Taming Aspects with Managed Data

Theologos A. Zacharopoulos
theol.zacharopoulos@cwi.nl

Summer 2016, 53 pages

Supervisor: Tijs van der Storm
Host organisation: Centrum Wiskunde & Informatica, http://www.cwi.nl

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

Chapter 1

Introduction

Cross Cutting Concerns (CCC) is a problem the classic programming techniques can not tackle with sufficiently. This results in scattered and tangled code, which affects the system's modularity and it's ease of maintenance and evolution. Since Object Oriented Programming (OOP) and Procedural Programming (PP) techniques can not solve this problem, Aspect Oriented Programming (AOP) was introduced [KLM+97] in order to provide a solution to the problem, by presenting the notion of *aspects*.

AOP results in a modular and *single-responsibility* based design, whose properties must be implemented as *components* (cleanly encapsulated procedure) and *aspects* (not clearly encapsulated procedure), both separate concepts that are combined for the result through an automated process called *weaving*. However, relying on AOP, paradoxically, does not improve the evolution of a project even with the modularity that it provides. The reason is that it introduces tight coupling between the aspects and the application. As a result, the way to address this problem is to consider of a new sophisticated and expressing crosscut language. CCC could be handled on a higher level of the language such as the data structuring and management mechanisms.

Managed data [LvdSC12] allows the developers to take control of important aspects of data as reusable modules. Using managed data a developer can build *data managers* that handle the fundamental data manipulation primitives that are usually hard-coded in the programming language, by introducing custom data manipulation mechanisms. Managed data have been researched and implemented under the Enso project[1], which is developed in Ruby[2] (a dynamic programming language) using the meta-programming framework of Ruby. Furthermore, it is considered[LvdSC12] that managed data cannot be fully supported in static languages directly, which makes it more challenging for this thesis since it is implemented in Java. In this thesis we use the Java reflection API in order to implement managed data and focus on specific aspects and design patterns implementations using the data managers concept.

Finally, in order to evaluate the implementation of aspects and how we deal with CCC in managed data, we have reimplemented a part of a well-known use case, the JHotDraw, and evaluated the results on a number of explicit criteria.

1.1 Initial Study

In their study on managed data, Cook et al. [LvdSC12] present the main idea of managed data, while using a show case of it in an implementation in Ruby. As a use case they present the Enso project in order to reuse database management and access control mechanisms across different data definitions.

This thesis is an extension of their work; we implement managed data in Java (a static programming language) using the Java reflection API[3] and dynamic proxies[4]. Although proxies in static programming languages can not implement the full range of managed data [LvdSC12]. Java provides a strong

[1] http://enso-lang.org/
[2] https://www.ruby-lang.org/en/
[3] https://docs.oracle.com/javase/tutorial/reflect/
[4] https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html

implementation of the Meta-Object Protocol (MPO) [KDRB91], which can be used though the Java Reflection API [FFI04]. Additionally, our work focuses on the aspects perspective and it provides a solution to the CCC problem by using managed data and their data managers.

The most famous implementation of AOP is the one provided by Kiczales et al. called AspectJ [KHH+01]. Although AspectJ has been used by a number of projects, some of them with significant research results [HK02] [MM], it includes all the trade-offs of AOP, which are presented in detail in Section 1.2. In this thesis we show how we use managed data in order to tame aspects and compare the results with an AspectJ show case, the AJHotDraw.

1.2 Problem statement

1.2.1 Problem Analysis

Predefined data structuring mechanisms

One of the most important characteristics of programming languages is the data structures definition. Different types of data structures can be found on different languages and paradigms including *structures*, *objects*, *predefined data structures*, *abstract data types* and more. The common characteristic of these definitions is that they are all predefined. Thus, they do not allow the developers to take control on the data structuring and management mechanisms, but only to create data of these types [LvdSC12].

The problem with this approach is that the predefined data structuring mechanisms can not implement Cross Cutting Concerns and other "common requirements" for data management. In particular, those requirements are not properties that belong to a data structure definition, since, although it is easy to define them individually for every data type, that introduces a significant amount of duplicated and scattered code through the program.

Consequently, in this thesis we implement managed data, which gives the programmers control over the data structuring mechanisms.

Crosscutting concerns

As it has be seen [HMK05] there are a number of concerns during software implementation, that a developer has to work with. For good software modularity, these concerns have to be implemented on different modules, each of these modules implement only one concern. However, some of these concerns can not fit to separate modules but their implementation cuts across the system's modules. Those concerns are called Cross Cutting Concerns and result to the problem of *scattered* and *tangled* code.

The problem we study focuses on the CCC that are scattered around the application, resulting in a hard to maintain system by tangling implementation logic and concerns code together. In order to deal with this problem a refactoring of those concerns has to take place, in which the tangled and scattered implementation has to be replaced with an equivalent *aspect* [HMK05].

In this thesis we focus on the modularization of such CCC in aspects, using managed data. We refactor those concerns in modular data structures each of which implement only one concern by lifting the data management up to the application level and allowing the developers to define the concerns in their own data structures.

Aspect Oriented Programming problems

Even though, AOP provides new modularization mechanisms, which should result in easier evolving software, it delivers solutions that are as hard and sometimes even harder to evolve than before [TBG03]. The problem lays on the aspects, which have to include a crosscut description of all places in the application where this code yields an influence. Thus, the aspects are tightly coupled to the application and this greatly affects the evolvability of the overall system.

Additionally, Steimann [Ste05] argues that modeling languages are not aspect ready. The problem that arises is located at the level of software modeling. More specifically, whereas in OOP roles are

tied to the collaborations, in *roles modeling* collaborations rely on interactions of objects and aspects are typically defined independently of one another.

Furthermore, in terms of order, it has been observed that aspects are not elements of the domain, they rather describe the order than the domain. Finally, aspects invariably express non-functional requirements, but if the non-functional requirements are not elements of domain models then neither are aspects.

1.2.2 Research Questions

Managed data has not been practically implemented in a static language before, therefore our first research questions states *"Can managed data be implemented in a static language?"*. Based on the previous argumentation about the relevance of AOP and the solutions that managed data can provide in Cross Cutting Concerns, our second research question is *"Can managed data solve the problem of crosscutting concerns?"*. Finally by using a software showcase, the JHotDraw framework, as well as its AOP implementation AJHotDraw [MM], we evaluate the implementation of managed data on an inventory of aspects and design patterns. As a result the third research question states *"To what extent can managed data tame an inventory of aspects and design patterns in the JHotDraw framework, compared to the original and the AOP implementation?"*.

1.2.3 Solution Outline

Our solution consists of an implementation of managed data in Java. In particular, we have implemented a framework that can be used in order to create managed data in Java. This framework provides all the mechanisms of managed data using Java reflection and dynamic proxies. Additionally, one can use the framework in order to refactor the CCC of an application.

As it has been already mentioned, to validate our hypotheses we have implemented managed data in Java using the Java Reflection API and Dynamic Proxies. More specifically we define *schemas* using Java interfaces and dynamic proxies for the *data managers*. Furthermore, we provide a proof of concept the examples given in [LvdSC12] but this time developed in Java using our framework. To stack data managers[LvdSC12], we use the *Decorator Pattern* [Gam95].

In order to see if managed data solves the problems that AOP introduces, we have implemented an inventory of the following aspects and design patterns from JHotDraw using data managers:

The Observer Pattern, which as presented in literature [TBG03] [HMK05] [MMvD05], is by nature not modularized and the scatters pattern code through the classes. This pattern is considered as a difficult case because it is used a lot in the original JHotDraw source code but with multiple variations, thus it is difficult to extract an abstract version.

The Undo aspect, which is analyzed extensively [Mar04] and a solution is provided by AJHotDraw. More specifically, this aspect consists of aspect-oriented refactoring of the *Command* pattern with *Undo* actions.

This inventory is implemented using data managers that have modularity as a main characteristic and is evaluated in a new JHotDraw implementation. We compared those aspects with the original version of JHotDraw, and the aspect version, AJHotDraw. Since our solution is a refactoring of the JHotDraw framework we needed a way to ensure the behavioral equivalence between the original and the refactored solution [Fow09]. To archive that, we used JHotDraw test suite.

1.2.4 Research Method

In order to answer our research questions we studied the theoretical background, we examined our managed data implementation in Java and we evaluated our implementation in an existing use case system, the JHotDraw.

Managed data implementation in a static programming language. In order to answer the question if managed data could be implemented in a static language, we've implemented man-

aged data in Java using Java's reflection capabilities[5], Java interfaces for schemas definition and dynamic proxies[6] for the data managers. An extensive presentation of the implementation is given in Chapter 4.

Use case implementation. In order to argue about the contribution of our implementation and managed data for aspects handling in general, we've used a use case application (JHotDraw) which is considered as a good design use case for OOP, along with it's AOP implementation (AJHotDraw). Thus, we have built our version of the JHotDraw application using our managed data framework to refactor the CCC.

Use case evaluation. In order to show if our managed data solved the issues of AOP in terms of modularity, we have gathered a number of metrics for each of the three implementations the results are presented extensively in Chapter 6.

1.3 Contributions

Contribution 1: Managed data implementation in Java. Our first contribution is the implementation of managed data in a static language, in our case we chose Java. The reason we chose Java as the programming language is because Java is a very popular, static, object oriented programming language, with meta-programming (reflective) capabilities which we took advantage of. Managed data implemented as an internal Domain Specific Language (DSL) in Java, using interfaces for schema definitions and dynamic proxies for the data managers.

Contribution 2: Managed data Java framework. The final deliverable is a Java library, which the developer can use to define managed data and data managers for them. Additionally, the developer can define and implement aspects as reusable modules and introduce them in an application without mixing the business logic with the concern logic. More specifically, the schemas and the data managers have to be defined by the developer, as well as any additional functionality that needs to be integrated to the patterns or roles of the application.

Contribution 3: Managed data Evaluation in JHotDraw. We implemented a new version of the JHotDraw application using our framework in order to evaluate our refactoring of CCC. More specifically, we focused on the *Undo* concern, which is a *Command Pattern* and it is scattered around the modules of the JHotDraw, as well as the *Observer Pattern* which has been used in multiple parts of JHotDraw and cuts "pattern code" on different modules.

Contribution 4: JHotDraw implementation results assessment and comparison with AJHotDraw. Finally, we present the results of our evaluation and we compare them with AJHotDraw which implements AOP using the AspectJ language, again in Java.

1.4 Related Work

In this section we discuss the related work of research that inspired this thesis. In particular, we discuss points that we followed and points that we've tried to improve as well as the reason of doing it.

Meta-Object Protocol

Managed data can be implemented using reflection and the MPO. The authors of Enso [LvdSC12] implemented it in Ruby using the meta-programming framework and in particular, the **method_missing** mechanism. In other languages (such as Java, JavaScript or C#) that support dynamic proxies, they can be used for the managed data implementation, which is the way we've implemented it. The MPO [KDRB91] was first implemented for simple OOP capabilities of the Lisp language in order to satisfy some developer demands including compatibility, extensibility and developers

[5] https://docs.oracle.com/javase/tutorial/reflect/
[6] https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html

experimentation. The idea was that the languages have been designed to be viewed as black box abstractions without giving the programmers the control over semantics or the implementation of those abstractions. MPO opens up those abstractions to the programmer so he can adjust aspects of the implementation strategy. Providing an open implementation can be advantageous in a wide range of high-level languages and thus MPO technology is a powerful tool for providing that power to the programmer [KDRB91]. Furthermore, MPO provides flexibility to the programmer because as a language becomes more and more high level and it's expressive power becomes more and more focused, the ability to cleanly integrate something outside the language's scope becomes more and more difficult. Thus, both MPO and managed data allow the programmer to be able to control the interpretation of structure and behavior in a program. However, MPO focuses on behavior of the objects and classes, while in managed data the focus rests solely on the data management. One could conclude that managed data is a subset of the MPO approach since managed data have a more narrow scope.

Adaptive Object Model

Managed data [LvdSC12] is closely related to the Adaptive Object Model (AOM). AOM [YJ02] is an architectural style that emphasizes flexibility and runtime dynamic configuration. Architectures that are designed to adapt at runtime to new user requirements by retrieving descriptive information that can be interpreted at runtime, are sometimes called a "reflective architecture" or a "meta architecture". An AOM system, is a system that represents classes, attributes, relationships, and behavior as metadata, something that is closely related to the managed data. However, on one hand AOM style is more general than the managed data since it is described at a very high level as a pattern language and it covers business rules and user interfaces, in addition to data management. On the other hand, AOM does not discuss issues of integration with programming languages, the representation of data schemas, or bootstrapping, which are central characteristics of managed data. AOM is also presented as a technique for implementing business systems, not as a general programming or data abstraction technique [LvdSC12].

Model Driven Software Development

Model Driven Software Development (MDSD) refers to a software development method which generates code from defined models. The models represent abstract data that consist of the structure and properties definition of an entity. The idea of the model in MDSD is closely related to the *schemas* in managed data. Similarly to the model definition, schemas define the structure, the properties and any metadata that describe an entity, followed by code generation that adds any extra functionality and manipulation mechanisms to that entity.

