Reusable Model Interfaces with Instantiation Cardinalities

Sunit Bhalotia and Jörg Kienzle(✉)

School of Computer Science, McGill University, Montreal, Canada
`Sunit.Bhalotia@mail.mcgill.ca, Joerg.Kienzle@mcgill.ca`

Abstract. The power of aspect-oriented modelling is that structural and behavioural properties of a crosscutting concern can be modularized within an aspect model. With proper care, such an aspect model can also be made reusable. If the functionality provided by such a modularized concern is needed repeatedly within a system, the reusable aspect model can be applied multiple times within the same target model. This paper reviews the pending issues related to multiple aspect model instantiations identified in past research, and then proposes to extend the customization interface of aspect models with instantiation cardinalities. This removes potential customization ambiguities for the model user, and gives 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. The approach is illustrated by presenting the aspect-oriented design of a behavioural, a structural and a creational design pattern.

1 Introduction

In the context of model reuse, the *artifact designer*, i.e., the developer creating the reusable artifact, and the *artifact user*, i.e., the developer applying the artifact to a specific application, are usually different people. The designer has in-depth domain knowledge about the concern that the reusable artifact addresses, and knows about the specific details of the functionality / properties / solutions encapsulated by the artifact that he created. The user on the other hand has in-depth domain knowledge about the application he is building, knows that the application he is building could benefit from reusing the reusable artifact, but is unaware of the artifacts inner working and limitations.

Aspect-orientation is a development paradigm that adds a new dimension to modularization. In aspect-oriented modelling (AOM), aspect models encapsulate structural and behavioural elements related to a particular concern. With proper care, aspect models can also be made reusable. However, 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. 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 [6,14,19], whereas others allow the use of wildcards in so-called pointcut expressions [8,10,20].

© Springer International Publishing Switzerland 2015
G. Taentzer and F. Bordeleau (Eds.): ECMFA 2015, LNCS 9153, pp. 108–124, 2015.
DOI: 10.1007/978-3-319-21151-0_8

To the best of our knowledge, none of the current AOM approaches specifies precisely how a *model user* should go about applying a reusable aspect model multiple times. Since *the model user does not know about the inner workings and limitations* of the reusable aspect model, he is faced with multiple possibilities: instantiating an aspect model multiple times, specifying multi-mappings or multiple individual mappings, or specifying a single complex pointcut expression vs. using several pointcut expressions. Furthermore, it has been shown in [16] 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.

This paper presents *instantiation cardinalities*, a novel concept that solves the aforementioned ambiguity while giving the designer explicit control over the number of instances of each model element that is created in the target model. Our proposed approach is illustrated in this paper using the *Reusable Aspect Models* notation (RAM)) [14], an aspect-oriented multi-view modelling approach for software design modelling.

The remainder of the paper is structured as follows. Section 2 introduces model interfaces, aspect-oriented modelling and the problem with multiple applications of the same aspect within a target model. Section 3 presents instantiation cardinalities, and illustrates them by means of the *Observer* design pattern. Section 4 introduces automated call forwarding, an extension to aspect-oriented weaving that integrates polymorphism and separation of concerns in the presence of multi-mappings. Section 5 illustrates the elegance of instantiation cardinalities by showing the aspect-oriented design of two additional design patterns. Section 6 presents related work, and the last section draws some conclusions.

2 Background

To explain the motivation behind this work, the first background subsection gives a brief overview of units of reuse for software design and the kind of interfaces that these units define. The next subsection introduces aspect-oriented modelling in general and the *Reusable Aspect Models* approach in particular, and how it can be used to express crosscutting software design concerns. The last subsection illustrates the ambiguity that a software developer faces when reusing aspect models, and highlights the need for flexible instantiation policies.

2.1 Interfaces

Units of reuse, e.g., units used in software design such as classes, components, frameworks, design patterns [9], software product lines [18], etc..., typically either explicitly or implicitly define *interfaces* that detail *how* the unit is supposed to be reused. [3] classifies these interfaces into three kinds: *usage*, *customization*, and *variation interfaces*.

