*

- The RAM model with instantiation cardinalities is a lot more *compact*, while it still clearly visualizes how the model is intended to be used. It captures the essence of *Abstract Factory completely.* The standard OO diagram shows only two `ConcreteFactories` and two `AbstractProducts`. In OO design pattern diagrams that depict multiple subclasses of a common supertype, it is typically shown by two classes with similar names and adding a numeric suffix to the names (e.g. `ConcreteFactory1`, `ConcreteFactory2`). In RAM, the fact that there can be one or more `AbstractProducts {1..*}` whereas there need to be at least two `ConcreteFactories {2..*}` is clearly shown in the notation.
- Similarly, since the maximum cardinality can be `*`, the RAM notation is *scalable.* The OO diagram relies on different suffix types (numerical and alphanumerical) to show the independence of the number of subclasses of `AbstractProduct` and `AbstractFactory`. This technique becomes problematic in case a third set of independent subclasses needs to be specified. In RAM, a designer simply needs to introduce a different variable for every class that can exist independently multiple times, e.g., `{q=1..*}`.

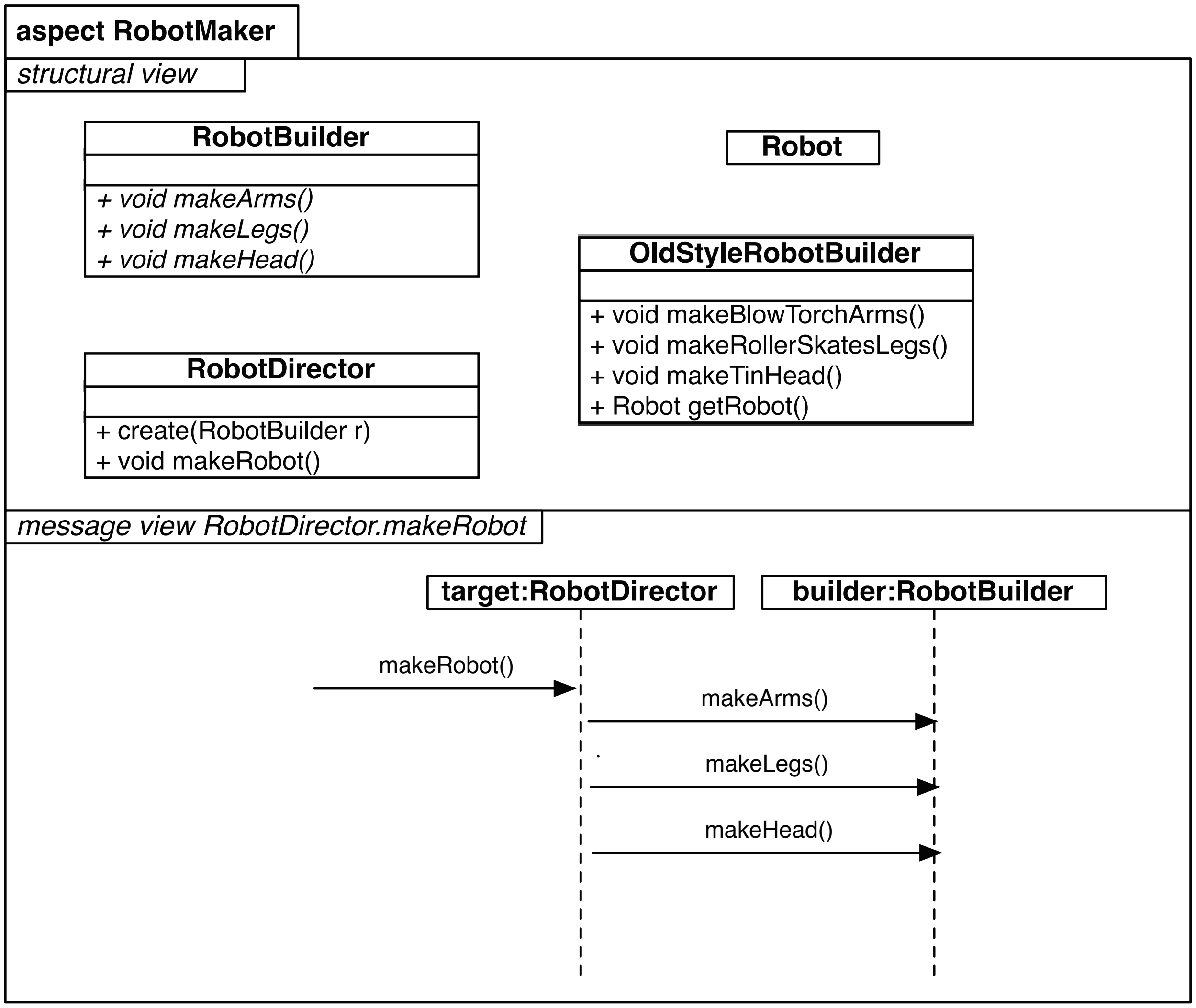


Figure 4–23: Example use of *Builder*

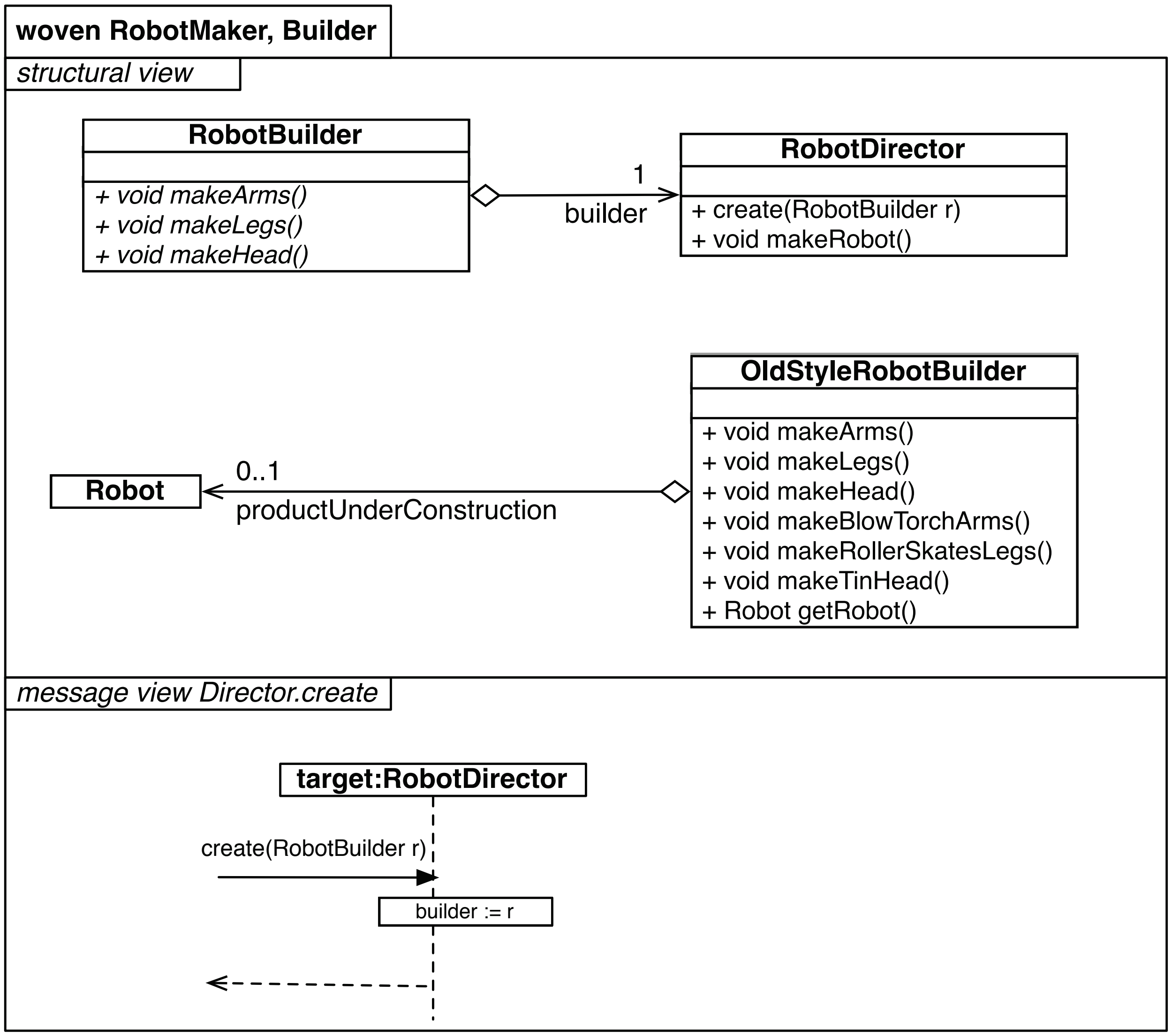
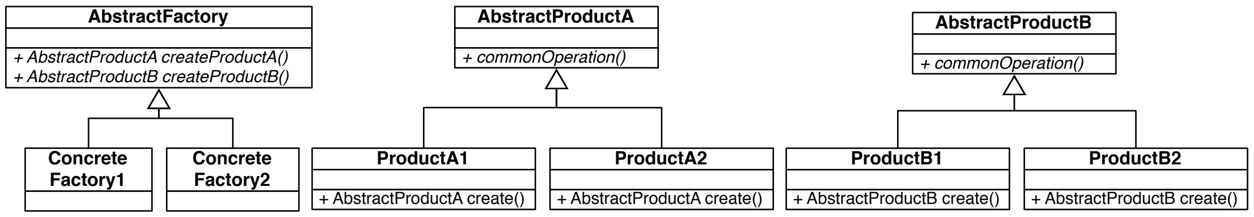


Figure 4–24: Woven Model using *Builder*



AbstractFactory
+ AbstractProductA createProductA()
+ AbstractProductB createProductB()
Concrete Factory1
Concrete Factory2
AbstractProductA
+ commonOperation()
ProductA1
+ AbstractProductA create()
ProductA2
+ AbstractProductA create()
AbstractProductB
+ commonOperation()
ProductB1
+ AbstractProductB create()
ProductB2
+ AbstractProductB create()

