Taming Aspects with Managed Data

Master's Project in Software Engineering

Theologos A. Zacharopoulos
theol.zacharopoulos@cwi.nl

Summer 2016, 64 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

TODO

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 its 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 though it provides modularity. The main 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 its meta-programming framework. 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 our first task it to implement it 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. Additionally, we compare our results with the AOP counterpart implementation of JHotDraw.

1.1 Initial Study

In their study on managed data, Cook et al. [LvdSC12] presented the main idea of managed data, while using a show case of it in a Ruby implementation. As a use case, they presented the Enso[1] 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

[1] http://enso-lang.org/
[2] https://www.ruby-lang.org/en/

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 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 language 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 [Par72]. 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 an 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 structure manipulation mechanisms.

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

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

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, which considered a challenge, therefore our first research questions states *"Can managed data be implemented in a static language?"*. Based on the 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. As a result the third research question states *"To what extent can managed data tame an inventory of aspects 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 implement 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 existing application.

To validate our hypotheses we have implemented managed data in Java. More specifically we define *schemas* using Java interfaces and dynamic proxies for the *data managers*. Furthermore, we provide as a proof of concept the an example given in Enso papers [LvdSC12], but this time developed in Java using our framework.

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

The Observer Pattern, which as presented in literature [TBG03] [HMK05] [MMvD05a], is by nature not modularized and the scatters "pattern code" through the participant 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.

We performed aspect refactoring 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 the original JHotDraw test suite that consists of 1218 executable tests in total.

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.

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 have implemented managed data in Java using Java's reflection capabilities[5]. 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 on taming aspects in general, we have used a use case application (JHotDraw) which is considered as a good design use case for OOP, along with its AOP implementation (AJHotDraw). Thus, we have built our version of the JHotDraw application (ManagedData-JHotDraw) 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, in `jar` format, 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 Refactoring of 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 *Observer* pattern, which has been used in multiple parts of JHotDraw and cuts "pattern code" on different modules, as well as the *Undo* concern, which is part of a *Command Pattern* and it is scattered through the modules that use this functionality.

Contribution 4: JHotDraw Refactoring Results and Comparison with AJHotDraw. Finally, we present the results of our aspect refactoring and we compare them with AJHotDraw which is implemented in 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 have 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 have 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/

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 it gives the ability to cleanly integrate something outside the language's scope. 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 important characteristics of managed data. AOM is also more focused on business systems implementation, 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[6] is the first implementation of managed data, it is open source[7] 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 one of the topics in this thesis and is going to be presented extensively in the following chapters.

[6] `http://enso-lang.org/`
[7] `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. Finally, a conclusion is given in Chapter 7.

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 OOP and PP [KLM+97]. 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 and evolve.

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 that 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, or it is an *aspect*, which is the opposite. 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, a process called *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.

Modularity Properties

During the assessment of their findings, the authors used four *modularity properties*. In this thesis we used the same properties to assess our results. The modularity properties are the following:

Locality The *locality* property refers to the ability of an existing class to be incorporated into a pattern instance with effortless adaptation. In this case, all the changes have to be made in the pattern instance.

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

Reusability *Reusability* holds when a class is not coupled to its role in a pattern. Therefore, it can be used in different contexts without modifications and the reusability of participants can be increased.

(Un)pluggability Both the *locality* and the *reusability* properties of a system make the pattern implementations *(un)pluggable*.

Composition transparency Having achieved *locality* and *reusability*, we have obtained an *(un)pluggable* system. This suggests that we can reuse generalized pattern code and localize the code for a particular pattern instance. Thus, creating multiple instances of the same pattern in one application, it is easier to understand (*composition transparency*). This way the problem of having multiple instances of a design pattern in one application is solved.

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 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 consequently this affects negatively the evolvability of a system since it makes it hard to change its structural. As Tourwe et al. propose, 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 aspects 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. Therefore, we need a more descriptive language that defines records and fields of records.

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 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 our case we will use the 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].

In order for this self-examination to be successful, the program needs to have a representation of itself as *metadata*. In a OOP language this metadata are 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 through its reflection API since the version 1.1. 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] https://docs.oracle.com/javase/tutorial/reflect/
[3] https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html

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. Its interface is shown in Listing 2.1.

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

Listing 2.1: The Invocation Handler Interface

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.