Usage Interface: The *usage interface* is the interface that is most common. For units that are used in software design, it *specifies the design structure and* *behaviour that the unit provides* to the rest of the application. In other words, the usage interface presents an abstraction of the functionality encapsulated within the unit to the developer. It describes *how* the application can trigger the functionality provided by the unit.

For instance for classes, the usage interface is the set of all *public* class properties, i.e., the attributes and the operations that are visible and accessible from the outside. For components, the usage interface is the set of services that the component provides (i.e., the *provided interface*). For frameworks, design patterns, and SPLs, the usage interface is comprised of the usage interfaces of all the classes that the framework/pattern/SPL offers.

Customization Interface: Typically, a unit of reuse has been purposely created to be as general as possible so that it can be applied to many different contexts. As a result, it is often necessary to tailor the general design to a specific application context. The *customization interface* of a reusable software design unit *specifies how to adapt the reusable unit to* the *specific needs* of the application under development.

For example, the customization interface of generic classes (also called template classes) allows a developer to customize the class by instantiating it with application-specific types. For components, the customization interface is comprised of the set of services that the component expects from the rest of the application to function properly (i.e., the *required interface*). The developer can use this information at configuration time to plug in the appropriate application-specific services. The customization interface for frameworks and design patterns is often comprised of interfaces/abstract classes that the developer has to implement/subclass to adapt the framework to perform application-specific behaviour.

Variation Interface: The *variation interface* of a reusable software design unit describes the available design variations and the impact of the different variants on high-level goals, qualities, and non-functional requirements. A variation interface typically takes the form of a *feature model* [12] that specifies the individual features that the unit offers. The impact of choosing a feature can be specified with goal models [11].

A reusable software design unit needs a variation interface only if it encapsulates several different design alternatives. In this paper, the focus is on improving reuse of a single design, and hence the variation interface is out of scope.

2.2 Aspect-Oriented Modelling

Aspect-orientation adds a new dimension to modularization, because the structure and functionality that aspects define can have a crosscutting effect on the rest of the application. In aspect-oriented modelling (AOM), aspect models encapsulate structural and behavioural elements related to a particular concern. Typically, the different elements within an aspect model need to interact closely with each other, i.e., invoke each other's behaviour or consult each other's state. The potentially crosscutting nature of the concern also requires that the

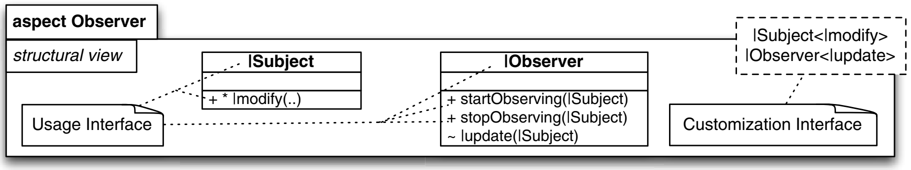

Fig. 1. Observer RAM Model Interface (Customization and Usage)

structure and functionality provided by the model can be applied several times within the same application.

To make aspect models reusable, interfaces are key. In software design modelling, having an explicit model interface makes it possible to apply proper information hiding principles [17] by concealing internal design details from the rest of the application. Because of aspect-oriented techniques, this is possible even if the encapsulated design details crosscut the rest of the application design. This is exemplified by our own *Reusable Aspect Models* approach (RAM)) [14], where each model has well-defined *usage* and *customization interfaces* [2].

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. To illustrate this, the usage interface of the RAM design of the *Observer* design pattern is shown in Fig. 1. The *Observer* design pattern [9] 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. 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 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. Fig. 1 shows that the customization