Figure 4–25: *Abstract Factory* in UML

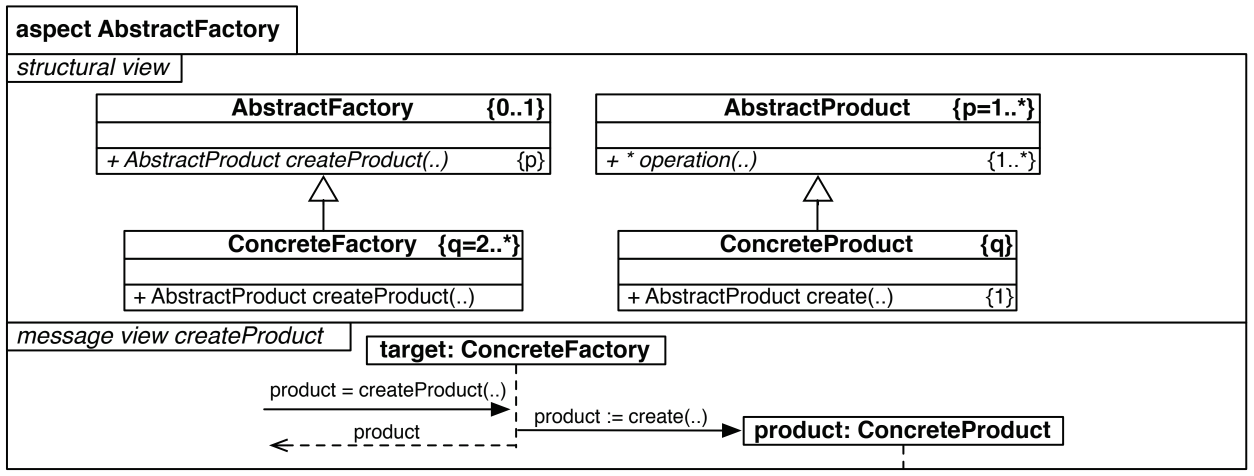


Figure 4–26: *AbstractFactory* RAM Model

- The RAM model shows the relationships between the number of classes *unambiguously*. In the standard OO diagram, the same kind of suffix is used to highlight the fact that the same number of subclasses is needed. For instance, there are subclasses `ConcreteFactory` and two subclasses `ProductA`. However, it is not clear whether from a design point of view number of `ConcreteFactories` is determined by the number of `ProductA`s, or if it is the other way round. In RAM, since instantiation cardinalities allow the possibility of *declaring* and *using* variables, it is clear that:

  - The number of different `AbstractProducts {p=1..*}` (variable `p` is declared) determines the number of constructor methods in the `AbstractFactory` class `{p}` (variable `p` is used).
  - For each `AbstractProduct`, there must be as many `ConcreteProduct` subclasses `{q}` (variable `q` used) than there are `ConcreteFactories {q=2..*}` (variable `q` is declared).
  - There is no direct relation between the number of `ConcreteFactories {q=2..*}` and `AbstractProducts {p=1..*}` (they *declare different variables* `p` and `q`).

- In one message view it is possible to define the behaviour for all `createProduct` operations of all `ConcreteFactories`, i.e., for p*q methods! Because of the different variable declarations, the weaver knows that when generating the message view for *createProduct<i,j>*, it is supposed to call the `create` method of the `jth` mapping of the `ConcreteProduct` subclass of the `ith` mapping of the `AbstractProduct` class. For example, given the instantiation in Fig. 4–27, the weaver can generate the message view `ToyotaFactory.createTruck` that calls `ToyotaTruck.create`.

```
AbstractFactory → VehicleFactory
   createProduct<1> → createCar
   createProduct<2> → createTruck
ConcreteFactory<1> → Toyota
ConcreteFactory<2> → Honda
AbstractProduct<1> → Car
   operation<1> → drive
AbstractProduct<2> → Truck
   operation<1> → drive
   operation<2> → load
AbstractProduct<3> → Motorcycle
   operation<1> → ride
ConcreteProduct<1,1> → ToyotaCar        (the first mapping dimension refers to
   create → buildToyotaCar               the mapping of the superclass)
ConcreteProduct<1,2> → HondaCar
   create → buildHondaCar
ConcreteProduct<2,1> → ToyotaTruck
   create → buildToyotaTruck
ConcreteProduct<2,2> → HondaTruck
   create → buildHondaTruck
ConcreteProduct<3,1> → ToyotaMotorcycle
   create → buildToyotaMotorcycle
ConcreteProduct<3,2> → HondaMotorcycle
   create → buildHondaMotorcycle
```