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, Section 2.1, we mentioned the *logging* CCC. Imagine 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
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 praised for its design 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 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 performed 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 [MMVD05b]. This led to the identification and documentation of CCC in JHotDraw, which helps the developers to identify CCC in it. In order to tame the aspects in a more consistent and formal way, Marin

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

et al. provided a list of *template aspect* solutions for 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, called TestJHotDraw[6], which ensures 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 allows for a better assessment of aspect orientation.

In this thesis we have used JHotDraw and AJHotDraw in order to evaluate our aspect refactoring in managed data. However, TestJHotDraw is written in AspectJ, a language we did not want include in our project, and therefore it is not used. Instead, we used the JHotDraw original test suite, which consists of 1218 test cases, for our refactoring behavioral preservation. The aspect refactoring process is described in Chapter 5.

2.5.1 Refactoring of Crosscutting Concerns

The refactoring of legacy code to aspect oriented code is also known as *Aspect Refactoring* [MMvD05a]. 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 [MMvD05a] 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

As discussed, 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.

Role-based Refactoring

The authors present a *role-based* refactoring, which consists of classifying the roles of the pattern in different aspects. The role-based refactoring approach helps the developer to transform a scattered implementation of 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).

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)

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

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 in Figure 2.1.

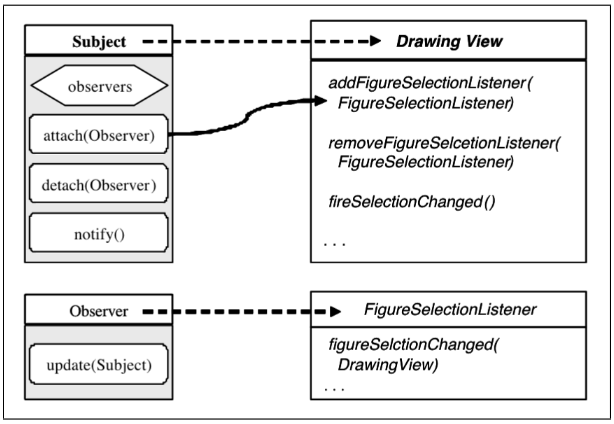

Figure 2.1: Observer pattern: Role Mapping [HMK05]

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

During the AJHotDraw implementation[MMvD05a], the authors proposed a type-based refactoring on the same *Observer* instance, as Hannemann did [HMK05], the `FigureSelectionListener`.

The concern sorts 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 invoking a specific action as a step in their execution.

The legacy design of JHotDraw is displayed in Figure 2.2.

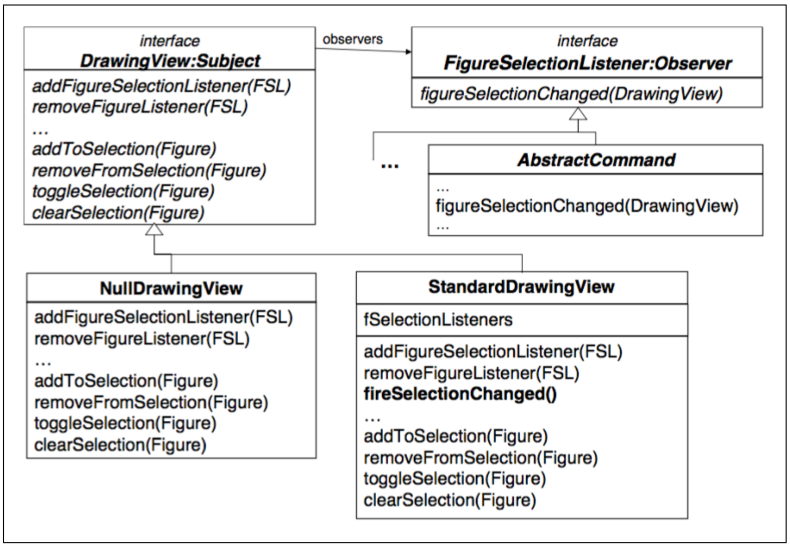

Figure 2.2: Observer pattern: Selection Listener [MMvD05a]

The `FigureSelectionListener` has the *Observer* role in the pattern implementation. Any class that is subject to 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. Note 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[MMvD05a] 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.