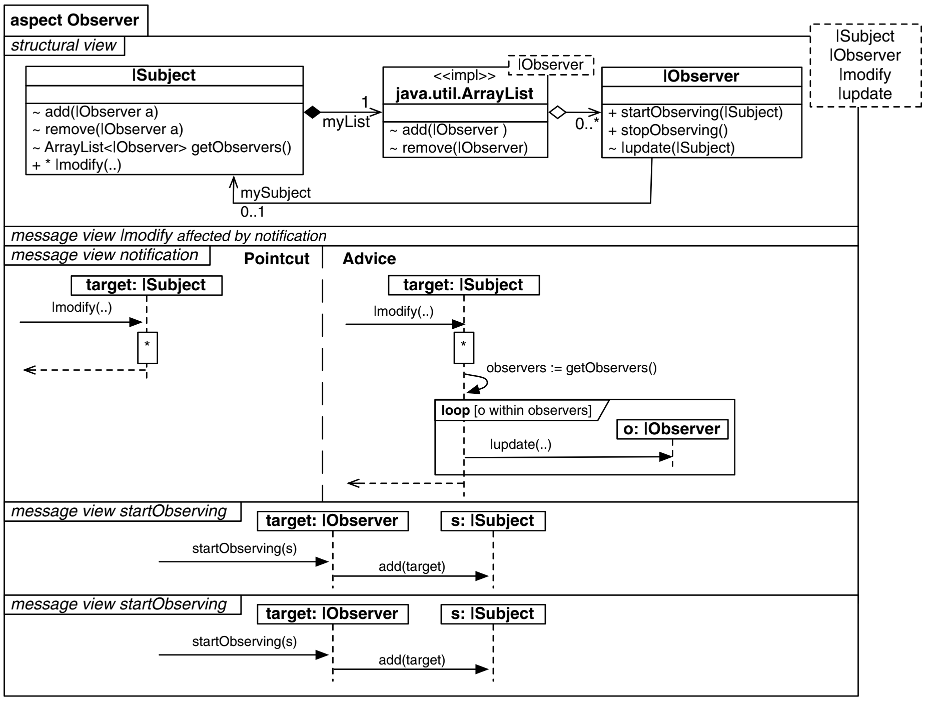

Fig. 2. Internal Design of Observer Aspect

interface for the *Observer* model comprises the class `|Subject` with a `|modify` operation, and the class `|Observer` class with an `|update` operation.

Fig. 2 shows a possible internal design for the *Observer* aspect. The subject maintains an `ArrayList` of `Observers` referenced by `myList`. The *notification* message view states that the behaviour of `|modify` is augmented to invoke `|update` on all registered observer instances after the behaviour of `|modify` completed execution.

2.3 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

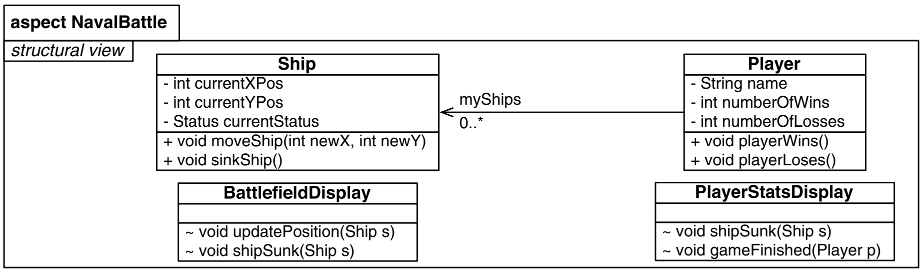

Fig. 3. Simplified Naval Battle Base Model

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 the current design model. If desired, *TouchRAM* [1], 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.

Fig. 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, and how many ships they sunk.

In such a design, the modeller may want to reuse the *Observer* concern shown in Fig. 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 `Battlefield-Display` instances that previously registered with the ship 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 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 [16]. 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 [16]). 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 that new instances should be created only for a given matched set or tuple of model elements in the target model (*PerMatchedElement* or *PerMatchedRole*).

3 Instantiation Cardinalities

This section introduces an extension to the customization interface that addresses the issues introduced in subsection 2.3: 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

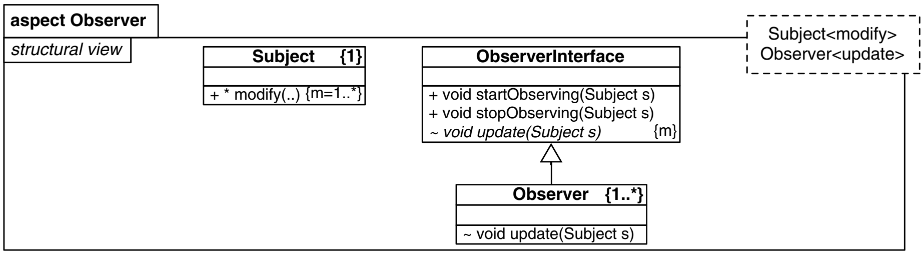

Fig. 4. Observer Model with Instantiation Cardinalities

have fine-grained control about how many instances of a specific model element defined in an aspect model are introduced into the target model.