The Enso Language

Enso project[7] is the first implementation of managed data, it is open source[8] and is used for EnsoWeb, a web framework written with managed data. Although Enso is implemented in Ruby, which is a dynamic language, the source code was a very helpful resource for our static implementation in Java. The design of Enso was an inspiration for our implementation even though some parts have changed completely in order to conform to our needs and support Java's static type system. Additionally, examples presented in Enso, are also implemented in our case and are presented in Chapter 3.

Aspect Oriented Programming

Although AOP is not directly connected to managed data, it is a mechanism that is relatively easy to be supported in managed data. This mechanism consists of the *weaving* of aspect code in specific join points. The way to support this in managed data is through data managers. How to tame aspects in managed data is the main topic of this thesis and is going to be presented extensively in the following chapters.

[7] http://enso-lang.org/
[8] https://github.com/enso-lang/enso

1.5 Document Outline

In this section we outline the structure of this thesis. In Chapter 2 we introduce the background, focusing on the concepts, which the reader must be familiar with in order to follow the next chapters. In Chapter 3 we demonstrate an example to show the capabilities of our implementation. In Chapter 4 the implementation of managed data in Java is presented and discussed, providing detailed explanation of our issues and implementation details. Next, in Chapter 5 a showcase is presented, by applying our implementation to refactor aspects in JHotDraw. In Chapter 6 an evaluation of the aspect refactoring is illustrated. Additionally, some metrics, claims and results are presented. Finally, a conclusion is given in Chapter 7 followed by further work in Chapter 8.

Chapter 2

Background

2.1 Cross Cutting Concerns

There is significant research in software engineering that focuses on the importance of software modularity. The most significant, among the many, advantage of modular systems is the extensibility and evolution of a system [Par72].

However, during the development of a system there is a number of concerns that have to be considered and implemented into the system. In order to follow the modularity principles, those concerns have to be implemented in separate modules, this way the program will be extensible and its evolution will be easier. Nonetheless, many of those concerns can not fit into the existing modular mechanisms in any of the existing programming paradigms including both Object Oriented Programming and Procedural Programming. In those cases, the concerns are scattered through the modules of the system, resulting to scattered and tangled code. Those concerns are called Cross Cutting Concerns [HMK05]. CCC are considered a serious issue for the evolution of a system and their effects of tangled and scattered code can be disastrous for a system's extensibility.

The reason is that the code which is related to a concern is scattered in multiple modules, while the concern code is tangled with the each module's logic resulting in a system, which does not follow the *Single Responsibility Principle* and consequently is hard to maintain.

Among many examples of those CCC are persistence, caching, logging, error handling [LL00] and access control. Additionally, some design patterns scatter "design pattern code" through the application, for instance the *Observer Pattern*, *Template Pattern*, *Command Pattern* etc. [HK02] [Mar04].

In order to solve this problem we need a way to refactor and transform the non-modularized CCC into a modular *aspect*. In other words, refactorings of CCC should replace all the scattered and tangled code of a concern with an equivalent module [HMK05], which in AOP terminology is called *aspect* [KLM+97].

2.2 Aspect Oriented Programming

Kiczales et al. present AOP [KLM+97] by using an example of a very simple image processing application. In that system, as in every system, whenever two properties which are being programmed must compose different tasks and yet be coordinated (in the example filters and loop-fusion), they **crosscut** each other. Because general purposes languages provide only one composition mechanism, which leads to complexity and tangling, the programmer must do the co-composition manually. According to their theory, a system's property can be either a *component*, if it can be clearly encapsulated in a generalized procedure, otherwise it is an *aspect*. AOP supports the programmer in separating components and aspects from each other by providing mechanisms that make it possible to abstract and compose them together when producing the whole system (*aspect weaving*). Alternatively to the common programming paradigms, OOP and PP that allow programmers to only separate the *components* from each other but crosscut the concerns.

However, implementing AOP programs is not that easy since several tools are needed in order to

succeed. More specifically, a general purpose language needs only a language, a *compiler* and a *program* written in the language that implements the application. In the case of an AOP based implementation, a program consists of a *component language*, in which the components are programmed, one or more *aspect languages*, in which the aspects are programmed, an aspect *weaver* for the combined languages, a *component program* that implements the components using the component language and finally, one or more *aspect programs* that implement the aspects using the aspect languages. Additionally, an essential function of the aspect weaver is the concept of *join points*, namely the elements of the component language semantics that the aspect programs coordinate with.

2.2.1 AspectJ

There is significant work in the area of aspect oriented languages but one the most important contribution is the AspectJ[1] project, which is a simple and practical aspect-oriented extension to Java and it has been introduced by Kiczales et al. [KHH+01]. The authors of AspectJ provide examples of programs developed in AspectJ and show that by using it, CCC can be implemented in clear form, which in other general purpose languages would lead to tangled and scattered code. AspectJ was developed as a compatible extension to Java so that it would facilitate adoption by current Java programmers. The compatibility lays on upward compatibility, platform compatibility, tool compatibility and programmer compatibility. One of the most important characteristics of AspectJ is that it is not a DSL but a general purpose language that uses Java's static type system. Our goal is to apply those properties for our managed data implementation.

2.2.2 Design Patterns in Aspect Oriented Programming

Hannemann et al. present a showcase of AOP [HK02] in which they conduct an aspect-oriented implementation of the Gang of Four design patterns [Gam95] in AspectJ, in which 17 out of 23 cases show modularity improvements. Even though design patterns offer flexible solutions to common software problems, those patterns involve crosscutting structures between roles and classes / objects. There are several problems that the OOP design patterns introduce in respect to CCC, specifically in cases when one object plays multiple roles, many objects play one role, or an object play roles in multiple patterns [Sul02] (design pattern composition).

The problem lays on the way a design pattern influences the structure of the system and its implementation. Pattern implementations are often binded to the instance of use resulting in their scattering into the code and losing their modularity [HK02].

Even worse, in case of multiple patterns used in a system (pattern overlay and pattern composition), it can become difficult to trace particular instances of a design pattern. Composition creates large clusters of mutually dependent classes[Sul02] and some design patterns explicitly use other patterns in their solution.

Observer pattern in Aspect Oriented Programming

Hannemann et al. [HK02] provide some example implementations of several design patterns, including the *Observer Pattern* in which they focus on a detailed implementation. As they mention, in an observer pattern implementation, both the *Subject* and the *Observer* have to know about their roles and have "pattern related code" in them. As a result, adding or removing code from a class requires changes of code in that class.

In their implementation [HK02] of the observer pattern in AspectJ, they separate abstract aspects for:

- The *Subject* and *Observer* roles in classes.
- Maintenance of a mapping from *Subjects* to *Observers*.
- The trigger of *Subjects* that update *Observers*.

[1] https://eclipse.org/aspectj/

And concrete aspects for each instance of the pattern fills in the specific parts:

- Which classes can be *Subjects* and which *Observers*.
- A set of changes on the *Subjects* that triggers the *Observers*.
- The specific means of updating each kind of *Observer* when the update logic requires it.

Modularity Properties

The modularity properties implemented in this thesis are[HK02]:

Locality: All the code that implements the observer pattern is in the abstract and concrete observer aspects, none of it is in the participant classes. The participant classes are entirely free of the pattern context and as a consequence there is no coupling between the participants.

Reusability: The core pattern code is abstracted and reusable. The abstract aspect can be reused and shared across multiple observer pattern instances.

Composition transparency: Because a pattern participants implementation is not coupled to the pattern, if a *Subject* or *Observer* takes part in multiple observing relationships their code does not become more complicated and the pattern instances are not confused.

(Un)pluggability: Because *Subjects* and *Observers* don't need to be aware of their role in any pattern instance, it is possible to switch between using a pattern and not using it in the system.

In general an object or a class that is not coupled to its role in a pattern can be used in different contexts without modifications and the reusability of participants can be increased. The locality also means that existing classes can be incorporated into a pattern instance without the need to adapt them with extra effort. All the changes are made in the pattern instance. This makes the pattern implementations themselves relatively (un)pluggable. If we can reuse generalized pattern code and localize the code for a particular pattern instance, this can result in multiple instances of the same pattern in one application, making it easy to understand (composition transparency). This solves the common problem of having multiple instances of a design pattern in one application.

2.2.3 Evolvability issues

Since modularization and separation of concerns make the evolution of an application a lot easier and AOP provides mechanisms for modularization and system decomposition, aspect-oriented programs should be easier to be evolved and maintained. Paradoxically, this is not the case [TBG03] since AOP technologies deliver applications that are as hard, and sometimes even harder, as non-AOP.

According to Tourwe et al. [TBG03] the problem is that aspects have to include a crosscut description of all places in the application. Thus, it is much harder to make such crosscuts oblivious to the application and most importantly to the rest of the modules. Additionally, current means for specifying concerns rely heavily in the existing structure of the application, therefore the aspects are tightly coupled to the application (and its structure) and of course this affects negatively the evolvability of a system since it makes it hard to change it's structural. As Tourwe [TBG03] proposes, a solution for this problem would be the creation of a new, more sophisticated crosscut language. A language that enables the developer to discriminate between methods based on what they actually do instead of what they look like, in a more intentional way. This new language that implements CCC in a modular way is something try to realize in our thesis.

2.3 Managed Data

Managed data [LvdSC12] is a data abstraction mechanism that allows the programmer to control the definition of the data and their manipulation mechanisms. Additionally it provides a modular way to control aspects of data. Managed data helps the programmer by giving them control over

the structuring mechanisms, which until now were predefined by the programming languages. The developers could not take control of them, they could only create data of those types. Managed data provides significant flexibility since it lifts data management up to the application level, by allowing the programmer to build data managers that handle the fundamental data manipulation primitives, normally hard-coded into the programming language.

Managed data has three essential components:

Data description language, that describes the desired structure and properties of data.

Data managers, that enable creation and manipulation of instances of data.

Integration, with a programming language to allow data to be created and manipulated.

In the traditional approach, the programming language includes data definition mechanisms and their processes, which are both predefined. However, with managed data, the data structuring mechanisms are defined by the programmer by interpretation of data definitions. Of course, since a data definition model is also data, it requires a meta-definition mechanism.

2.3.1 Schemas

The schemas are the way to describe the structure of the data to be managed. They can be just a simple data description language which programmers can use to describe simple kind of data. For example Cook et al. [LvdSC12] used Ruby hash for the data description on a simple example where the hash was an object that represented a mapping from values to values. However, a simple schema format like this can not be used to describe itself because it is not a record.

A self-describing schema would allow schemas to be managed data themselves. We therefore need a self-describing schema that can be used to describe schemas, namely the "Schema-Schema". A Schema-Schema is also managed data with its own data manager. This process is called *Bootstrapping* and it is needed in order to "jumpstart" the process. This extends the benefits of programmable data structuring to their own implementation. Schemas can be interpreted in many different ways to create different kinds of records based on the manipulation provided by the data managers.

2.3.2 Data Managers

Data managers are the mechanisms that interpret *schemas* with defined manipulation strategies set by the programmers. The input to a data manager is a schema, which describes the structure of the data to be managed. Since the schema is only known dynamically, the data managers must be able to determine the fields and methods of the managed data object dynamically as well. In order to implement such an operation we need a meta-programming mechanism that dynamically analyses the structure of a schema and applies to it the functionality of the data managers. In their implementation Cook et al. [LvdSC12] used the "missing_method" implementation in order to succeed that. In out case we will use Java's reflection API and dynamic proxies.

2.4 Java Reflection and Dynamic Proxies

The Java programming language provides the programmer with a Reflection API[2] that offers the ability to examine or modify the runtime behavior of applications running in the Java Virtual Machine (JVM). Additionally, Java comes with an implementation of Dynamic Proxies[3] which is a class that implements a list of interfaces specified at runtime.

2.4.1 Reflection

Reflection is the ability of a running program to examine itself and its environment and to change what it does depending on what it finds [FFI04].

[2] https://docs.oracle.com/javase/tutorial/reflect/

[3] https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html

In order for this self-examination to be successful, the program needs to have a representation of itself, which is called *metadata*. In a OOP language this metadata is organized into objects, hence they are called *metaobjects*. Finally, the process of the runtime self-examination of these metaobjects is called *introspection*.

Java supports reflection with its reflection API since the version 1.1. Java provides many operations for using metaobjects performing intercession. Using Java reflection a running program can learn a lot about itself. This information may derive from classes (the `Class` metaobject), class name, class methods, a class super and sub classes, methods (the `Method` metaobject), method name, method parameters, method type, variables, variables handlers and more. Querying information from these metaobjects is called introspection. Additionally to the examining of the these metaobjects, a developer has the ability to dynamically call a method that is discovered at runtime. This process is called *dynamic invocation*. Using dynamic invocation, a `Method` metaobject can be instructed to invoke the method that it represents during the program's runtime.

Although reflection is considered helpful for developing flexible software, it has some known pitfalls:

Security. Since metaobjects give a developer the ability to invoke and change underlined data of the program, it also gives access them to places that are supposed to be secure (e.g. `private` variables).