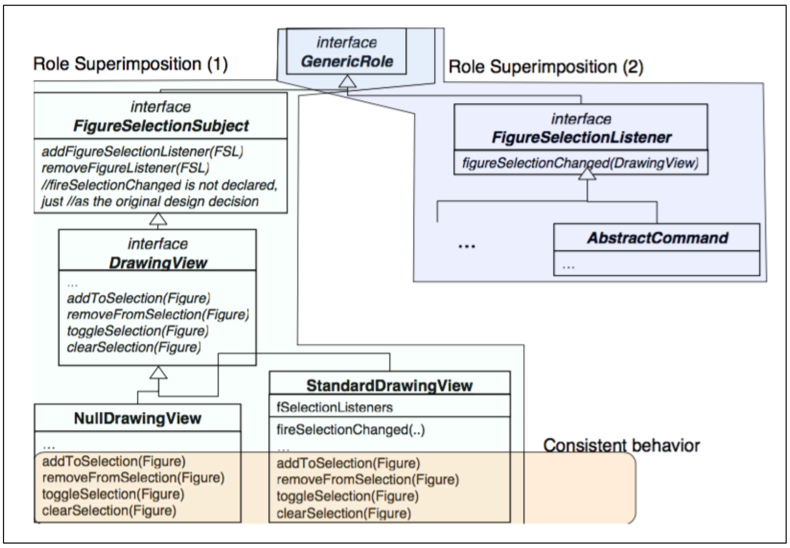

Figure 2.3: AJHotDraw: The concern types in Selection Listener [MMvD05a]

2.5.4 The "Undo" Concern of JHotDraw

Marin et al. have also identified the "Undo" concern in JHotDraw. A number of activities use this functionality including font sizes and colors, image rotation and a lot more. The authors propose the refactoring of the undo concern and more specifically a specific case the *Change Attribute Command* [Mar04].

A general representation of the elements in the JHotDraw implementation of the "Undo" concern can be seen in Figure 2.4.

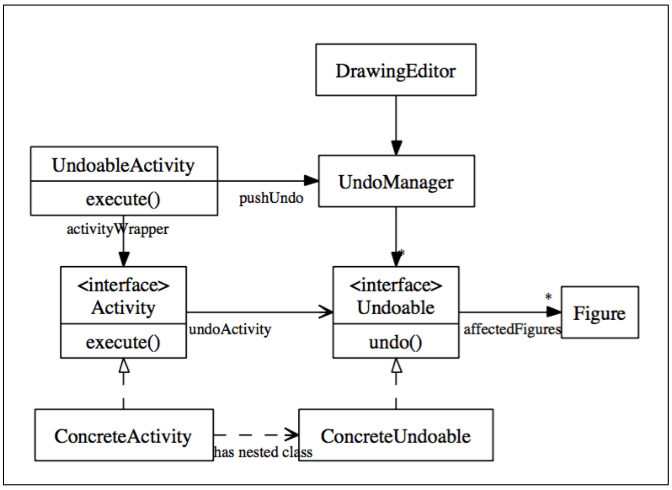

Figure 2.4: Participants for undo in JHotDraw [Mar04]

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 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`.

In particular, this proposition is applied in the `ChangeAttributeCommand` [Mar04]. The undo CCC has been identified, then removed from the system, and finally re-added in an aspect-specific manner.

In this thesis we investigated both the `FigureSelectionListener` and the `ChangeAttributeCommand` (*Undo*) concerns by refactoring them in a new version of the JHotDraw and compare them to their AOP counterpart.

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 of aspect refactoring, we present a showcase. The showcase consists of a very simple state machine application. A similar example is presented in Enso paper as a showcase for its Object Grammar capabilities [SCL+12].

Consider the requirements of the state machine as 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 reasons of 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.

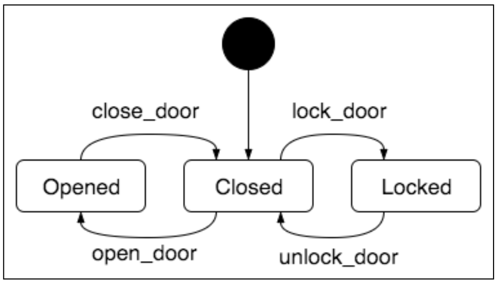

Figure 3.1: Basic door state machine

To implement this we need to define the models, interpret the definition given a list of events and finally add any additional functionality (*concern*) needed, for instance monitor the state of the door.

3.1 Schemas definition

As a first step, all the models of the state machine program need to be defined. An object diagram is illustrated in Figure 3.2.