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. The instantiation cardinality is shown in curly brackets to the right of the name of the model element.

Fig. 4 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 2.3 (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 notation we also define a default cardinality, i.e., {`0..1`}.

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. 4 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 to the

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

variable `m` in the `Subject` class (by specifying {`m=1..*`}), we are able to express this constraint on the `update` method of the `ObserverInterface` (by specifying {`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 called on a `Player` instance, `gameCompleted` of the registered `PlayerStatsDisplay` instances is invoked.

4 Weaver Considerations

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 follows:

- If the instantiation multiplicity of class A is {0}, {0..1} or {1}, and the instantiation multiplicity of B is {0}, {0..1} or {1}, then one single instance of r is created in the target model.
- If the instantiation multiplicity of class A is {q=1..*}, and the multiplicity of B is {0}, {0..1} or {1}, then q instances of the relationship r are created in the target model.
- If the instantiation multiplicity of class A is {q=1..*}, and the multiplicity 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.
- If the instantiation multiplicity of class A is {q=1..*}, and the multiplicity 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.

4.1 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 from above, if the player statistics are kept on a remote web server, then one might want to reuse the *Observer* aspect model defined in Fig. 4 as follows:

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

5 Design Patterns Revisited

Section 3 introduced instantiation cardinalities by means of the *Observer* behavioural design pattern. This case study section applies our ideas to two additional design patterns, the structural design pattern *Composite* [9] and the creational design pattern *Abstract Factory* [9], in order to demonstrate the elegance of instantiation cardinalities.

5.1 Composite

The *Composite* design pattern is a well-known structural design pattern that allows individual objects and collections of objects to be treated uniformly [9]. 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. 5 shows that the RAM structural view of the *Composite* pattern is similar to the classic OO UML diagram found in [9]. Instantiation cardinalities

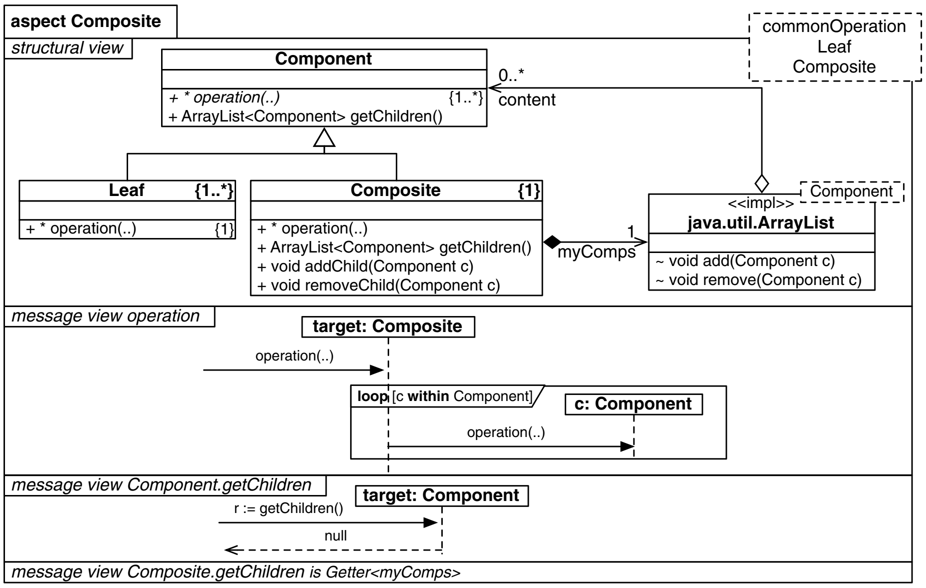

Fig. 5. The *Composite* RAM Model

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` 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.

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

```
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
```

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.