Code complexity. Consider a program that uses both normal objects and metaobjects. That introduces an extra level of complexity since now a developer has to deal with different kinds of objects on different levels, meta and normal level.

Runtime performance. Of course the runtime dynamic examination and introspection introduce significant overhead on most language implementations. In the case of Java's dynamic proxies a 6.5x overhead observed [MSD15]. However, this is not something that we take into consideration in our implementation.

2.4.2 Dynamic Proxies

Since 1.3 version Java supports the concept of *Dynamic Proxies*. A *proxy* is an object that supports the interface of another object (*target*), so that the proxy can substitute for the target for all practical purposes[FFI04]. A proxy has to have the same interface as the *target* so that it can be used in exactly the same way. Additionally it *delegates* some or all of the calls that it receives to its target and thus acts as either an intermediary or a substitute object. As a result, a programmer has the capability to add behavior to objects dynamically. The Java reflection API contains a dynamic proxy-creation facility, in `java.lang.reflect.Proxy`.

There are several examples of dynamic proxies implementation in Java including implicit conformance, future invocations [PSH04], dynamic multi dispatch, design by contract or AOP [Eug06].

Proxy Objects

A proxy is an object which conforms to a set of interfaces, for which that proxy was created for. The corresponding proxy class extends the class `Proxy` and implements all its interfaces. Thus, conforming to all those interfaces, a proxy can be casted to any of them and any method defined in those interfaces can be invoked on the proxy object [Eug06].

Invocation Handlers

All the proxy objects have an associated object of type `InvocationHandler`, which handles the method invocations performed on the proxy, and its interface is shown in Listing 2.1.

The arguments of the `invoke` method include the object on which the method was originally invoked (i.e., the proxy), the method itself that was invoked on the proxy, and the arguments of that method, if any. Therefore, the `invoke` method is capable of handling any method invocation.

```
1 public interface InvocationHandler {
2   public Object invoke(Object proxy, Method method,Object[] args) throws Throwable;
3 }
```

Listing 2.1: The Invocation Handler Interface

Issues

A proxy instance is an object and it responds to the methods declared by `java.lang.Object`. Thus, when these methods should be invoked and from which object is an issue that arises [FFI04].

The methods `equals`, `hashCode`, and `toString` are inherited by all classes from the `Object` class and they are handled just like custom methods. If they are proxied then they are also overridden by the proxy classes and invocations to them are forwarded to the invocation handler of the proxy. Other methods defined in `Object` are not overridden by proxy classes, as they are `final` [Eug06].

Logging Example

Previously we mentioned the *logging* CCC 2.1. Image that every method invocation of an object has to be logged into the console, in that case we would need to write logging code on each of the methods of that class. This would lead to scattered logging code through the methods. This is problem can be solved with dynamic proxies and a simple solution is presented in Listing 2.2.

```java
import java.lang.reflect.*;
import java.io.PrintWriter;

public class TracingIH implements InvocationHandler {
  public static Object createProxy(Object obj, PrintWriter out) {
    return Proxy.newProxyInstance(
        obj.getClass().getClassLoader(),
        obj.getClass().getInterfaces(),
        new TracingIH( obj, out));
  }

  private Object target;
  private PrintWriter out;

  private TracingIH(Object obj, PrintWriter out) {
    target = obj;
    this.out = out;
  }

  public Object invoke(Object proxy, Method method,Object[] args) throws Throwable {
    Object result = null;

    try {
      out.println(method.getName() + "(...) called" );
      result = method.invoke( target, args );
    } catch (InvocationTargetException e) {
      out.println(method.getName() + " throws " + e.getCause());
      throw e.getCause();
    }
    out.println(method.getName() + " returns" );
    return result;
  }
}
```

Listing 2.2: An invocation handler for a proxy that traces calls [FFI04]

2.5 JHotDraw And AJHotDraw

JHotDraw[4] is a Java GUI framework for technical and structured graphics. It is an open-source, well-designed and flexible drawing framework of around 18,000 non-comment lines of Java code. JHotDraw's design relies heavily on some well-known design patterns[Gam95] and it is considered as a showcase for software quality techniques provided to the OOP community.

The fact that JHotDraw is appraised as a so well-designed application makes it an ideal candidate as a showcase for an aspect oriented migration. Marin and Moonen [MM] use this showcase for adoption of aspect-oriented techniques in existing systems. In particular, they present AJHotDraw[5], which is an aspect-oriented version of JHotDraw developed in Java and AspectJ. The goal of AJHotDraw is to take the existing JHotDraw and migrate it to a functionally equivalent aspect-oriented version.

The authors presented a fan-in analysis of JHotDraw [MVDM04] and implemented an idiom-driven approach to aspect-mining. This way they could extract a number of aspects in JHotDraw. Next, they perform a concern exploration in order to expand their mining results leading to concern sorts. Concern sorts is a consistent way to address crosscutting concerns in source code. That led to the

[4] http://www.jhotdraw.org/
[5] http://swerl.tudelft.nl/bin/view/AMR/AJHotDraw

identification and documentation of crosscutting concerns in JHotDraw, which helps to aware the developers of the existence of CCC in JHotDraw. To tame the aspects in a more consistent and formal way, Marin et al. defined a list of *template aspect* solution their concern sorts. Finally, they performed aspect refactoring of JHotDraw by presenting the AJHotDraw, which according to them, was the largest migration to aspects available to date. Their refactoring aimed at maintaining the conceptual integrity of the original design.

In order to refactor the existing framework, the first thing that AJHotDraw developers needed to do was to create a test subproject for the JHotDraw, which is called TestJHotDraw[6] in order to ensure behavioral equivalence between the original and the refactored solution. Refactoring implies preserving the observable behavior of an application[Fow09] and since the developers of AJHotDraw ought to test their functionality, TestJHotDraw was created. There are several contributions of the aspect-oriented implementation approach[MM]. The authors suggest that the project contributes to a gradual and safe adoption of aspect-oriented techniques in existing applications and allow for a better assessment of aspect orientation.

In this thesis we have used JHotDraw and AJHotDraw in order to evaluate our CCC refactoring in managed data. We could not use the TestJHotDraw since is written in AspectJ and we wanted to be separate from it. Thus we used the JHotDraw original test suite which includes a large number of cases, which include our refactoring. The detailed evaluation is described in Chapter 6.

2.5.1 Refactoring of Crosscutting Concerns

The refactoring of legacy code to aspect oriented code is also known as *Aspect Refactoring* [MMvD05]. During this process it is important to identify which elements are going to be refactored and which *aspect* solutions will replace them. To evaluate the refactored elements [Fow09], a testing component is needed in order to ensure behavior conservation, hence some coherent criteria to organize CCC are needed. Marin, Moonen and Deursen [MMvD05] organize the CCC into types, which are descriptions of similar concerns that share the following properties:

- A generic behavioral, design or policy requirement to describe the concern within a formalized, consistent context (e.g., role superimposition to modular units (classes), enforced consistent behavior, etc.),
- An associated legacy implementation idiom in a given (non-aspect oriented) language (e.g., interface implementations, method calls, etc.)
- An associated (desired) aspect language mechanism to support the modularization of the type's concerns (e.g., `pointcut` and `advice`, introduction, composition models).

2.5.2 The Observer Pattern

Design patterns introduce CCC by scattering "pattern code" through an application. Hannemann et al. [HMK05] discuss this with an example of CCC refactoring of the observer pattern. The observer pattern is by its nature not-modularized. The pattern crosscuts a large number of modules, tangling the scattered concern implementation with other code, making changes very complex.

Role-based Refactoring

The authors present a *role-based* refactoring, which consist of separating the roles of the pattern to different aspects. The role-based refactoring approach helps the developer to transform a scattered implementation of a CCC into an equivalent but modular AOP implementation. Both CCC and refactoring are described in terms of roles.

According to the authors[HMK05], the steps of role-based refactoring are the following:

Selecting a CCC refactoring: The refactoring includes an abstract description of the CCC it targets and a set of instructions to produce a modular AOP implementation of the refactoring (e.g. the Observer pattern CCC refactoring).

[6] `http://swerl.tudelft.nl/bin/view/AMR/TestJHotDraw`

Stating a mapping: Map role elements comprising the CCC description to the program elements of the scattered code (e.g. the Subject and the Observer role to concrete classes)

Planning the refactoring: make the right choices for specific cases since a CCC refactoring involves modifying several parts of a codebase (e.g. naming).

Execution: transform the code according to the refactoring instructions (e.g. modularizes Observer pattern as a result).

Thus, to refactor CCC it is required a mapping from the abstract CCC description to programming components that explicitly describe the CCC implementation. This mapping for the case of the observer pattern is presented by Hannemann et al. [HMK05] in Figure 2.1.

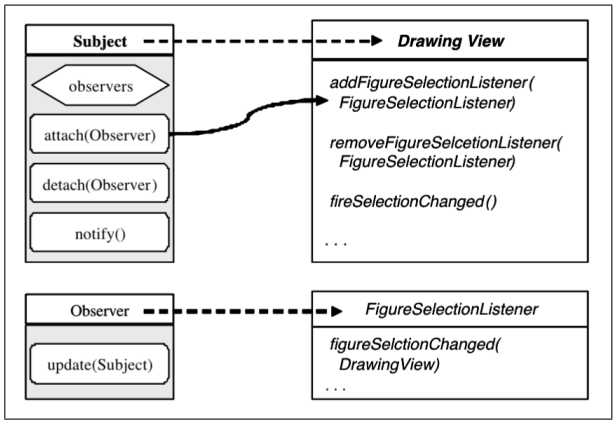

Figure 2.1: Observer pattern: Role Mapping [HMK05]

This figure describes the roles mapping in a specific case on JHotDraw, the *Figure Selection Listener*. However the authors have shown an abstract and reusable way of describing those roles.

2.5.3 The Figure Selection Observer of JHotDraw

As it can be seen from the previous paragraph, Hannemann et al. [HMK05] presented an approach for the *Observer* design pattern refactoring in JHotDraw. Likewise, during the AJHotDraw implementation[MMvD05] [HMK05], the authors propose a type-based refactoring on the same *Observer* instance, the `SelectionListener`.

The concern sorts that identified in this case are; the *Role Superimposition* which is similar to the role-based refactoring described previously and the *Consistent Behavior*, which describes a set of methods consistently invoke a specific action as a step in their execution.

The legacy code architecture of JHotDraw is displayed in Figure 2.2.

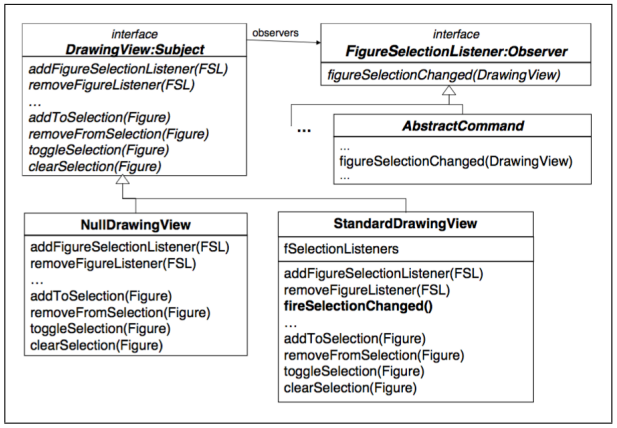

Figure 2.2: Observer pattern: Selection Listener [MMvD05]

The `FigureSelectionListener` has the *Observer* role in the pattern implementation. Any class that is interested in changes of the selection of figures in a `DrawingView` implements this interface. The `DrawingView` interface partially defines the *Subject* role by including two methods `addViewChangeListener` and `removeViewChangeListener`. From the classes that implement this interface only one, the `StandardDrawingView`, contains a non-empty implementation of the *Subject* role in the `fireSelectionChanged` method. Not that this method is only defined in the concrete class, which deviates from the standard Observer pattern implementation.

In their aspect refactoring they described an aspect that is constructed comprising both the *Subject* and *Observer* roles definition and maintaining a list of associations between each *Subject* and its *Observer* objects. Their type-based refactoring[MMvD05] distinguishes several crosscutting elements in JHotDraw's *Observer* pattern. First, the role superimposition, applied twice, for each of the two roles. Second, consistent behavior to notify the observers of the changes in the *Subject* object. Where superimposition is defined as the aspect implementation of a specific role and consistent behavior as the aspect implementation of a *consistent behavior* for a number of method elements that can be captured by a natural pointcut. The `GenericRole` (empty) interface documents the crosscutting type of role superimposition. Specific roles, like *Observer* and *Subject* (`SelectionSubject`) extend the interface. These elements are shown in Figure 2.3.