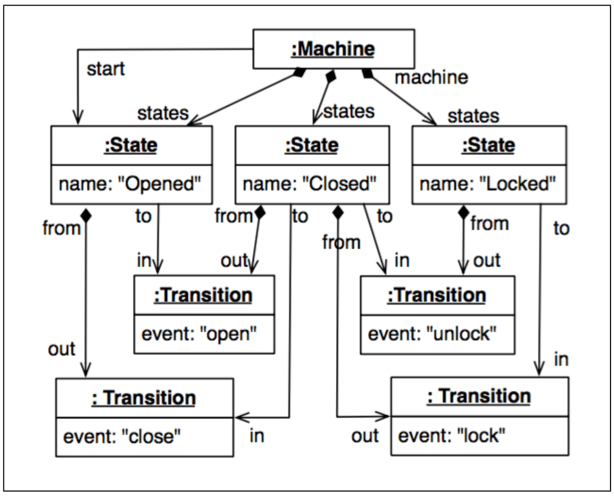

Figure 3.2: Basic door state machine object diagram

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.

```java
public interface Machine extends M {
  State start(State... startingState);

  State current(State... currentState);

  @Contain
  Set<State> states(State... states);
}
```

Listing 3.3: The Machine Schema

As it can be seen in Listing 3.3, the `Machine` schema definition requires 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 suggests that the `states` field is part of the spine tree and it is not a cross-reference. This will be further explained in Chapter 4.

```java
public interface State extends M {
  @Key
  String name(String... name);

  @Inverse(other = Machine.class, field = "states")
  Machine machine(Machine... machine);

  @Contain
  Set<Transition> out(Transition... transition);

  @Contain
  Set<Transition> in(Transition... transition);
}
```

Listing 3.4: The State Schema

For the `State` definition, Listing 3.4, we need a `name` field, which represents the name of the state. This `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`, one field of the `Transition` schema has to be marked as *key*. In this case, it is the `name` field (Line 2 Listing 3.5). Finally, the field `machine` represents the state machine that the state is 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 another schema. In this case, 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, we need an `event` that corresponds to the event of the transition and is the **key** of that schema. The `from` and `to` fields represent the state that the machine changes from and to respectively. However, these are just references to the `State` schema (Listing 3.4).

3.2 Factory definition

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

```
public interface StateMachineFactory {
  Machine Machine();  // constructor for Machine managed objects
  State State();      // constructor for State managed objects
  Transition Transition(); // constructor for Transition managed objects
}
```

Listing 3.6: The StateMachine Factory

3.3 Basic Data Manager

As mentioned above, 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 interpreting a schema definition to instances of *managed objects*. Conclusively, in order to make a *managed object*, the data manager 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 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 interpret the managed object that based on `stateMachineSchema`, shown in Line 6. The basic data manager also supports the field accessors of those data, namely, the setters and getters of their values. An basic interpreter for the state machine is shown in Line 42. As it can be seen, the schema 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 could be *monitoring*, which would log every change in the current state of the state machine. Another concern could be *notification*, which would fire an action when a specific state is set.

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

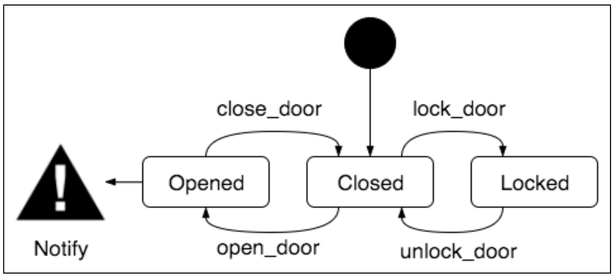

Figure 3.3: Simple door state machine: notify closed door

In order to implement those concerns we need a mechanism that continuously monitors the changes (transitions) of the machine's `current` state and reacts accordingly. Usually, this would lead to scattered *monitoring* and *notification* code in the interpretation method or the models themselves (the machine model). This is where data managers come to the rescue. A data manager can implement concerns as modular aspects without crosscutting 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

Regarding 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 objects*, where observes can be attached in order to be notified 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 one adding a new aspect of data in a modular way.

```java
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: *monitoring* and *notification*. 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.

```java
public class StateMachineMonitoring {
    public static void monitor(Object obj, String field, Object value) {
        if (field.equals("current")) {
            logger.log(" > Current state changed to " + ((State) value).name());
        }
    }

    public static void notify(Object obj, String field, Object value) {
        if (field.equals("current") && ((State) value).name().equals(OPEN_STATE)) {
            if (EmailSender.send("Danger!", "Someone just opened the door!")) {
              logger.notify(" > Danger e-mail sent!.");
            }
        }
    }
}
```

Listing 3.8: Door state machine concerns definition

Since there is an observable data manager and the concerns are implemented in a separate and reusable module, completely unrelated to our logic code, we still need to integrate them in the original code. The integration code is presented in Listing 3.9. The only part that changes is Line 9 of the original code, where 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.