Figure 4–27: Example Instantiation of *AbstractFactory*

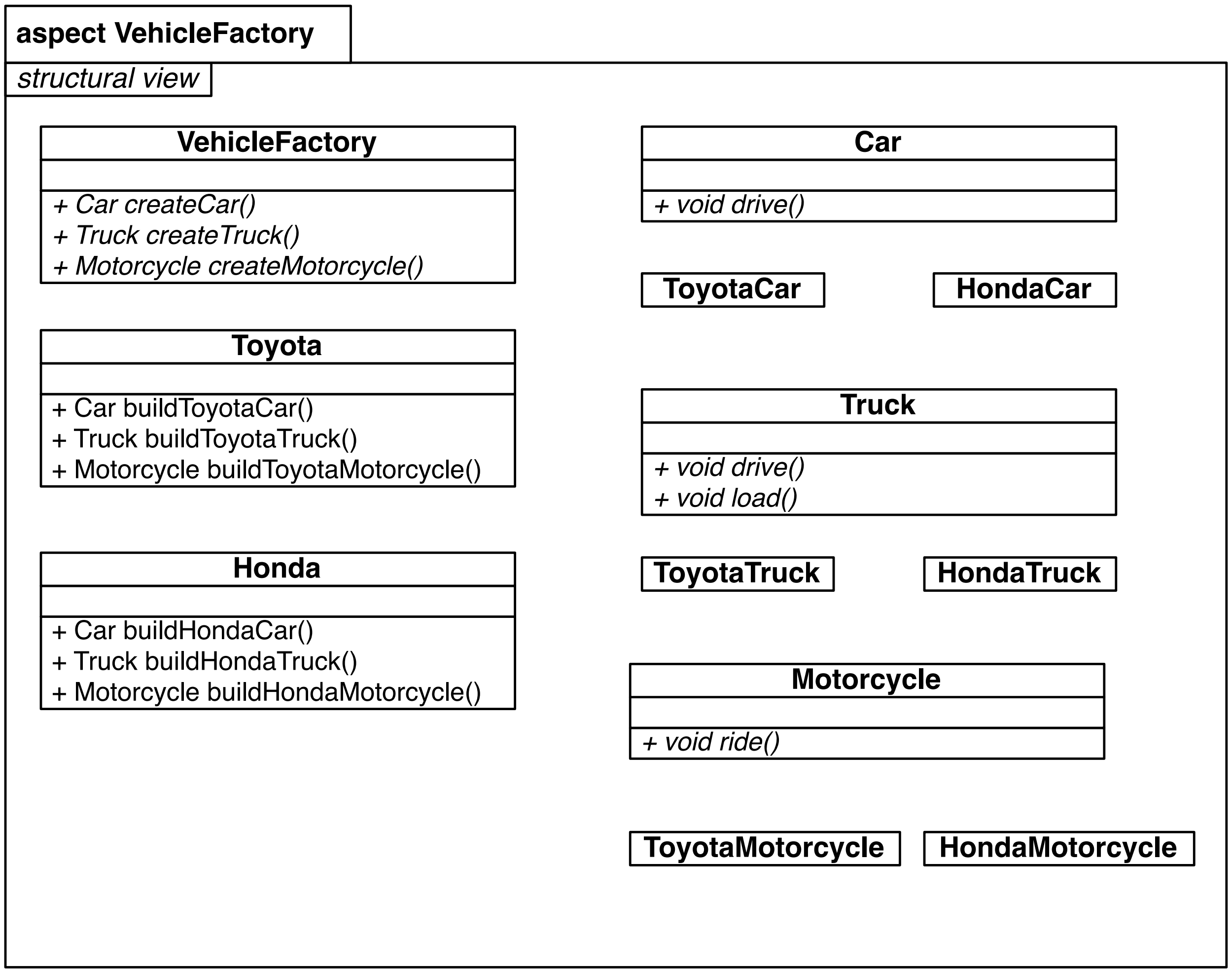


Figure 4–28: Example use of *AbstractFactory*

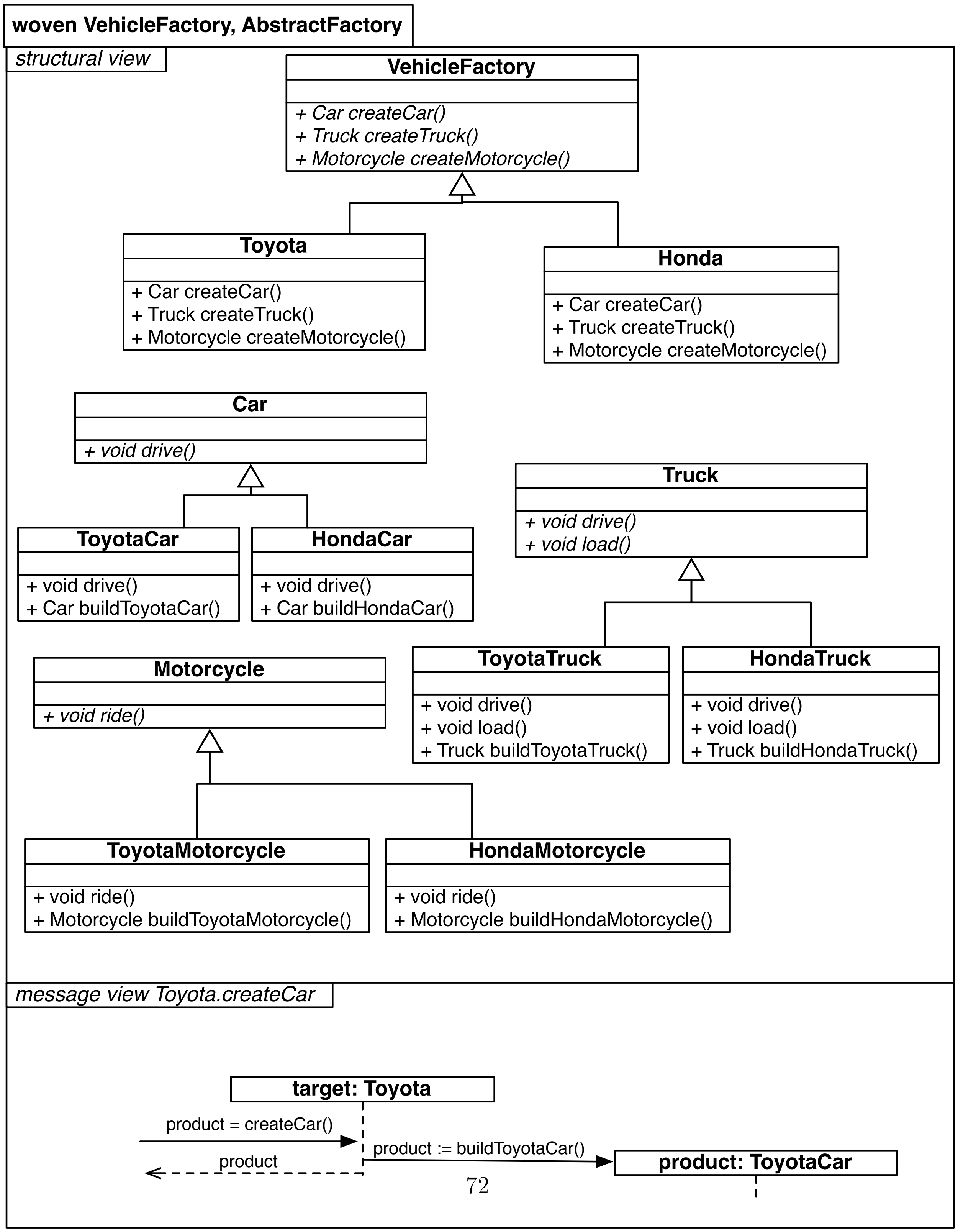


Figure 4–29: Woven Model using *AbstractFactory*

Chapter 5
Related Work

Section 5.1 reviews the most well-known AOM approaches, focussing on describing those features that are related to instantiations and model element creation. Section 5.2 presents related work in programming languages.