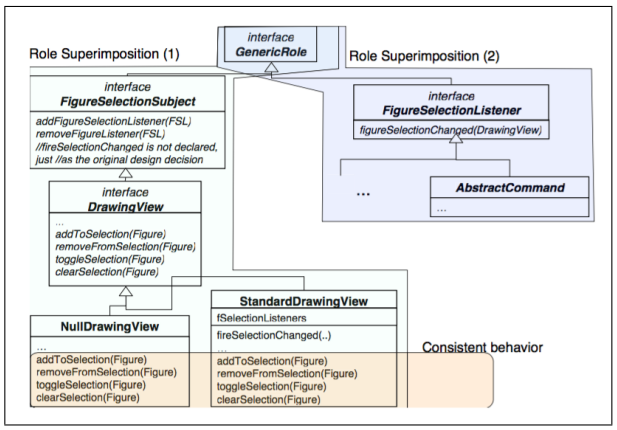

Figure 2.3: The concern types in Selection Listener [MMvD05]

2.5.4 The "Undo" Concern of JHotDraw

After a fan-in analysis of JHotDraw [MVDM04], Marin identified the "Undo" concern in JHotDraw and he presents an approach to its aspect-oriented refactoring [Mar04]. He uses the *(un)pluggability* property of a concern as an estimate of its refactoring cost. The author proposes the refactoring of

the "Undo" concern in JHotDraw using AspectJ with the implementation of AJHotDraw. During the fan-in analysis [MVDM04], the results have shown about 30 undo activities defined for various elements of JHotDraw. A representation of the elements in the JHotDraw implementation of the "Undo" concern can be seen in Figure 2.4.

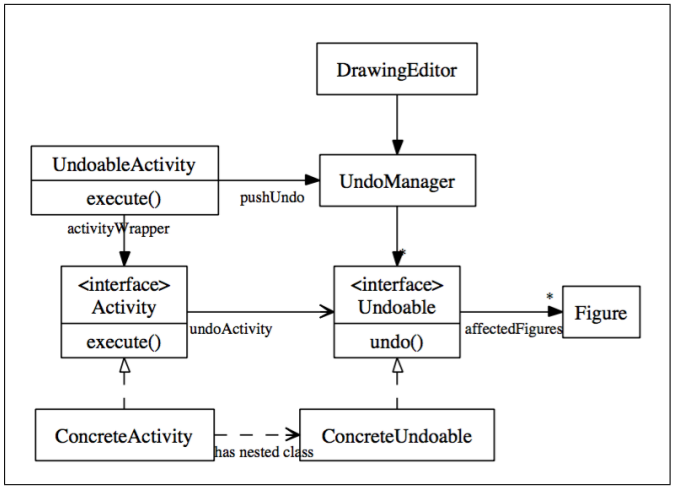

Figure 2.4: Participants for undo in JHotDraw [Mar04]

The `Activity` component participates in the implementation of the *Command* design pattern[Gam95]. Many of these activities have support for undo functionality, which in JHotDraw is implemented by means of nested (undo) classes. The nested class knows how to undo the given activity which maintains a list of affected figures whose state is also affected if the activity would be "undone". Whenever the activity modifies its state it also updates fields in its associated undo activity to actually perform the undo. The `Undoable Command` object serves three roles:

- it consumes the request to execute the command
- then, it delegates the command's execution to the wrapped command
- and last, acquires a reference to the undo activity associated with the wrapped command and it pushes it into a stack managed by an `UndoManager`. When executing an `Undo Command`, the top undo activity in the stack is extracted and, after the execution of its `undo()` method, is pushed into a redo stack managed by the same `UndoManager`.

Given this implementation, it is obvious that the primary decomposition of *Command* is crosscut by a number of elements as follows:

- The field by `AbstractCommand` for storing the reference to the associated `Undo Activity`.
- The accessors for this field implemented by the same class.
- The `UndoActivity` nested classes implemented by most of the concrete commands that support undo.
- The factory methods for the undo activities declared by each concrete command that can be undone.
- The references to the before enumerated elements from non-undo related members.

The implementation of AJHotDraw succeeded in refactoring this concern in JHotDraw through the following steps [Mar04]:

1. An undo-dedicated aspect is associated to each of undo-able command. The aspect will implement the entire undo functionality for the given command, while the undo code is removed from the command class.

2. Each aspect will consistently be named by appending `UndoActivity` to the name of its associated command class to enforce the relation between the two.

3. Next, the command's nested `UndoActivity` class moves to the aspect. The factory methods for the undo activities also move to the the aspect, from where are introduced back, into the associated command classes, using inter-type declarations.

4. Finally, the undo setup is attached to those methods from which was previously removed, namely execute() method, by means of an AspectJ `advice`.

This proposition [Mar04] provides an easy migration to an aspect-based solution. The CCC has been identified, then removed from the system, and finally re-added in an aspect-specific manner.

(Un)pluggability of the Undo concern

In order to evaluate the CCC refactoring of "Undo" Marin [Mar04] used the (Un)pluggability property. The author groups the complexity of the commands based two criteria. First, on the degree of *tangling* of the undo setup in the command's logic, particularly the activity's `execute()`method. Second, on the impact of removing the undo-related part from its original site, which can be estimated by the number of references to the factory method and to the methods of the nested undo activity. Thus, the (un)pluggability property gives a measure of how clearly the concern is distinguished in the original code and is a good estimate of the refactoring costs.

AspectJ Drawbacks and Limitations

By executing aspect-refactoring the two concerns are separated, modularized and the secondary concern of undo is no longer tangled into the implementation of the primary one.

However, there are certain drawbacks to this approach. For instance, AspectJ's mechanisms do not allow introduction of nested classes, the post-refactoring association will only be an indirect one, based on naming conventions ("`UndoActivity`"). This is a weaker connection than the one provided by the original solution. Another drawback that is observed is the change of the visibility of the methods introduced from the aspects, for example the inter-type declarations. The visibility declared in the aspect refers to the aspect and not to the target class. Such drawbacks could be overcome by a better aspect language support, which will be further discussed in this thesis.

Chapter 3

Example Application: State Machine Monitoring

In this chapter in order to show how our managed data implementation works in practice, and in particular in terms aspect refactoring, we present a showcase. The showcase consists of a very simple state machine application. Similar example is presented in Enso papers as a showcase for its Object Grammar capabilities [SCL+12].

Consider the requirements of the state machine are the following:

- A state `Machine` consists of a number of named `State` declarations.
- Each `State` contains `Transitions` to other states, which are identified by a `name`, when a certain event happens.
- A `Transition` is identified by a certain `event`.

For simplicity, this example will be a very basic *door state machine*, which includes three states **Open**, **Close** and **Locked** accompanied by their transitions: **open_door**, **close_door**, **lock_door** and **unlock_door** respectively. Figure 3.1 illustrates the door state machine.

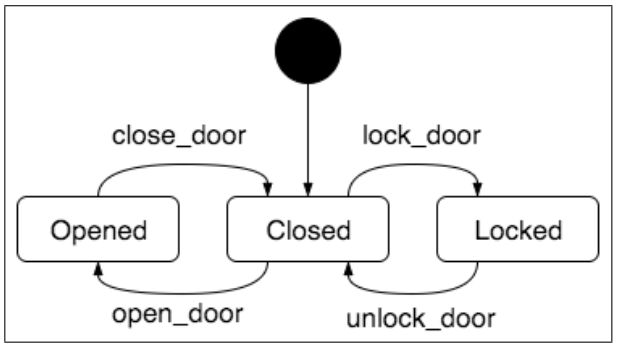

Figure 3.1: Basic door state machine

In order to implement this, first it is needed to define the models, next to interpret the definition given a list of event and finally add any additional functionality (*concern*) needed, for instance monitor the state of the door.

3.1 Schemas definition

First, we need to define all the models of the state machine program. An object diagram is illustrated in Figure 3.2.

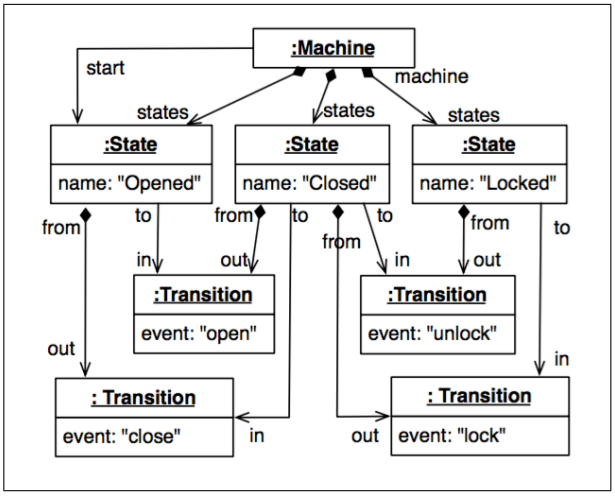

Figure 3.2: Basic door state machine object diagram

In our implementation we define schemas using Java interfaces with a set of meta-data described with Java annotations. Therefore, as extracted from the requirements we need `Machine` (Listing 3.3), `State` (Listing 3.4) and `Transition` (Listing 3.5) schemas.

```
1  public interface Machine extends M {
2    State start(State... startingState);
3
4    State current(State... currentState);
5
6    @Contain
7    Set<State> states(State... states);
8  }
```

Listing 3.3: The Machine Schema

As it can be seen in Listing 3.3, for the `Machine` schema definition we need a `starting` state, the `current` state of the machine and a set of `states` that the machine can be into at each time. Note that the `@Contain` annotation defines that the `states` field of this schema is part of the spine tree and it is not a cross-reference, this will be exampled in detail in Chapter 4.1.1.

```
1  public interface State extends M {
2    @Key
3    String name(String... name);
4
5    @Inverse(other = Machine.class, field = "states")
6    Machine machine(Machine... machine);
7
8    @Contain
9    Set<Transition> out(Transition... transition);
10
11   @Contain
12   Set<Transition> in(Transition... transition);
13 }
```

Listing 3.4: The State Schema

For the `State` definition, Listing 3.4, we need a `name` field, which is representing name of the state. The `name` field has been annotated with the `@Key` annotation, which indicates uniqueness, the states field of Machine can be indexed by name. Moreover, the schema includes a set of `in` and `out` `Transitions`. Since those two fields are of type `Set`, that means that a field of the `Transition` schema has to be marked as key. In this case is the `name` field (Line 2 Listing 3.5). Finally, the field `machine` represents which is the state machine that the current state part of. As it can be seen in the schema definition, Listing 3.4, the `machine` field has been annotated with `@Inverse`, which indicates that this field is a reference to a field of an other schema Klass. Thus, the `machine` field of `State` schema is a reference to `states` field of `Machine` schema.

```
1  public interface Transition extends M {
2    @Key
3    String event(String... event);
4
5    @Inverse(other = State.class, field = "out")
6    State from(State... from);
7
8    @Inverse(other = State.class, field = "in")
9    State to(State... to);
10 }
```

Listing 3.5: The Transition Schema

Finally, in the `Transition` schema definition, Listing 3.5, all we need is an `event` that represents the event of the transition which is also the **key** of that schema Klass. Additionally, the `from` and `to` fields represent the state that the machine changes from and to respectively. However, these are just reference to the `State` schema (Listing 3.4) to the `in` and `out` fields since they are defined with the `@Inverse` annotation.

3.2 Factory definition

Since we have our schemas we need a way to build instances of managed objects that these schemas describe. In Java to create the three schemas as managed data we need to define a factory, which creates managed data instances (managed objsects) for each of these schemas 3.6. Note that the method definitions work as `Constructors` of managed objects.

```
1  public interface StateMachineFactory {
2    Machine Machine();  // constructor for Machine managed objects
3    State State();      // constructor for State managed objects
4    Transition Transition(); // constructor for Transition managed objects
5  }
```

Listing 3.6: The StateMachine Factory

3.3 Basic Data Manager

As it is mentioned, in order to interpret and manage the defined data we need data managers. Our implementation includes the definition of a `Basic data manager` that is responsible of making a schema definition to an instance of *managed object*. However, in order to make the *managed object* it needs its schema definition (the interfaces that define the schemas) and the schema factory (the interface that defines the constructors of the schemas).

3.3.1 A simple without concerns program

In the case of a simple program without any concerns, we have to use our managed data to define the state machine and then interpret it. The definition of the door state machine is given in Listing 3.7 in Java.

In practice, the basic data manager needs to provide us with mechanisms that interprets the managed object that comply the `stateMachineSchema`, shown in Line 6. The basic data manager supports also the field accessors of those data, namely, set and get their values. A simple interpreter for the state machine is shown in Line 43. As it can be seen, the schemas factory is used to create managed objects. The *setup* of the fields is done automatically by the data manager who is responsible for the managed object interpretation.

3.4 Monitoring and notification concerns

Consider a case in which we want to add concerns at the previous door state machine implementation. A simple concern would be a *monitoring*, which would log every change in the current state of the state machine. Another concern would be the *notification*, which would be fire an action when a specific state is set to the machine.

Suppose that the system has to notify someone in case the door is opened. In the case that the door opens, the **Open** state will set to the current state of the machine. In that case, a notification has to be sent by e-mail. That looks similar to the *monitoring* concern; however, in this case the notification is a specific action: send an e-mail in case the door has opened.