```java
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 solely build managed objects, but the observable data manager also provides the functionality of attaching concerns in the managed objects after a specified event. Concluding, the example presented a modular solution of CCC without scattering and tangling code in the components.

Chapter 4

Managed data in Java

As it has already been mentioned, the programming languages include data definition mechanisms that are predefined. This makes them unable to define CCC without repeating and scattering code through the components [LvdSC12]. Notably, the problem is that CCC are not considered features of the data types, but instead features of data management. 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, testing our first research question, which states *"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 to our implementation's design. 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 allows the programmer to handle the fundamental data manipulation mechanisms using *Data Managers*, one of its distinguishing features being modularity. Using a data description language the programmer defines *Schemas*. *Schemas* are the input of *Data Managers*. A *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. *Schemas* and *Data Managers* are essential components of managed data, along with *Integration* in the programming language, in our case being 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 the one used in Enso. For our implementation we chose to use **Java Interfaces** as a data description language to define records of managed data. By using Java interfaces we use Java's syntax 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 *Schemas*.

As a result, 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 `Fields`, each of which has a name and a `Type`. Since Java interfaces are used to define a `schemaKlass` we need a way to define `Fields` for that `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, considered meta data, that help define the structure of a `Schema`. In order to define the meta data in our data description language (interfaces), we use *Java Annotations*. Annotations are very declarative in the way they express meta data in interfaces and they are consistent with the system (Java).

Thus, to provide 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 by 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 of wrong types it notifies the programmer. Additionally, this is beneficial when a programmer uses IDE's that perform real time type inspection[4]. In those cases errors on the definitions will be spotted immediately.

The list of the available structure concepts that are supported in our language is presented below [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 interpreted 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 can include `null` values. `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 schema, even itself (see Section 4.2). Schema definition examples are presented in Chapter 3 Listings 3.3, 3.4 and 3.5. In those definitions the above concepts can be recognized and their meaning can be revealed in context.

[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[6] for this functionality since we need them to be managed data and not ordinary objects. Thus, we use Java interfaces to define *Schema Factories*. A *Schema Factory* is a list of constructor definitions for specific schema Klasses.

The methods in this interface are used similarly to the constructors in a Java class, 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 regarding constructors with initialization values, making 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[7], the returned values are not ordered the way as defined in the source code. Consequently, since the schema definition is reflectively analyzed in the data managers and is dependent on that order, those methods can not be used in the initialization of values.

However, we overcame this difficulty and were able to support this feature in an alternative manner. In our implementation both the defined methods and the fields are **alphabetically ordered** by name before being initialized.

That feature can be used by the programmer although it can be confusing. Therefore, as an advice, we suggest to either provide constructors without initialization values or to write constructors with only **primitive** initialization values in **alphabetical order**. Otherwise we risk getting values in a random order 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 for interpreting the schema and building virtual objects (managed objects). The managed object's fields are defined by the given schema and acts according to the specifications given by the data manager. Additionally, the data manager ensures that the data given are valid with respect to the schema. More specifically, the data managers describe how a schema definition is handled from the outside world and what its specifications are. These properties may include CCC that can be described separately by special data managers, separating schema and concern definitions. Thus, a managed object can have multiple interpretations based on the data manager that is used to interpret it.

A data manager is initialized with a `Schema` and provides a new `Managed Object` instance whose properties are defined by that data manager. Additional to the `Schema` that includes a Set of `Type`s (`Primitive`s or `Klass`es), it also needs a `Schema Factory` that declares the constructors of the given schema `Klass`. After the initialization of a data manager and the interpretation of the schemas, a data manager provides the mechanism of building new `Schema Factories`, which in turn create `Managed Objects` with the specifications of the data manager.

In the example presented in listing 4.11, Line 3 defines a basic data manager that gets the schema factory and the schema of a state machine as input. Next, Line 7shows a new schema factory being created, which builds managed objects with the specifications attached from the basic data manager. Finally, Line 10 illustrates how the managed object instances with those specifications can be built.