5.1 Aspect-Oriented Modelling

Aspect-oriented modelling techniques have been applied to extend many popular modelling notations. For example, AOM approaches have been proposed for UML class diagrams [13, 35], sequence diagrams [21, 20], and protocol models [24, 25]. Our own RAM approach integrates these three approaches into one multi-view modelling approach, to which we have shown how instantiation cardinalities can be applied in this thesis.

Other AOM approaches target modelling notations such as state diagrams [12, 40, 16], live sequence charts [23], activity diagrams, the Specification and Description Language (SDL) [10, 11], the User Requirements Notation (URN) [30, 29], and more. To the best of our knowledge, none of these AOM approaches provide customization interfaces for aspect models that expose information equivalent to what instantiation cardinalities provide.

We therefore believe that they could also benefit from adding instantiation cardinalities to their models in a way that is similar to how we extended RAM. Some more information on certain related approaches is presented in the following subsections.

5.1.1 Theme/UML

For example, in Theme/UML [9], the models that contain crosscutting structure and behaviour are called themes. A theme is a parameterized UML package, and it exposes the generic model elements that must be bound to application specific elements in form of UML template parameters. Just like in RAM before the introduction of instantiation cardinalities, it is not obvious for a modeller to know if she can bind a parameter to several model elements (similar to multi-mapping in RAM), or rather bind a theme multiple times to elements in a target model. Furthermore, when a theme is applied n times, it seems like all model elements that are created by the theme will be introduced n times as well.

5.1.2 HiLA

Zhang et al. propose the *High-Level Aspects for UML State Machines (HiLA)* approach [40, 16], in which they significantly extend UML state machines with aspect-oriented modelling techniques. They use state machines to specify behaviour of base machines and aspect machines, which can be parameterized using UML template parameters similar to RAM models. They provide several asymmetric pointcut-advice composition mechanisms that enable aspects to disallow and restrict transitions, describe mutual exclusion between two states in orthogonal regions and coordinate multiple state machines. In the case of HiLA, instantiation cardinalities could be used to prevent orthogonal, concurrent states to be introduced multiple times when an aspect is applied several times within a target model.

5.1.3 MATA

MATA [39] is a graph-based approach for composing UML diagrams that supports pattern matching to determine where an aspect model is to be applied. If in the aspect model a model element is tagged with the stereotype `<<create>>`, it means that this model element is created in the target model whenever the pattern matches. This is equivalent to the instantiation policy *PerPointcut-Match* described in [28]. [8] later extended the notation with additional stereotypes `<<create++>>` for introducing new model elements into a package common to all aspect models (equivalent to the *Global* policy described in [28]), `<<create+>>` to introduce new model elements into a package common to all pattern matches, and `<<create->>` to introduce new model elements into a new package that is specific to each parameter binding. Although this allows for more fine-grained control over how many times model elements are introduced when an aspect is applied, it does not help the designer decide on whether to write one complex pattern match or several specific ones.

5.1.4 SmartAdapters

As mentioned in section 2.2, the paper [28] points out different introduction policies that a pattern-based model weaver must support when aspect models are applied multiple times within the same target model. The different policies are:

- *PerPointcut-Match*, where new instances of the element are created each time the aspect model is instantiated,
- *Global*, where only a single instance is created regardless of how many times the aspect model is instantiated,
- *PerMatchedElement*, where a new instance is created for each model element that is matched by the pointcut pattern, regardless of what role in the pattern it plays, and
- *PerMatchedRole*, where a new instance is created each time a model element is matched by the pointcut pattern in a different role.

The paper does not talk about how these policies could be specified by the model designer, but it explains how the different policies can be implemented inside a model weaver using the SmartAdapters [27] model transformation language. Although the implementation is interesting, the presented techniques can not be applied as such to the TouchRAM tool, since RAM does not use patterns for specifying multi-mappings.

5.2 Aspect-Oriented Programming Languages

There has been a lot of work on introducing advanced modularization features into programming languages that is related to instantiation cardinalities.

5.2.1 Control over Aspect Instances

At a programming level, some aspect-oriented programming languages have introduced features that give the programmer fine-grained control over the number of instances of aspects that are created at run-time.

In *AspectJ* [18], for example, an aspect has per default only one instance that cuts across the entire program. Consequently, because the instance of the aspect exists at all join points in the running of a program (once its class is loaded), its advice is run at all such join points. However, *AspectJ* also proposes some elaborate aspect instantiation directives, such as: 1) *perthis(pointcut)* aspects, meaning that an instance of the aspect is created for every different object that is executing when the specified pointcut is reached; *pertarget(pointcut)*, meaning that an instance of the aspect is created for every object that is the target object of the join points matched by pointcut; 3) *percflow(pointcut)*, meaning that an instance of the aspect is created for each flow of control of the join points matched by the specified pointcut. These elaborate aspect instantiations are all dynamic, i.e., they are based on the execution of a program, and might become relevant in future AOM approaches that support execution of models.

5.2.2 AOP and the Observer Pattern

Several works on AOP have used design patterns to illustrate the shortcomings of the modularization features of popular AOP languages, and presented new modularization techniques to address these drawbacks.

CaesarJ

In [26], Mezini et al. use the *Observer* design pattern to point out several deficiencies of *AspectJ*'s join point interception model, namely:

- *Lack of support for sophisticated mappings*: the authors demonstrate with examples that the mapping from aspect abstractions to base classes via the *declare parents* construct is effective only when each aspect abstraction has a corresponding base class.
- *Lack of support for reusable aspect bindings*: the authors argue that the aspect-to-class binding achieved via the *declare parents* construct strongly binds an aspect to a particular base class; hence, such bindings cannot be effectively reused.
- *Lack of support for aspectual polymorphism*: this limitation is comparable to the lack of support for per-object association of aspects identified in this paper. The paper argued that it is not possible in *AspectJ* to determine at runtime whether an aspect should be applied or not, or which implementation of the aspect to apply.