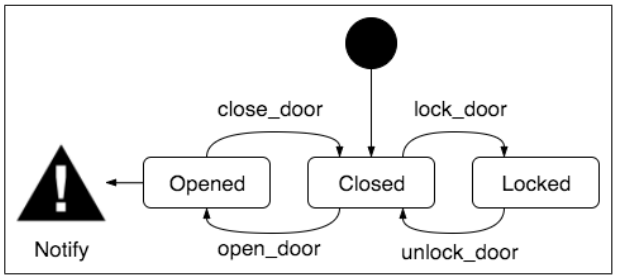

Figure 3.3: Simple door state machine: notify closed door

In order to implement those concerns we need a mechanism that continuously monitors the changes (transitions) of the machine's `current` state and react accordingly. In a traditional way this would lead to scattered *monitoring* and *notification* code in the interpretation method or the models themselves (the machine model). However, in managed data this is the role of the data managers. A data manager can implement concerns as modular aspects without crosscut code to the components. The programmer can define a manipulation mechanism of his/her data that includes an aspect of preference. Therefore, by implementing our concerns with data managers we can keep the component and aspect code separate.

3.4.1 Observable Data Manager

Towards to the *observation* of changes in the current state of our door state machine, we need a data manager that observes those changes in the managed object. In particular, the `Machine`'s current `State` field. This data manager creates concrete managed objects, namely *observable managed object*, with which one attach observers that will be notified in case of changes. It is important to mention that this new data manager has to inherit the basic one in order to include the basic functionality of schema interpretation and field access. This leads to a stack of two data managers each of which adds a new aspect of data in a modular way.

```
public class StateMachineExample {
  public static void main(String[] args) {
    Schema schemaSchema = ...;
    Schema stateMachineSchema = ....;
    BasicDataManager basicDataManager =
        new BasicDataManager(StateMachineFactory.class, stateMachineSchema);
    StateMachineFactory stateMachineFactory = basicDataManager.make();

    Machine doorStateMachine = stateMachineFactory.Machine();

    State openState = stateMachineFactory.State(OPEN_STATE);
    openState.machine(doorStateMachine);

    State closedState = stateMachineFactory.State(CLOSED_STATE);
    closedState.machine(doorStateMachine);

    State lockedState = stateMachineFactory.State(LOCKED_STATE);
    lockedState.machine(doorStateMachine);

    Transition closeTransition = stateMachineFactory.Transition(CLOSE_TRANSITION);
    closeTransition.from(openState); closeTransition.to(closedState);

    Transition openTransition = stateMachineFactory.Transition(OPEN_TRANSITION);
    openTransition.from(closedState); openTransition.to(openState);

    Transition lockTransition = stateMachineFactory.Transition(LOCK_TRANSITION);
    lockTransition.from(closedState); lockTransition.to(lockedState);

    Transition unlockTransition = stateMachineFactory.Transition(UNLOCK_TRANSITION);
    unlockTransition.from(lockedState); unlockTransition.to(closedState);

    doorStateMachine.start(closedState);

    interpretStateMachine(doorStateMachine, new LinkedList<>(Arrays.asList(
            LOCK_TRANSITION,
            UNLOCK_TRANSITION,
            OPEN_TRANSITION)));
  }
}

private static void interpretStateMachine(
    Machine stateMachine, List<String> commands)
{
    stateMachine.current(stateMachine.start());
    for (String event : commands) {
        for (Transition trans : stateMachine.current().out()) {
            if (trans.event().equals(event)) {
                stateMachine.current(trans.to());
                break;
            }
        }
    }
}
```

Listing 3.7: Door state machine

3.4.2 Monitor and notify concerns

In the example the *observers* are the concerns, which are *monitoring* and *notification* by e-mail in case of opened door. Accordingly, the `current` state of the state machine is the *subject* that informs the observers of its change. The definition of the concerns is given in Listing 3.8.

```
1  public class StateMachineMonitoring {
2      public static void monitor(Object obj, String field, Object value) {
3          if (field.equals("current")) {
4              logger.log(" > Current state changed to " + ((State) value).name());
5          }
6      }
7
8      public static void notify(Object obj, String field, Object value) {
9          if (field.equals("current") && ((State) value).name().equals(OPEN_STATE)) {
10             if (EmailSender.send("Danger!", "Someone just opened the door!")) {
11               logger.notify(" > Danger e-mail sent!.");
12             }
13         }
14     }
15 }
```

Listing 3.8: Door state machine concerns definition

Since there is an observable data manager and the concerns implemented in a separate and reusable module, completely unrelated to our logic code, it is still needed to be integrate them in the door state machine original code. The integration code is presented in Listing 3.9. The only part that it changes is the line 9 of the original code, in which the data manager of the `Machine` managed object has changed to the new observable data manager. Additionally, the concerns are attached to the machine object very easily, as can be seen in lines 12 and 13 of Listing 3.9.

```
1  ...
2  // State Machine monitoring
3  ObservableDataManager observableDataManager =
4        new ObservableDataManager(StateMachineFactory.class, stateMachineSchema);
5
6  StateMachineFactory observableStateMachineFactory = observableDataManager.make();
7
8  // Door State Machine definition, with observable data manager
9  Machine doorStateMachine = observableStateMachineFactory.Machine();
10
11 // Add monitoring and notification concerns
12 ((Observable) doorStateMachine).observe(StateMachineMonitoring::monitor);
13 ((Observable) doorStateMachine).observe(StateMachineMonitoring::notify);
14 ...
```

Listing 3.9: Door state machine with concerns

By running the program with the commands `LOCK_TRANSITION`, `UNLOCK_TRANSITION` and `OPEN_TRANSITION`, the output is presented in Listing 3.10.

```
> Current state changed to Closed
> Current state changed to Locked
> Current state changed to Open
> Danger e-mail sent!
```

Listing 3.10: Door state machine with concerns: output

The basic data manager allows to just build managed object but the observable data manager came in hand with the functionality of attaching concerns in the managed objects after an specified event. Concluding, that example just presented a modular solution of CCC without scattering and tangling code in the components.

Chapter 4

Managed data in Java

As it has been already mentioned, the programming languages include data definition mechanisms that are predefined. This fact makes them insufficient to define CCC without repeating and scatter code through the components [LvdSC12]. Notably, the problem is that CCC consider as features of the data management and not of the data types themselves. As a result, we implement managed data to allow the developer to define the mechanisms of data manipulation. This chapter describes our managed data implementation in Java, which consists of our first research question, which is *"Can managed data be implemented in a static language?"*. It is important to mention that our implementation is inspired by Enso[1], which is written in Ruby. Although Ruby is a dynamic language, Enso significantly contributed on our implementation's structure. In this chapter we preset the implementation of managed data in Java, which is available also online as an open-source project called JavaMD (Java Managed Data)[2].

4.1 Managed Data Implementation

Managed data provides the programmer to handle the fundamental data manipulation mechanisms using *Data Managers* and emphasizes strongly on modularity. Using a data description language the programmer can define *Schemas* that are the input of the *Data Managers*. The *Data Manager* in turn interprets the data description language that is used to define the structure and the behavior of the data to be managed. The *Schemas* and the *Data Managers* are the essential components of managed data along with the *Integration* with the programming language which in our case is Java.

4.1.1 Data description with Schemas

To create instances of data, we first need to define their structure. *Schemas* describe the outline structure of our data. In order to define *Schemas* in managed data we need a data description language that allows to define records as collections of fields. This language can be anything, e.g. XML, JSON or a different formalism like in Enso. For our implementation we chose to use **Java Interfaces** for data description language that allows to define records of managed data. By using Java interfaces we use Java's semantics for our definitions. Moreover, Java interfaces use several conventions to encode semantics, for instance Java annotations, which are very useful for meta data definition on *Schema*s.

Consequently, to define a *Schema* we first need to define a set of classes that describe that schema. A schema `Klass` [3] is described by a name and a set of `Field`s, each of which has a name and a `Type`. Since Java interfaces are used to define `schemaKlass` we need a way to define `Field`s for a `schemaKlass`. A `Field` in our data description language can be defined by using **Java's Method** definition.

[1] https://github.com/enso-lang/enso

[2] https://github.com/TheolZacharopoulos/JavaMD

[3] We use the "Klass" instead of "Class" convention in order to avoid any kind of ambiguities between Java's Class type and our type system. Klass is used to describe our own class type while Class describes Java's native class type.

Additionally, there are several attributes that help with the structure defining of a `Schema`, which considered as meta data. In order to define the meta data in our data description language (interfaces), we use *Java Annotations*. Annotations consist of a very declarative way to express meta data in interfaces and they are coupled structurally to the Java system. Thus, to annotated a field with meta data, we define annotations in a *Method* target level since a `Field` is defined by a *Method* declaration Java interfaces.

Note that, using Java interfaces and annotations for our schemas definition, we gain a first level of type checking from JVM. The reason is that before we run our runtime interpretation of schemas, JVM performs type checking in the definitions and in case wrong types it notifies the programmer. Additionally, this benefits the programmer who uses IDE's that perform real time type inspection[4] while typing, since errors on the definitions will be spotted immediately.

A list of the available structure concepts that are supported in our language is the following [LvdSC12]:

@Key When a method (field definition) is annotated with the `@Key` annotation that forces its value to be unique within collections of this field's Klass. The key should be used on a single field of a Type and its value represents the uniqueness of its Klass's instance. Another way to look at this is as a counterpart of the `hashCode` in traditional Java programs. This way when many values of a Klass are in a Set, the key field ensures uniqueness in its context.

@Inverse This annotation includes two *annotation element definitions* [5]. When a method is annotated with the `@Inverse(Class other, String field)` annotation, then the inverse `field` element must be a `Field`'s name in the `Class` interface, given by the `type` element. This meta data is used as a reference declaration in schemas meaning that when a programmer updates the value of a field that is annotated with inverse, then the value of the field that refers to will be also updated. This mechanism is interpreter by the managed object and is used for automated *wiring* of the field across a schema.

@Contain When a field is annotated with the `@Contain` annotation, then this field is considered as *traversal*. In general, traversals describe a minimum spanning tree that is called *spine* and ensures reachability of values. The spine is used in implementations that need a depth-first search by distinguishing between the actual information and the cross-references of the spanning tree. If a spanning tree is defined, then all nodes in a model must be uniquely reachable by following just the spine fields [SCL+12]. An example of such functionality is the equivalence between managed objects that is presented in Section 4.4.1. Sometimes traversal fields describe composition, or "is a part of", relationships [LvdSC12].

@Optional When the `@Optional` annotation is on a field's definition this field is allowed to not include value (`null`). `Inverse` fields are `Optional`.

Java Inheritance In addition to the Java annotations, our language uses more Java mechanisms for schemas definition. Java inheritance is one of them. A `schemaKlass` can extend another Klass (super), which works as the traditional Java inheritance, supporting sub typing mechanisms. Implementing this we introduce a *Type Hierarchy* model that includes super and sub classes on managed objects. Note that since we use interfaces for `schemaKlass`, we implicitly support multiple inheritance because a Java interface can extend more than one interfaces.

Java Collections Finally, another Java mechanism that we use is the definition of a field that includes many values. To define such a field, a programmer has to declare a field's `Type` as a `java.util.List` or a `java.util.Set` of this `Type`.

Using all the aforementioned constructs of our data definition language, a programmer can define any kind of schemas, even itself (see Section 4.2). An example of schema definition is presented in Chapter 3 Listings 3.3, 3.4 and 3.5. In those definition the above concepts can be recognized and their meaning in context can be revealed.

[4] https://www.jetbrains.com/help/idea/15.0/code-analysis.html
[5] https://docs.oracle.com/javase/tutorial/java/annotations/declaring.html

4.1.2 Schema Factories

However, even if we have the definitions of schemas, we still need a way to create instances of managed data described by them. We can not use Java's mechanisms for this functionality since we need them to be managed data and not ordinary objects. Thus, we use Java interfaces (again) to define *Schema Factories*. A *Schema Factory* is a list of constructor definitions for specific schema Klasses.

The methods of this interface are used like the constructor definition on a Java class. However, in our case they are defined as methods in a Java interface while their implementation is handled by the data managers. Since those methods are constructors, we can define a constructor with or without initial values. Unfortunately, we have encountered a limitation that considers the constructors with initialization values, which makes them inappropriate to use in complicated schemas.

Methods Ordering Issue

The problem lays on Java's reflection mechanisms in terms of methods ordering. More specifically, when the methods of a `java.lang.Class` are requested by using the `public Method[] getMethods()` method[6], the returned values are not ordered with the same order as they are defined in the source code. Consequently, since the schema definition is reflectively analyzed in the data managers and it depends on that order, those methods can not be used in the initialization of values.

However, we have implemented an alternative way in order to support such feature. In our implementation we **alphabetical order** by name both the defined methods and the fields before we initialize them. That feature can be used by the programmers although it can be proved confusing. Therefore, as an advice about the usage of the constructor definition, we suggest to either provide only constructors without initialization values or to write constructors with only **primitive** initialization values in **alphabetical order**. In the case of initialization with values the danger that the fields of the schema will get values in different order exists, leading to an error or a wrong value assignment.