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

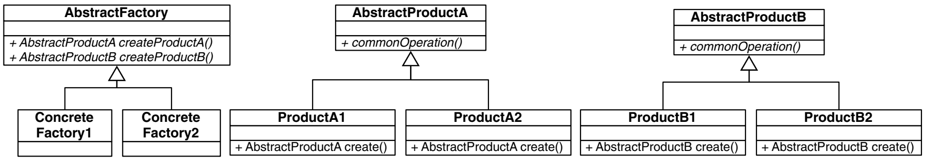
AbstractFactory
+ AbstractProductA createProductA()
+ AbstractProductB createProductB()
Concrete Factory1
Concrete Factory2
AbstractProductA
+ commonOperation()
ProductA1
+ AbstractProductA create()
ProductA2
+ AbstractProductA create()
AbstractProductB
+ commonOperation()
ProductB1
+ AbstractProductB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

Fig. 6. Abstract Factory in UML

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. Because of automated call forwarding, she can simply map the methods to `operation` defined in *Composite* without the need to modify any existing method names. When the two aspects are woven together by the *TouchRAM* tool, the weaver will create a `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.

5.2 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 [9].

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. 6 and 7 highlight the difference between a standard *Abstract Factory* UML diagram (taken from [9]) and the *Abstract Factory* RAM model. The advantages of using instantiation cardinalities are obvious:

- 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

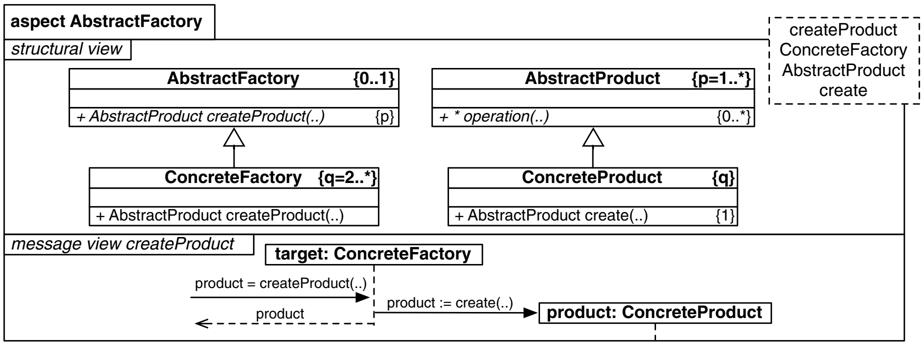

Fig. 7. Abstract Factory with Instantiation Cardinalities

(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..*`}.
- 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 the number of `ConcreteFactories` is determined by the number of `ProductAs`, 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 `j`th mapping of the `ConcreteProduct` subclass of the `i`th mapping of the `AbstractProduct` class. For example, given the instantiation in Fig. 8, the weaver can, for example, 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 → drive
AbstractProduct<2> → Truck
    operation<1> → drive
    operation<2> → load
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
```

Fig. 8. Example Instantiation of *AbstractFactory*

6 Related Work

To the best of our knowledge, none of the well-known AOM approaches provide customization interfaces for aspect models that expose information equivalent to what instantiation cardinalities provide.

For example, in Theme/UML [6], 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 the 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.

MATA [20] 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 [16]. [5] 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 [16]), `<<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 model user decide on whether to write one complex pattern match or several specific ones.

We believe that the AOM approaches presented above, and even others such as HiLA [10], GeKo [15] or the Motorola WEAVR [7], could benefit from adding instantiation cardinalities to their models in a way that is similar to how we extended RAM.

At a programming level, some aspect-oriented programming languages have introduced features that give the programmer fine-grained control over the number of instantiations of aspects. In *AspectJ* [13] 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.

Many other AOP approaches, such as package templates [4], provide means to instantiate crosscutting aspects/modules/templates multiple times and to resolve arising inconsistencies by specifying renamings/mappings. However, this does not solve the problem in the context of reuse, where the designer needs to communicate to the user how and how many times the elements in the reusable aspect were intended to be applied.

7 Conclusion

In this paper we have presented *instantiation cardinalities*, a new 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 a) 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 b) 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 model users face with most aspect-oriented approaches 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 when it is applied. As a result, the designer of the reusable aspect is able to specify all the instantiation policies identified in [16].

In order to allow for true separation of concerns, we have proposed to extend the TouchRAM weaver with automated call forwarding. As a result, it is possible to maintain polymorphic treatment of a set of subclasses in one aspect, while not requiring uniform naming of polymorphically related operations in each individual subclass.

For illustration purpose, the paper 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 a behavioural design pattern (*Observer*), a structural design pattern (*Composite*) and a creational design pattern (*Abstract Factory*).

References

1. Al Abed, W., Bonnet, V., Schöttle, M., Yildirim, E., Alam, O., Kienzle, J.: TouchRAM: a multitouch-enabled tool for aspect-oriented software design. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 275–285. Springer, Heidelberg (2013)
2. Al Abed, W., Kienzle, J.: Information hiding and aspect-oriented modeling. In: 14th Aspect-Oriented Modeling Workshop, Denver, CO, USA, Oct. 4th, 2009, pp. 1–6, October 2009
3. Alam, O., Kienzle, J., Mussbacher, G.: Concern-oriented software design. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 604–621. Springer, Heidelberg (2013)
4. Axelsen, E.W., Sørensen, F., Krogdahl, S., Møller-Pedersen, B.: Challenges in the design of the package template mechanism. T. Aspect-Oriented Software Development **7271**, 268–305 (2012)
5. Barreiros, J., Moreira, A.: Reusable model slices. In: 14th Aspect-Oriented Modeling Workshop, Denver, CO, USA, Oct. 4th, 2009, October 2009
6. Carton, A., Driver, C., Jackson, A., Clarke, S.: Model-driven theme/UML. In: Katz, S., Ossher, H., France, R., Jézéquel, J.-M. (eds.) Transactions on Aspect-Oriented Software Development VI. LNCS, vol. 5560, pp. 238–266. Springer, Heidelberg (2009)
7. Cottenier, T., van den Berg, A., Elrad, T.: Motorola WEAVR: Aspect and model-driven engineering. Journal of Object Technology **6**(7), 51–88 (2007). http://dx.doi.org/10.5381/jot.2007.6.7.a3
8. Cottenier, T., Berg, A.V.D., Elrad, T.: The motorola weavr: model weaving in a large industrial context. In: AOSD 2006 Industry Track. ACM, March 2006
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison Wesley, Reading, MA (1995)

10. Hölzl, M., Knapp, A., Zhang, G.: Modeling the car crash crisis management system using HiLA. In: Katz, S., Mezini, M., Kienzle, J. (eds.) Transactions on Aspect-Oriented Software Development VII. LNCS, vol. 6210, pp. 234–271. Springer, Heidelberg (2010)
11. International Telecommunication Union (ITU-T): Recommendation Z.151 (10/12): User Requirements Notation (URN) - Language Definition, October 2012
12. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001)
14. Kienzle, J., Al Abed, W., Klein, J.: Aspect-Oriented Multi-View Modeling. In: AOSD 2009. pp. 87–98. ACM Press, March 2009
15. Kramer, M.E., Klein, J., Steel, J.R.H., Morin, B., Kienzle, J., Barais, O., Jézéquel, J.-M.: Achieving practical genericity in model weaving through extensibility. In: Duddy, K., Kappel, G. (eds.) ICMB 2013. LNCS, vol. 7909, pp. 108–124. Springer, Heidelberg (2013)
16. Morin, B., Klein, J., Kienzle, J., Jézéquel, J.-M.: Flexible model element introduction policies for aspect-oriented modeling. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part II. LNCS, vol. 6395, pp. 63–77. Springer, Heidelberg (2010)
17. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the Association of Computing Machinery **15**(12), 1053–1058 (1972)
18. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York Inc, Secaucus, NJ, USA (2005)
19. Reddy, Y.R., Ghosh, S., France, R.B., Straw, G., Bieman, J.M., McEachen, N., Song, E., Georg, G.: Directives for composing aspect-oriented design class models. In: Rashid, A., Akşit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 75–105. Springer, Heidelberg (2006)
20. Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A., Araújo, J.: MATA: a unified approach for composing UML aspect models based on graph transformation. In: Katz, S., Ossher, H., France, R., Jézéquel, J.-M. (eds.) Transactions on Aspect-Oriented Software Development VI. LNCS, vol. 5560, pp. 191–237. Springer, Heidelberg (2009)