The authors then proposed a new aspect-oriented programming tool called *CaesarJ* [5] to address these deficiencies. *CaesarJ* is based on *Aspect Collaboration Interfaces* (ACI). In ACIs, the aspect implementation is decoupled from the aspect binding in independent, indirectly connected modules. *CaesarJ* relies on a new type called a *weavelet* to compose the implementation and the binding of the aspect to form the final system. Different *weavelets* can combine an aspect binding with different aspect implementations, or a particular aspect implementation with different aspect bindings; making both the aspect bindings and implementations independently reusable. As opposed to *AspectJ*, compiling these *weavelets* with the base application does not have any effect on the execution of the application. This is because the *weavelets* must be explicitly deployed to activate their pointcuts and advice. The *weavelets* can be deployed statically or dynamically; hence, the support for runtime deployment of aspects on a per-object basis.

Sally

In [15], the authors investigate how design patterns can be modularized and implemented in *AspectJ* [18] and *Hyper/J* [32]. They point out that both languages fail to encapsulate design patterns in a reusable way and illustrate the problems using the *Singleton*, *Visitor* and *Decorator* design patterns. They then proceed to present *Sally*, an AspectJ-like programming language that provides the programmer with parametric introductions, a language feature which can be seen as the aspect-oriented equivalent to generic types or parametric types. A parametric introduction allows an aspect to statically declare fields that are to be introduced into other classes just like standard introductions, but in addition they can be parameterized with types that are determined during weave-time, i.e., when the introduction is bound to a specific class. The authors then demonstrate how parametric introductions solve the inadequacies for modularizing design patterns effectively.

Package Templates

In [6], the authors utilize the *package template* mechanism with a small aspect-oriented extension to provide a reusable package for the *Observer* pattern. A package template (PT) is a mechanism for code modularization that targets the development of collections of reusable interdependent classes. The authors describe package templates as being syntactically derived from Java packages with significant semantic differences. A package template can be instantiated at compile time, which creates a local copy of template classes. The authors then introduce a minimal set of constructs for AOP extensions to the PT mechanism. Pointcuts and advice are defined as members of template classes, and aspects are limited in scope by corresponding template instantiation. The authors then demonstrate their approach using the *Observer* pattern with single subject, single observer classes and multiple subject, multiple observer classes.

Chapter 6
Conclusion

In this thesis we have presented *instantiation cardinalities*, a novel concept useful in the context of aspect-orientation in general and aspect-oriented modelling in particular. It allows the designer of a reusable aspect that comprises multiple structural entities to:

- Specify the customization interface of the module, i.e., highlight which entities are generic and need to be completed with application-specific structure in order for the reusable aspect to be usable in a specific context, and
- Clearly specify maximally how many times each structural entity can be mapped to application-specific entities.

By declaring and using variables within the instantiation cardinality specification, dependencies between the number of mappings of structural entities can be expressed in a precise way. This solves the inherent ambiguity that most aspect-oriented approaches exhibit when it comes to reusing existing aspects within an application, and gives the model designer fine-grained control over how many instances of each model element are created in the target model. As a result, the designer of the reusable aspect is able to specify all the instantiation policies identified in [28].

Since most aspect-oriented modelling approaches and programming languages include object-oriented features as well, the thesis describes how to integrate instantiation cardinalities with standard object-oriented concepts such as inheritance and polymorphism. We described the effects that the cardinalities of a superclass and its methods have on its subclasses and the methods that they override. Furthermore, we introduced a technique called *automated call forwarding* that is applied by the weaver in order to allow for polymorphic treatment of multi-mapped subclasses in one model, while not requiring uniform naming of polymorphically related operations in each individual subclass.

In order to illustrate the usefulness of instantiation cardinalities, this thesis presented how instantiation cardinalities integrate with the *Reusable Aspect Models* approach. Furthermore, the practicality and elegance of the approach was demonstrated by showing the detailed aspect-oriented design models of seven design patterns: the creational design patterns *Builder* and *Abstract Factory*, the structural design patterns *Composite* and *Decorator*, as well as the behavioural design patterns *Observer*, *Template Method* and *Command*.

6.1 Future Work

We are convinced that *instantiation cardinalities* is a very powerful approach that solves significant problems. The possibilities for future work are as follows:

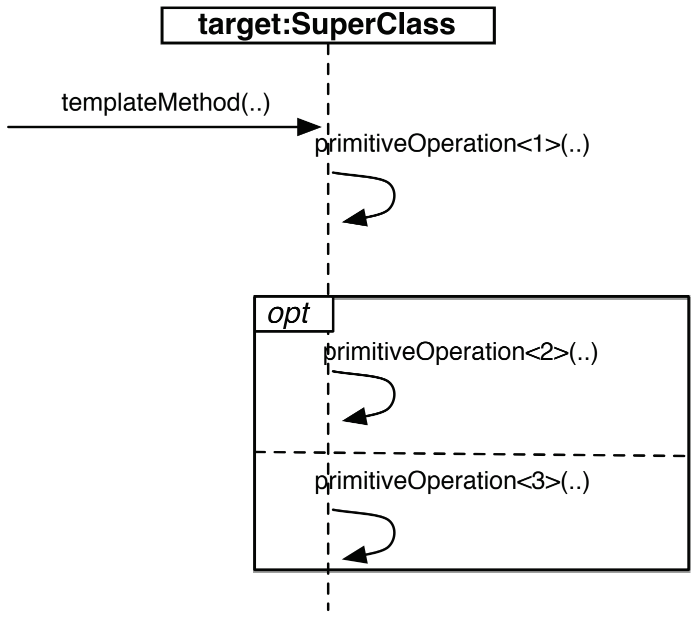
1. It might be possible to extend the approach of instantiation cardinalities to allow a model designer to specify minimum and maximum number of parameters for constructors and methods. As of now, we have depended on well-known design patterns to develop the ideas of instantiation cardinalities. However, parameters do not feature significantly enough in the design patterns to be able to derive the insight to extend cardinalities in a convincing manner to parameters. TouchRAM, the RAM tool that is built to demonstrate and experiment with RAM, is currently under active development and it is hoped that the use of TouchRAM to create complex real-world software will