4.1.3 Data Managers Implementation

However, the schemas are not a complete managed data specification without a corresponding `Data Manager`. A data manager is responsible to interpret the schema and build virtual objects (managed objects). The managed object's fields are specified by the given schema and acts according to the specification given by the data manager. Additionally, the data manager ensures that the data given are valid with respect to schema. More specifically, the data managers describe how a schema definition is handled from the outside world and what are its specifications. These properties many include CCC that can be described separately by special data manager, separating schema and concern definitions. Thus, a managed object can have multiple interpretations based on the data manager that they interpret it.

A data manager is initialized with a `Schema` and provides a new `Managed Object` instance which has the properties given by that data manager. Additional to the `Schema` that includes a Set of `Types` (`Primitives` or `Klasses`), it also needs a `Schema Factory` that declares the constructors of the given schema `Klass`. After the initialization of data manager and the interpretation of the schemas, a data manager provides the mechanism of creation new `Schema Factories`, which in turn create `Managed Objects` with the specifications of the data manager.

An example presented in Chapter 3 is shown in listing 4.11, the Line 3 defines a basic data manager, that gets as input the schema factory and the schema of a state machine. The schema in these case has been already loaded, this processes is described in Section 4.2.3. Next, in Line 7 a new schema factory is created that builds managed objects with the specification that are attached from the basic data manager. Finally, Line 10 shows how those managed object instances with those specification can be built.

[6] As it is mentioned in https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#getMethods--, the elements in the returned array are not sorted and are not in any particular order.

```
// Create a basic data manager for state machines
BasicDataManager basicDataManagerForStateMachines =
        new BasicDataManager(StateMachineFactory.class, stateMachineSchema);

// Create a schema factory that makes managed objects
// with the specifications of the basic data manager.
StateMachineFactory stateMachineFactory = basicDataManagerForStateMachines.make();

// Build an instance of managed object with those specifications.
Machine stateMachineInstance = stateMachineFactory.Machine();
```

Listing 4.11: Basic data Manager Example

Basic Data Manager

As described, we use Java interfaces to define schema Klasses that include fields. Those fields are know only dynamically and a data manager has to be able to determine the fields and methods of the managed data object during runtime. In addition, when a data manager adds functionality on a managed object then it delegates the calls to the fields of an instance, to its specifications first. In order to dynamically interpret a schema inside a data manager and delegate functionality, we used Java Reflection and Dynamic Proxies.

In our implementation we have separated the Proxy factory (`DataManager`) from the Invocation Handlers (`MObjects`). This way, the `DataManager` class is responsible for creating proxied instances of managed data, while the `MObject` instances are responsible for interpreting the schema and delegate actions with their invocation handling mechanisms. Figure 4.1 illustrates their structure. As it can be seen the data manager is a *factory* that has only one exposed method, `make()` that it is used to build an `SchemaFactory`, which in turn builds `MObject` instances.

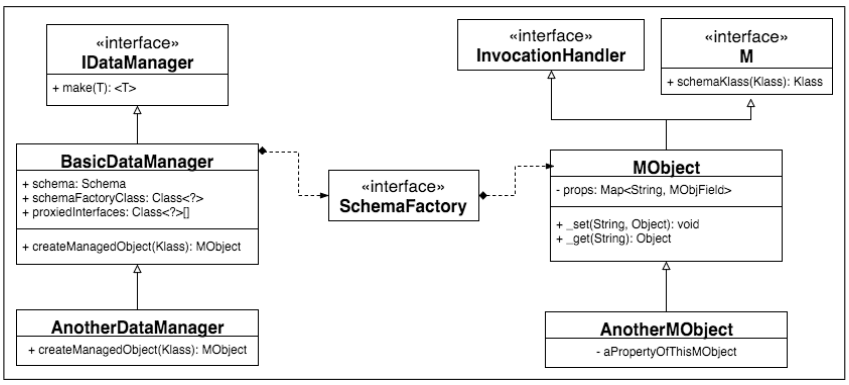

Figure 4.1: Data Manager and MObject

Stacking Data Managers

In order to create a stack of data managers that they combine the behavior and specifications, we can use inheritance. Figure 4.1 show how this works. In detail, the `AnotherDataManager` data manager extends the `BasicDataManager` and overrides only the `createManagedObject()` method. This method is responsible for creating a new instance of an `MObject`. In the case of `AnotherDataManager` implementation the `createManagedObject()` method will create a new `AnotherMObject` instance. Additionally, the constructor of data managers needs to accept a dictionary of initialization parameters for overriding data managers that require different inputs. Note that it is important that the data managers inherit from a base one, this leads to the **modular aspect** of the data managers. As it can be seen for stacking data managers we used *Decorator Pattern* [Gam95] which is mentioned also in Cook et al. [LvdSC12] as a strategy for static OOP languages.

4.1.4 Managed Objects

As it is mentioned the `MObject`, is an implementation of the `InvocationHandler` interface. Thus, the `MObject`'s `invoke()` method is called in every field access of an managed object's instance. To manipulate the its fields values this object has two methods, `_set()` and `_get()`. In the implementation of these methods additional checks are performed to ensure the correctness of types and structure of the values. Those methods can be overridden from derived `MObject`s in order to *Decorate* the basic `MObject` with their functionality, of course they require to call their `supers` for running all the checks.

The fields of the `MObject` are specified by its the `schemaKlass`. The `MObject` is the *backing object* that stores a reference to the `schemaKlass`. That `schemaKlass` is a meta class that describes the layout of the `MObject`. The `schemaKlass` is a `Klass` that keeps the `Field`s and their `Type`s of the `MObject`. The `MObject` implementation is an instance of that `schemaKlass`. During construction, the `MObject` setups all the fields based on its `schemaKlass`. Then whenever a field check has to be performed, the `MObject` uses its `schemaKlass`. The usage of the `schemaKlass` for setup the fields is shown in Listing 4.12.

The `schemaKlass` is given to the `MObject` by the `DataManager` that is responsible for creating it. Using this `schemaKlass` the `MObject` setups the `Fields` of the Klass, Line 3. Inside the `setupField` method the interpretation of the schema is performed. In particular, in Line 10 we check if that field is a multi-value field, if it is not we just setup it as a `Primitive` or a `Klass` accordingly. Consider that the `field.type().schemaKlass().name()` is used like a common `instanceof` in Line 14. In the case that the field has many values, then we first check if it is `Primitive` because we do not support Set of `Primitive`s. If not, we check if a `Key` field exists on that field's type and if it is that field is therefore a Set otherwise is a List.

```
public MObject(Klass schemaKlass, Object... initializers) {
  this.schemaKlass = schemaKlass;
  this.schemaKlass.fields().forEach(this::safeSetupField);
  if (initializers != null) {
    this.safeInitializeProps(initializers);
  }
}

protected void setupField(Field field) {
  if (!field.many()) {

    // if it is a primitive make it a Primitive field,
    // otherwise a reference (managed object)
    if (field.type().schemaKlass().name().equals("Primitive")) {
      this.props.put(field.name(), new MObjectFieldSinglePrimitive(this, field));
    } else {
      this.props.put(field.name(), new MObjectFieldSingleMObj(this, field));
    }
  } else {

    // in case it is a Primitive, then is always a List
    // Sets of Primitives are not supported (yet)
    if (field.type().schemaKlass().name().equals("Primitive")) {
      this.props.put(field.name(), new MObjectFieldManyList(this, field));
    } else {

      Klass klassType = (Klass) field.type();
      if (klassType.key()!= null) {
        this.props.put(field.name(), new MObjectFieldManySet(this, field));
      } else {
        this.props.put(field.name(), new MObjectFieldManyList(this, field));
      }
    }
  }
}
```

Listing 4.12: MObject: setup fields

4.1.5 Implementing a Data Manager

The implementation and the integration of a new data manager is straight forward in our framework. As it can be seen in Figure 4.1, the basic ingredients of a new data manager implementation is the `Data Manager` class (proxy) and the `MObject` class (invocation handler).

First, to follow the modularity aspect and the ability to stack data managers together combining their specifications, we need to inherit from, at least, the `BasicDataManager` and `MObject` respectively. A simple data manager that could be proved useful, is a data manager that introduces immutability of its managed objects. A `Lockable` data manager should first inherit the `BasicDataManager` to get its field access specification. The implementation of the `LockableDataManager` is illustrated in 4.13.

```
public class LockableDataManager extends BasicDataManager {

  public LockableDataManager(Class<?> moSchemaFactoryClass, Schema schema) {
    // Add the Lockable class in order to use it in the managed object.
    super(moSchemaFactoryClass, schema, Lockable.class);
  }

  @Override
  protected MObject createManagedObject(Klass klass, Object... _inits) {
    return new LockableMObject(klass, _inits);
  }
}
```

Listing 4.13: Lockable Data Manager

Additionally it should add some *locking* mechanism to ensure immutability of its objects. This is defined in the `Lockable` interface which is responsible of ensuring the implementation of the specifications. Figure 4.14 shows the specification of the interface.

```
public interface Lockable {
  void lock();
}
```

Listing 4.14: Lockable Interface

Since we have the specifications and the data manager that creates the *Lockable* managed object, we still need the implementation. The implementation is located in the MObject and in this case the `LockableMObject`, Figure 4.15.

```
public class LockableMObject extends MObject implements Lockable {
  private boolean isLocked = false;

  public LockableMObject(Klass schemaKlass, Object... initializers) {
    super(schemaKlass, initializers);
  }

  public void lock() {
    isLocked = true;
  }

  @Override
  public void _set(String name, Object value)
  throws NoSuchFieldError, InvalidFieldValueException, NoKeyFieldException {
    if (isLocked) {
        throw new IllegalAccessError(
          "Cannot change " + name + " of locked object " + schemaKlass.name() + ".");
    }
    super._set(name, value);
  }
}
```

Listing 4.15: Lockable Managed Object

The `LockableMObject` by extending the `MObject` and implementing the `Lockable` interface it inherits the basic functionality of a managed object in and gets a specification description respectively. Its role is to implement the logic of the immutability, which is as simple as it looks. In order to use this functionality, one needs to create managed objects using this data manager. An example is shown in Figure 4.16.

```
final PointFactory lockablePointFactory = lockableFactory.make();
final Point2D lockablePoint = lockablePointFactory.Point2D(1, 2);

// It was mutable until now, now it is locked (immutable).
((Lockable)lockablePoint).lock();

try {
  lockablePoint.x(2); // Should throw here since its immutable.
} catch (IllegalAccessError e) {
  System.out.println("IllegalAccessError: " + e.getMessage());
}
```

Listing 4.16: Immutability Example

4.2 Self-Describing Schemas

As it is described by Cook et al. [LvdSC12], a self-describing schema is a schema that can be used to define schemas, including itself. Our framework is fully self-described, the schemas are also described by schemas which are both models [KBJV06]. To allow schemas to be managed data we need a "self-describing schema mechanism" or *SchemaSchema*. Through the *SchemaSchema* the approach of managed data can be applied at the meta level as well.

The reason that a self-describing schema is important is because schema schemas can be used from schema factories to create schemas. The schema of schemas is just a schema that allows the creation of schemas, including its own schema [SCL+12]. Additionally, by presenting the schema as the first-class model[KBJV06], they can be extended in the same way just like ordinary models.

4.2.1 SchemaSchema

By using Java interfaces the *Schema* classes are tightly coupled structurally to the Java interface we use to define them. Since we want to decouple from Java interfaces and reflection we need our own *Klass system*. In order to be self-describing we want this Klass system to be represented as managed data as well. To model the structure of a `Schema` itself we need to be able to describe a class as a collection of `Fields`, each of which it has a `name` and a `Type` [LvdSC12]. Thus, for our *SchemaSchema* definition we need a `Type`, a `Field` and a `Schema` as a collection of `Types`. A `Type` could be both a `Primitive`, no `Fields` and a `Klass`, has a set of `Fields`. Additionally, those `Fields` may have some extra meta data attributes that explained in Section 4.1.1.

A schema like this can describe itself since every concept used in the explanation is included in the definition. For a self-describing implementation we need to describe our own SchemaSchema.

Figure 4.2 illustrates the modeling of this definition.