provide new insights on how the idea of instantiation cardinalities can be extended to include parameters.

2. Instantiation cardinalities have solved persisting issues with RAM in a manner that we found to be elegant. While working on the thesis, we tried other approaches to solve these issues. We found that with all the other approaches that were tried, significantly more effort was involved in communicating the working of the approach. Based on our experience, the approach of instantiation cardinalities is easy to understand and straightforward to apply by a model user. The approach is amenable to user feedback for further improvement as well as verification of its usefulness. A user study should be conducted to study the efficacy of the approach and its effect on learning time once TouchRAM is sufficiently developed.
3. There are also indications that it might be possible to get rid of *abstract methods* in the RAM metamodel. In this work, apart from instantiation cardinalities we have also introduced the notion of *Automated Call Forwarding.* In Java, abstract methods have two important characteristics: they are not allowed to have a method definition and they must be overridden in subclasses. In fact, it might be argued that the primary reason that they are not allowed to have a method definition is *because* they must be overridden. Also, the idea of overriding can be rephrased as saying that the subclass must provide its own definition for the method. In the Composite Pattern (4.1.1) it was shown that it is possible in RAM to a force a method in a subclass to have message view by having {1} cardinality for the method. This means that when an aspect is reused, instantiation cardinalities can provide the exact same information to a method, that is, it must be defined. From our experience, it is not clear if not allowing a method

to have a definition is important by itself. However, to be able to say that abstract methods are not needed with confidence, we will need to gather more experience using instantiation cardinalities to model real-world software systems.

4. RAM currently does not allow a higher-level aspect to modify the control flow of existing message views, nor can a lower-level aspect define constraints that must be upheld by message views defined in a higher-level aspect. In the *TemplateMethod* aspect shown in Fig. 4–9, for example, the designer wants to specify that any concrete definition of `templateMethod` in a higher-level aspect should call each of the operations `primitveOperation<x>` at least once. For example, the situation shown in Fig. 6–1 where a user wants to always call `primitiveOperation<1>`, and then use an *alt* fragment to call either one of the methods, `primitiveOperation<2>` or `primitiveOperation<3>` in a higher-level aspect would be a correct behavioural refinement for `templateMethod`. Further research is needed to determine the best way of addressing this issue. One approach could be to extend the notion of pointcut and advice to allow arbitrary modifications of behaviour defined in lower-level aspects. Another approach could be to define language features that allow for the specification of constraints in lower-level aspects.


target:SuperClass
templateMethod(..)
primitiveOperation<1>(..)
opt
primitiveOperation<2>(..)
primitiveOperation<3>(..)

Note: In actual woven view, the different mappings of primitiveOperation(..) would be visible and not primitiveOperation<1>(..), etc. The diagram above is for illustration only.

Figure 6–1: Refining the Behaviour of Template Method in a Higher-Level Aspect

References

[1] The Eclipse Project. URL: http://www.eclipse.org, 2011.

[2] NewThinkTank Design Patterns. URL: http://www.newthinktank.com, 2014.

[3] Wisam Al Abed, Valentin Bonnet, Matthias Schöttle, Omar Alam, and Jörg Kienzle. TouchRAM: A multitouch-enabled tool for aspect-oriented software design. In *5th International Conference on Software Language Engineering - SLE 2012*, number 7745 in LNCS, pages 275 – 285. Springer, October 2012.

[4] Wisam Al Abed and Jörg Kienzle. Information Hiding and Aspect-Oriented Modeling. In *14th Aspect-Oriented Modeling Workshop, Denver, CO, USA, Oct. 4th, 2009*, pages 1–6, October 2009.

[5] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135–173, 2006.

[6] Eyvind W. Axelsen, Fredrik Sørensen, and Stein Krogdahl. A reusable observer pattern implementation using package templates. In *Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, ACP4IS '09, pages 37–42, New York, NY, USA, 2009. ACM.

[7] Abir Ayed and Jörg Kienzle. Integrating Protocol Modelling into Reusable Aspect Models. In *Proceeding of the 5th ACM SIGCHI Annual International Workshop on Behaviour Modelling - Foundations and Applications - BM-FA 2013, Montpellier, France*, pages 1–12. ACM, July 2013.

[8] Jorge Barreiros and Ana Moreira. Reusable model slices. In *14th Aspect-Oriented Modeling Workshop, Denver, CO, USA, Oct. 4th, 2009*, October 2009.

[9] Andrew Carton, Cormac Driver, Andrew Jackson, and Siobhan Clarke. Model-driven theme/uml. In *Transactions on Aspect-Oriented Software Development VI*, volume 5560 of *Lecture Notes in Computer Science*, pages 238–266. Springer, 2009.

[10] Thomas Cottenier, Aswin Van Den Berg, and Tzilla Elrad. The Motorolla WEAVR: Model Weaving in a Large Industrial Context. In *Industry Track of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany, 2006. ACM.