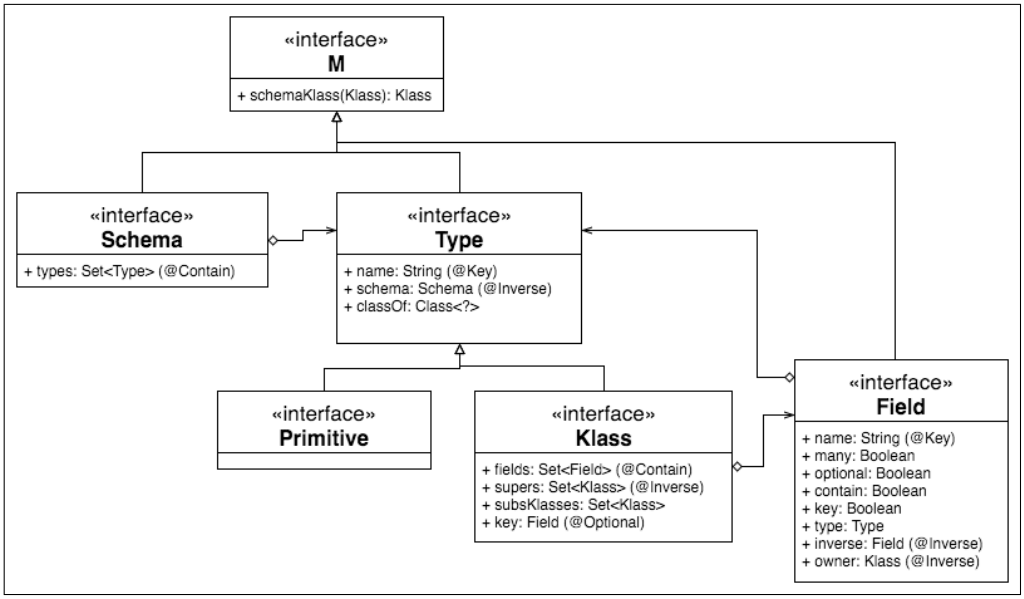

Figure 4.2: The schema of schemas

4.2.2 SchemaFactory

Considering that we have the schema of our schema (*SchemaSchema*) we need a way to create instances of those *schemaSchemaKlasses*. In this case, as we do in the normal schemas, we use a schema factory. However, this time it is a *schemaSchemaFactory* that defines constructors of all the schema klasses that need to describe our *SchemaSchema*. Listing 4.17 shows its definition.

```
1 public interface SchemaFactory {
2     Schema Schema();
3     Primitive Primitive();
4     Klass Klass();
5     Field Field();
6     Field Field(
7       Boolean contain, Boolean key, Boolean many, String name, Boolean optional);
8 }
```

Listing 4.17: Schema SchemaFactory

4.2.3 Schema Loading

To construct the Klass system we need to analyze the Java interfaces, using reflection, and then build actual instances of the Schema, Klass, Field etc. using the appropriate factory. The `SchemaLoader` is responsible of this process.

`SchemaLoader`'s `load` static method takes as input a Set of interfaces, which are the schema definitions, a `SchemaSchemaFactory` that includes constructor definitions of the `SchemaSchema` and a instance of the `SchemaSchema`. During the reflective analysis of the input interfaces the `SchemaLoader` builds the corresponding `Types` and `Fields` of those interfaces using the `SchemaSchemaFactory`. A `Schema` consists of the Set of these `Types`. An example taken from Chapter 3, is shown in Listing 4.18.

```
final Schema schemaSchema = ...;
final SchemaFactory schemaFactory = ...;

final Schema stateMachineSchema = SchemaLoader.load(
  schemaFactory,
  schemaSchema,
  Machine.class,
  State.class,
  Transition.class);
```

Listing 4.18: SchemaLoader Example

In the code, the `SchemaLoader` gets as input a `SchemaSchemaFactory` and a `SchemaSchema`, which are not important how they have created yet. Moreover, it gets a set of interfaces that describe the state machine schema. This schema consists of a set of schema Klasses that are described by interfaces, namely `Machine.class`, `State.class` and `Transition.class`. Next, the `SchemaLoader` analyzes the definition of those schemas using reflection and then makes a `Schema` by using the `SchemaFactory` that has been given. A simple description of that process is shown in Listing[7] 4.19. It can be seen that first we implement the instances and after we use setters to wire them up and the reason for is that not everything exists the time that needs to be set.

[7] Most of the implementation has been excluded for brevity.

```
public static Schema load(
  SchemaFactory factory, Schema schemaSchema, Class<?>... schemaKlassesDef)
{
  // create an empty schema using the factory, will wire it later
  final Schema schema = factory.Schema();

  // build the types from the schema klasses definition
  final Set<Type> types = new LinkedHashMap<>();
  for (Class<?> schemaKlassDefinition : schemaKlassesDefinition) {
    final String klassName = schemaKlassDefinition.getSimpleName();

    // build the fields from method definitions
    final Set<Field> fieldsForKlass = new LinkedHashMap<>();
    for (Method schemaKlassField : schemaKlassDefinition.getMethods()) {

      // field the field metadata though annotations
      // ...
      // add its fields, the owner Klass will be added later
           final Field field = factory.Field();
           field.name(fieldName);
           field.contain(contain);
           field.key(key);
           field.many(many);
           field.optional(optional);

           fieldsForKlass.add(field);
    }

    // create a new klass
    final Klass klass = factory.Klass();
    klass.name(klassName);
    klass.schema(schema);
    // wire the owner klass in fields,
    fieldsForKlass.values().forEach(field -> field.owner(klass));
  }

  // wire the types on schema,
  // it is inverse so it will refer to schema.types() immediately
  types.forEach(type -> type.schema(schema));

  return schema;
}
```

Listing 4.19: SchemaLoader

Listing 4.19 shows the usage of Java reflection in our implementation. However, because Java reflection capabilities are limited, this restricted us in our implementation as well.

4.3 Bootstrapping

Considering that SchemaSchema is itself managed data, we can use the SchemaLoader to build a new SchemaSchema. Nonetheless, we need a description of that SchemaSchema, which will be used during the loading process to build the schema Klasses. As a result, we need a *Bootstrap Schema* to jumpstart this process. The *Bootstrap Schema* is necessarily self-describing as it must manage itself

[LvdSC12] and it is hardcoded in its own class `BootSchema`.

4.3.1 Cutting the umbilical cord

Having a `BootSchema` in place we can now create "real" `SchemaSchemas` [8]. For Consistency, we use those "real" `SchemaSchemas` in order to build other schemas, this way everything is managed data. After building a real SchemaSchema we do not need the `BootSchema` anymore, which leads to a process that we can call "Cutting the umbilical cord". An example of "Cutting the umbilical cord" is shown in Listing 4.3.1, in which we use the `BootSchema` to build the `realSchemaSchema` and then we use this `realSchemaSchema` to build another `realSchemaSchema` (`realSchemaSchema2`).

```
final Schema bootstrapSchema = new BootSchema();
BasicDataManager basicFactory =
  new BasicDataManager(SchemaFactory.class, bootstrapSchema);

// Create a schema Factory which creates Schema instances.
final SchemaFactory schemaFactory = basicFactory.make();

// The schemas are described by the SchemaSchema,
// this schemaSchema is also self-describing.
final Schema realSchemaSchema =
       SchemaLoader.load(
         schemaFactory,
         bootstrapSchema,
         Schema.class,
         Type.class,
         Primitive.class,
         Klass.class,
         Field.class);

BasicDataManager basicFactory2 =
  new BasicDataManager(SchemaFactory.class, realSchemaSchema);
final SchemaFactory schemaFactory2 = basicFactory2.make();
final Schema realSchemaSchema2 =
       SchemaLoader.load(
         schemaFactory2,
         realSchemaSchema,
         Schema.class,
         Type.class,
         Primitive.class,
         Klass.class,
         Field.class);
```

Listing 4.20: Cutting the umbilical cord

Figure 4.3 illustrates the models during a bootstrapping process. As it can be seen, the Boot Schema is used in order to describe the Schema Schema, after that the Schema Schema is independent and managed data it self. Thus, it can be used to create other schemas like the Machine schema or even itself.

[8] We call them real because they are managed data and not hard-coded.

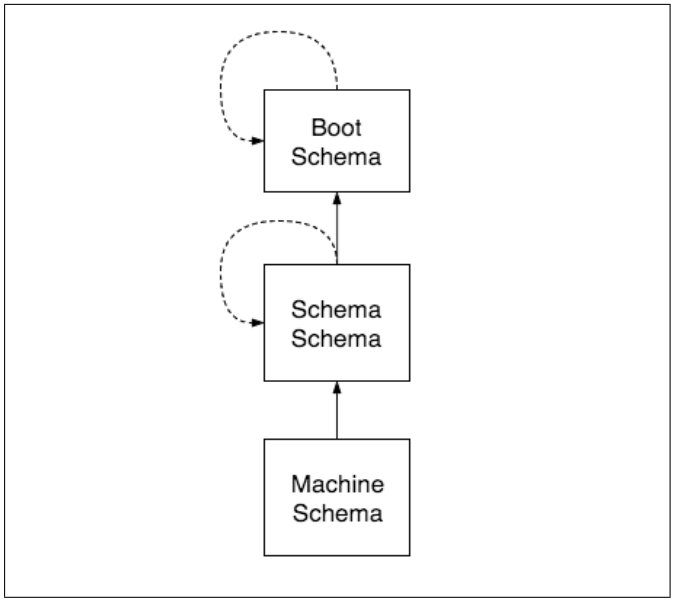

Figure 4.3: Boot Schema models

4.4 Implementation Issues

The fact that we use Java reflection and dynamic proxies, along with the fact that everything is managed data, even the schemaSchema, introduces some issues including the methods ordering problem described in Section 4.1.2.

4.4.1 Equivalence

The `bootstrapSchema`, `realSchemaSchema` and `realSchemaSchema2` managed objects from the Listing 4.3.1 should be equal because they ultimately describe the same *Schema*. However, since they are managed data and not normal Java objects, at least the real SchemaSchemas, we need a to check for equality on managed objects. We have implemented the equivalence functionality for managed objects, using the *Equality Checking for Trees and Graphs algorithm* by Michael D. Adams and R. Kent Dybvig [AD08].

4.4.2 The classOf field

As it has be presented in Section 2.4.2, for a proxy object to conform with interfaces and be casted to any of them, it needs these interfaces during its initialization. To support that, we have added the `classOf` field in the `Type` schema Klass, which is of type `java.lang.Class` and is a reference of the Java class that this schema Klass is described to.

4.4.3 Hash-code of Managed Objects

To avoid any unpredictable activities that a `hashCode` invocation would bring in case of its invocation in managed objects, we have omitted it. We do not depend on the ordinary `hashCode` for managed objects, we have not implemented it and we do not call it. If it is a collection field type, then the field has to have a `Key` field, thus, we obtain the value of the key field and index into a `HashMap`. Using the `Key` field as the key of the hashmap work either it is a primitive or not, since we get the `Object.hashCode()` of that key. However, that suggests that the key is not of our schema Klass system but a Java type. Finally, the `Mobject` invocation handler delegates the call of `hashCode` method to the real object so that would never fail fail, although this is not suggested because it may lead to unpredictable results.

4.4.4 Java 8 Default Methods

Java 8 supports the definition of default methods in interfaces. According to the specification[9], default methods enable you to add new functionality to the interfaces and they can be used as method implementation in abstract classes. We use Java 8 default methods in order to add default functionality in our schemas. In particular, methods that are defined as *default* are ignored during the interpretation and no fields are created for them. We consider this as a helpful mechanism for defining functionality inside the schemas. A notable feature is that make the default method invocation in the MObject is `protected`, which makes possible to the derived data managers to "monitor" when a default method is invoked and use it.

4.5 Benefits and Limitations

One of the advantages of this language is the simplicity of its usage. A programmer just needs to define the schemas followed by the data managers and can easily write a program using them. The language takes care of the dependencies, references and any other underline mechanisms. Moreover, it uses Java concepts, which it makes it safer in terms of type checking and definitions it is easier for Java developers to adapt. Furthermore, by being a self-describing language it is no longer bounded to the Java constructs and everything now is managed data. Finally, the effortless mechanism of stacking data managers makes it significantly modular on every level, meta or not.

On the other hand, in addition to the implementation issues which described in the previous section, there are significant performance implications since we use Java reflection and dynamic proxies to dynamically interpret schemas. This makes it undesirable for applications that focus on performance and are based on JVM optimizations. Another issue that arises is that it is hard to be used in existing systems because to make a consistent integration, every model has to redefined as schema and every functionality to reimplemented in data managers. However, an existing system integration is presented in Chapter 5.

4.6 Claims

We claim that managed data leads to a powerful data abstraction that gives the programmers control over fundamental mechanisms of creation and manipulation of data [LvdSC12]. Those mechanisms are used to be predefined by the programming languages and managed data gives control over them using data managers. Moreover, we claim that managed data introduces a modular way to define data and aspects of them. We are going to present it in Chapter 5, by showing how can we *aspect refactor* an existing application using managed data.

[9] https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

Chapter 5

Taming Aspects of JHotDraw with managed data

To put our implementation and practice and evaluate the ability of managed data to tame aspects, we have refactor some of the CCC of JHotDraw. More specifically, in this chapter is presented the refactoring of the *FigureSelectionListener* observer pattern as well as the *Undo* concerns of JHotDraw. For their refactoring we used our implementation of managed data in Java that presented in the previous chapter. However, to do that, we have migrated JHotDraw in managed data. The result of this migration is available to an open-source project called ManagedDataJHotDraw[1]. Additionally, we claim that this is the first aspect refactoring of an application using managed data to date.

5.1 Refactoring Process

The refactoring of such application like JHotDraw required a significant amount of time to study and familiarizing with the existing system. In particular, we tried to focus only on the parts that we were going to refactor since we wanted to assess the same refactorings that AJHotDraw developers [MM] did. Thanks to their fan-in analysis [MVDM04], with which identified a number of aspects in the legacy system, we also emphasize on them in order to make a fair comparison. Furthermore, while the AJHotDraw focused on a completely new version of JHotDraw written using AspectJ, we implemented ManagedDataJHotDraw maintaining the coherence and the original design.

We first need to migrate ....

5.2 Migration of JHotDraw to Managed Data

Everything as managed data?

[1] https://github.com/TheolZacharopoulos/ManagedDataJHotDraw

5.2.1 The MDStandardDrawingView

MDStandardDrawingView Factory

5.2.2 Issues

The @NotManagedData annotation

5.2.3 Limitations

5.3 Aspect Refactoring of JHotDraw

Aspect refactoring refers to the refactoring of legacy in aspect oriented code. However, here we present an aspect refactoring of legacy code in managed data.

5.3.1 FigureSelectionListener

The `FigureSelectionListener` observer pattern of JHotDraw is a case that first presented by Hannemann et al. [HMK05] in their tole-based refactoring of design patterns in AspectJ. Later Marin et al. picked the same aspect and migrated it to their AJHotDraw implementation. Likewise, we have also considered the same aspect for our refactoring in order to assess our concern solution with the existing one.

5.3.2 FigureSelectionListener in JHotDraw

The original observer pattern of the `FigureSelectionListener` functionality in JHotDraw is illustrated in Figure 5.1.

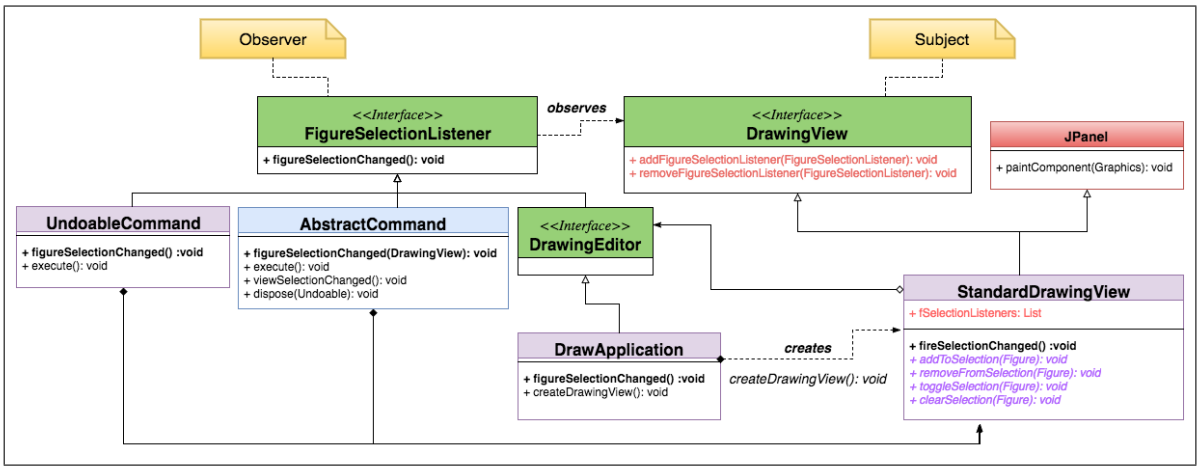

Figure 5.1: FigureSelectionListener in JHotDraw

As the figure illustrates,

5.3.3 Refactoring FigureSelectionListener in AJHotDraw

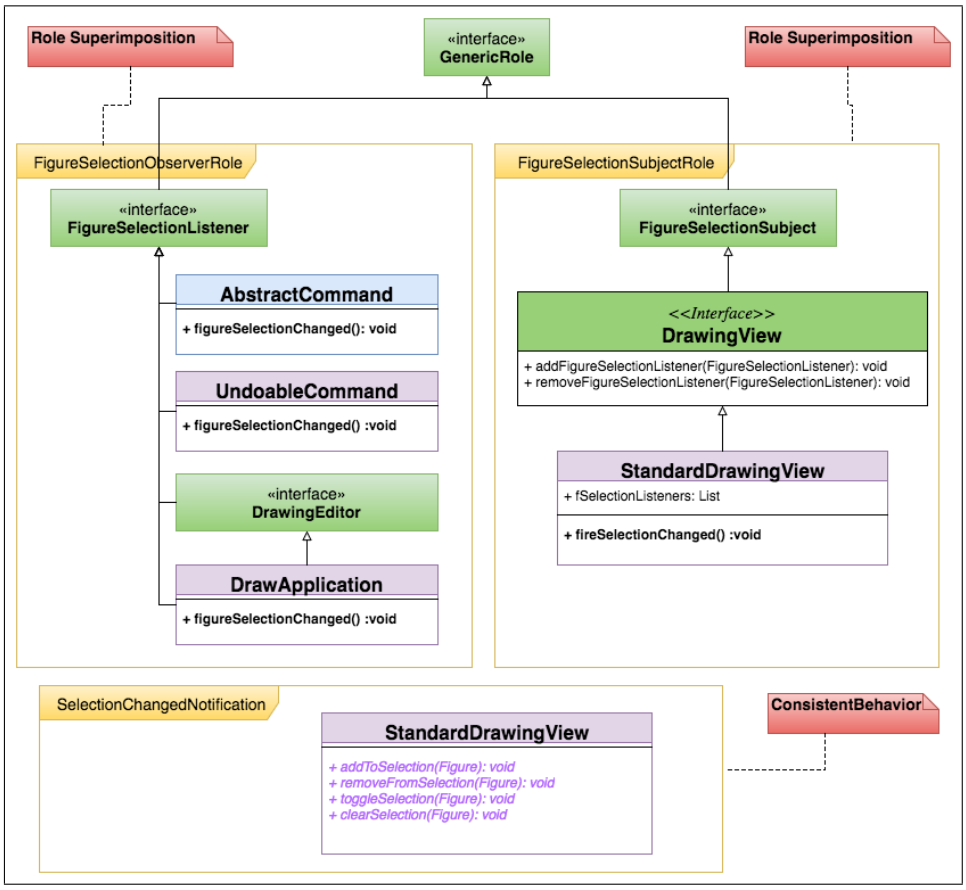

Figure 5.2: FigureSelectionListener in AJHotDraw

Role Superimposition

Consistent Behavior

```
1  public aspect SelectionChangedNotification {
2    pointcut invalidateSelFigure(StandardDrawingView sdw) :
3      (   withincode(boolean StandardDrawingView.addToSelectionImpl(Figure))
4        withincode(void StandardDrawingView.removeFromSelection(Figure)))
5      && call(void Figure.invalidate())
6      && this(sdw);
7
8    pointcut clear_toggleSelection(StandardDrawingView sdw):
9      (execution(void StandardDrawingView.clearSelection())
10      execution(void StandardDrawingView.toggleSelection(Figure)))
11     && this(sdw);
12
13   after(StandardDrawingView sdw): invalidateSelFigure(sdw) {
14     sdw.fireSelectionChanged();
15   }
16
17   after(StandardDrawingView sdw): clear_toggleSelection(sdw) {
18     sdw.fireSelectionChanged();
19   }
20 }
```

Listing 5.21: AJHotDraw: Consistent Behavior in FigureSelectionListener

5.3.4 Refactoring FigureSelectionListener in ManagedDataJHotDraw

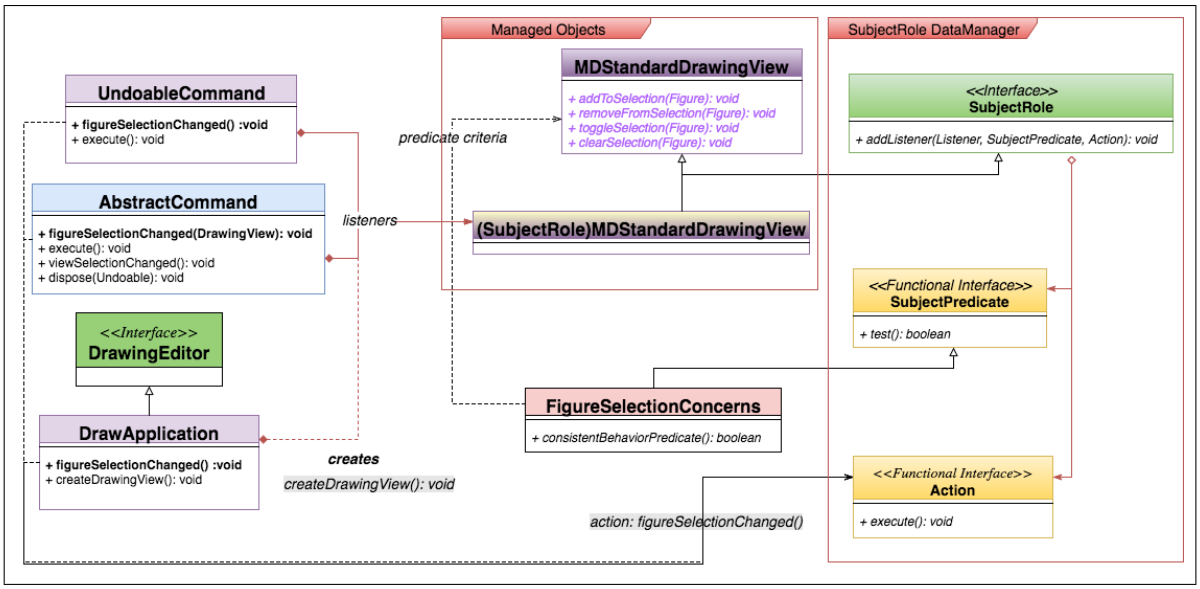

Figure 5.3: FigureSelectionListener in ManagedDataJHotDraw

5.3.5 SubjectRole Data Manager

Data manager

MObject

Predicate

Action

5.3.6 SubjectRole Integration

Consistent Behavior Predicate

FigureSelectionChanged Action

5.4 Claims

5.5 Threads to Validity

Chapter 6

Evaluation

6.1 Research Questions and Answers

6.2 Evidence

6.3 Results

6.3.1 Locality

6.3.2 Reusability

6.3.3 Composition transparency

6.3.4 (Un)pluggability

6.4 Claims

6.5 Discussion

6.5.1 In Practice

6.5.2 Benefits and Pitfalls

6.5.3 Modularity

6.6 Remarks

Chapter 7

Conclusion

Chapter 8

Further Work

Acknowledgments

Bibliography

[AD08] Michael D Adams and R Kent Dybvig. Efficient nondestructive equality checking for trees and graphs. In *ACM Sigplan Notices*, volume 43, pages 179–188. ACM, 2008.

[Eug06] Patrick Eugster. Uniform proxies for java. *ACM SIGPLAN Notices*, 41(10):139–152, 2006.

[FFI04] Ira R Forman, Nate Forman, and John Vlissides Ibm. Java reflection in action. 2004.

[Fow09] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 2009.

[Gam95] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *ACM Sigplan Notices*, volume 37, pages 161–173. ACM, 2002.

[HMK05] Jan Hannemann, Gail C Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 135–146. ACM, 2005.

[KBJV06] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Model-based dsl frameworks. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 602–616. ACM, 2006.

[KDRB91] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *ECOOP 2001Object-Oriented Programming*, pages 327–354. Springer, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97Object-oriented programming*, pages 220–242. Springer, 1997.

[LL00] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd international conference on Software engineering*, pages 418–427. ACM, 2000.

[LvdSC12] Alex Loh, Tijs van der Storm, and William R Cook. Managed data: modular strategies for data abstraction. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 179–194. ACM, 2012.

[Mar04] Marius Marin. Refactoring jhotdraws undo concern to aspectj. In *Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE 2004)*, 2004.

[MM] Marius Marin and Leon Moonen. Ajhotdraw: A showcase for refactoring to aspects.

[MMvD05] Marius Marin, Leon Moonen, and Arie van Deursen. An approach to aspect refactoring based on crosscutting concern types. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5. ACM, 2005.

[MSD15] Stefan Marr, Chris Seaton, and Stéphane Ducasse. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 545–554. ACM, 2015.

[MVDM04] Marius Marin, Arie Van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 132–141. IEEE, 2004.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[PSH04] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for java futures. In *ACM SIGPLAN Notices*, volume 39, pages 206–223. ACM, 2004.

[SCL+12] T Storm, WR Cook, A Loh, K Czarnecki, and G Hedin. Object grammars: Compositional & bidirectional mapping between text and graphs. 2012.

[Ste05] Friedrich Steimann. Domain models are aspect free. In *Model Driven Engineering Languages and Systems*, pages 171–185. Springer, 2005.

[Sul02] Gregory T Sullivan. Advanced programming language features for executable design patterns" better patterns through reflection. 2002.

[TBG03] Tom Tourwé, Johan Brichau, and Kris Gybels. On the existence of the aosd-evolution paradox. *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, 2003.

[YJ02] Joseph W Yoder and Ralph Johnson. The adaptive object-model architectural style. In *Software Architecture*, pages 3–27. Springer, 2002.