[11] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Stateful aspects: The case for aspect-oriented modeling. In *Proceedings of the 10th International Workshop on Aspect-oriented Modeling*, AOM '07, pages 7–14, New York, NY, USA, 2007. ACM.

[12] Tzilla Elrad, Omar Aldawud, and Atef Bader. Expressing aspects using UML behavioral and structural diagrams. In R.E. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development*, pages 459–478. Addison-Wesley, 2005.

[13] Robert France, Indrakshi Ray, Geri Georg, and Sudipto Ghosh. Aspect-oriented approach to early design modelling. *IEE Proceedings Software*, pages 173–185, August 2004.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, USA, 1995.

[15] Stefan Hanenberg and Rainer Unland. Parametric introductions. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, AOSD '03, pages 80–89, New York, NY, USA, 2003. ACM.

[16] Matthias Hölzl, Alexander Knapp, and Gefei Zhang. Modeling the Car Crash Crisis Management System with HiLA. *Transactions on Aspect-Oriented Software Development VII*, LNCS 6210:234–271, 2010.

[17] Stuart Kent. Model Driven Engineering. In *International Conference on Integrated Formal Methods – IFM*, pages 286–298, London, UK, 2002. Springer-Verlag.

[18] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[19] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-Oriented Multi-View Modeling. In *AOSD 2009*, pages 87 – 98. ACM Press, March 2009.

[20] Jacques Klein, Franck Fleurey, and Jean Marc Jézéquel. Weaving multiple aspects in sequence diagrams. *Transactions on Aspect-Oriented Software Development (TAOSD)*, III:167–199, 2007.

[21] Jacques Klein, Loïc Hélouët, and Jean-Marc Jézéquel. Semantic-based weaving of scenarios. In *Proceedings of the 5th International Conference on Aspect-oriented Software Development*, AOSD '06, pages 27–38, New York, NY, USA, 2006. ACM.

[22] Thomas Kühne. Matters of (Meta-) Modeling. *Software and Systems Modeling*, 5:369 – 385, December 2006.

[23] Mark Mahoney and Tsilla Elrad. Weaving crosscutting concerns into live sequence charts using the play engine. In *7th International Workshop on Aspect-Oriented Modeling, Montego Bay, Jamaica, Oct. 2nd, 2005*, 2005.

[24] Ashley McNeile and Ella Roubtsova. Composition semantics for executable and evolvable behavioral modeling in mda. In *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, BM-MDA '09, pages 3:1–3:8, New York, NY, USA, 2009. ACM.

[25] Ashley McNeile and Ella Roubtsova. Aspect-oriented development using protocol modeling. *Transactions on Aspect-Oriented Software Development VII*, pages 115–150, 2010.

[26] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development*, AOSD '03, pages 90–99, New York, NY, USA, 2003. ACM.

[27] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@run.time to support dynamic adaptation. *IEEE Computer*, 42(10):44–51, October 2009.

[28] Brice Morin, Jacques Klein, Jörg Kienzle, and Jean-Marc Jézéquel. Flexible Model Element Introduction Policies for Aspect-Oriented Modeling. In *13th International Conference on Model Driven Engineering Languages and Systems - MoDELS 2010, Oslo, Norway, Oct. 3 - 8th, 2010*, number 6395, pages 63 – 77, October 2010.

[29] Gunter Mussbacher, Daniel Amyot, João Araújo, and Ana Moreira. Requirements Modeling with the Aspect-oriented User Requirements Notation (AoURN): A Case Study. In Shmuel Katz, Mira Mezini, and Jörg Kienzle, editors, *Transactions on Aspect-Oriented Software Development VII*, volume 6210 of *Lect. Notes Comp. Sci.*, pages 23–68. Springer, 2010.

[30] Gunter Mussbacher, Daniel Amyot, and Michael Weiss. Visualizing Early Aspects with Use Case Maps. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development III*, volume 4620 of *Lect. Notes Comp. Sci.*, pages 105–143. Springer, 2007.

[31] Object Management Group. *Unified Modeling Language: Superstructure (v2.4.1)*, December 2011.

[32] Harold Ossher and Peri Tarr. Hyper/j: Multi-dimensional separation of concerns for java. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, pages 734–737, New York, NY, USA, 2000. ACM.

[33] F. Paas, J.E. Tuovinen, H. Tabbers, and P.W.M. Van Gerven. Cognitive load measurement as a means to advance cognitive load theory. *Educational psychologist*, 38(1):63–71, 2003.

[34] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the Association of Computing Machinery*, 15(12):1053–1058, December 1972.

[35] Y.R. Reddy, S. Ghosh, R. France, G. Straw, J. Bieman, N. McEachen, E. Song, and G. Georg. Directives for Composing Aspect-Oriented Design Class Models. *Transactions on Aspect-Oriented Software Development I*, LNCS 3880:75–105, 2006.

[36] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39:41–47, 2006.

[37] J. Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12(2):257–285, 1988.

[38] Jon Whittle. "the truth about model-driven development in industry - and why researchers should care". http://www.slideshare.net/jonathw/whittle-modeling-wizards-2012/, 2012.

[39] Jon Whittle, Praveen K. Jayaraman, Ahmed M. Elkhodary, Ana Moreira, and João Araújo. MATA: A unified approach for composing UML aspect models based on graph transformation. *Transactions on Aspect-Oriented Software Development VI*, 5560:191–237, 2009.

[40] Gefei Zhang and Matthias Hölzl. Hila: High-level aspects for uml state machines. In *Proceedings of the 2009 International Conference on Models in Software Engineering*, MODELS'09, pages 104–118, Berlin, Heidelberg, 2010. Springer-Verlag.