[6] `new` keyword

[7] 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.

```java
// 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 above, we use Java interfaces to define schema Klasses that include fields. Those fields are dynamically discovered by the data manager who has the ability to determine the fields and methods of the managed data object during runtime. In addition, when the data manager adds functionality on a managed object then it first delegates the calls to its specifications and then to the fields of an instance. 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 Handler (`MObject`). 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 delegating actions with their invocation handling mechanisms. Figure 4.1 illustrates this structure. As it can be seen the data manager is a *factory* that has only one exposed method, `make()`, that is used to build a `SchemaFactory`, which in turn builds `MObject` instances.

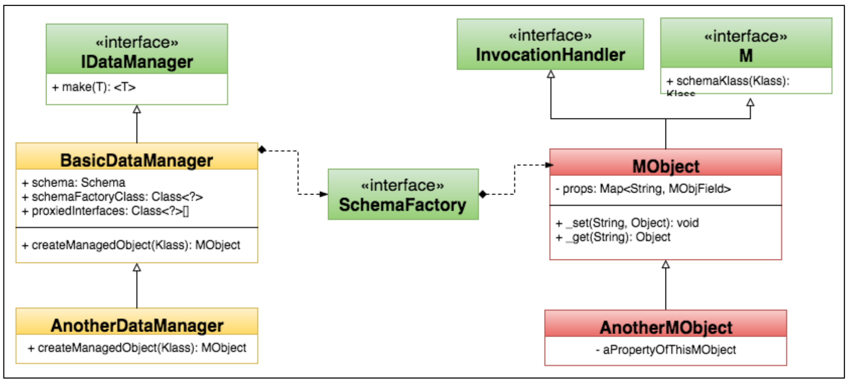

Figure 4.1: Data Manager and MObject

Stacking Data Managers

In order to create a stack of data managers that combine behavior and specifications, we can use inheritance. Figure 4.1 shows how this works. In detail, `AnotherDataManager` extends `BasicDataManager` and only overrides the `createManagedObject()` method. This method is responsible for creating a new instance of an `MObject`. In this case, 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 data manager, leading to the modular aspect of the data managers. As it can be seen, for stacking data managers we used the *Decorator Pattern* [Gam95] which is mentioned also in Cook et al. [LvdSC12] as a strategy for static OOP languages.

4.1.4 Managed Objects

The `MObject`, is an implementation of the `InvocationHandler` interface. Thus, the `MObject`'s `invoke()` method is called in every field access of the managed object's instance. To manipulate 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 `MObjects` in order to *Decorate* the basic `MObject` with their functionality. Of course they require to call their `supers` for running all the checks.

The `MObject` is the *backing object* that stores a reference to the `schemaKlass` and its implementation represents an instance of that `schemaKlass`. That `schemaKlass` is a meta class that describes the layout of the `MObject` and keeps the `Fields` and their `Types`. During construction, the fields of the `MObject` are specified by its `schemaKlass`. When a field check has to be performed, the `MObject` uses its `schemaKlass`. The usage of the `schemaKlass` for setting up 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 4. Inside the `setupField` method the interpretation of the schema is performed. In particular, in Line 11 we check if that field is a multi-value field, and if not, we just set it up as a `Primitive` or a `Klass` accordingly. Consider that the `field.type().schemaKlass().name()` is used like a common `instanceof` in Line 12. In case the field has many values, we first check if it is `Primitive`, since we do not support Set of `Primitives`. Following that, we check if a `Key` field exists on that field's type and in that case this field is a Set, otherwise it is a List.

```
1  public MObject(Klass schemaKlass, Object... initializers) {
2    this.schemaKlass = schemaKlass;
3  
4    this.schemaKlass.fields().forEach(this::safeSetupField);
5    if (initializers != null) {
6      this.safeInitializeProps(initializers);
7    }
8  }
9  
10 protected void setupField(Field field) {
11   if (!field.many()) {
12     if (field.type().schemaKlass().name().equals("Primitive")) {
13       this.props.put(field.name(), new MObjectFieldSinglePrimitive(this, field));
14     } else {
15       this.props.put(field.name(), new MObjectFieldSingleMObj(this, field));
16     }
17   } else {
18     if (field.type().schemaKlass().name().equals("Primitive")) {
19       this.props.put(field.name(), new MObjectFieldManyList(this, field));
20     } else {
21 
22       Klass klassType = (Klass) field.type();
23       if (klassType.key()!= null) {
24         this.props.put(field.name(), new MObjectFieldManySet(this, field));
25       } else {
26         this.props.put(field.name(), new MObjectFieldManyList(this, field));
27       }
28     }
29   }
30 }
```

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 components of a new data manager implementation are 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 useful is a data manager that introduces immutability to 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 specifications 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.

```java
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, inherits the basic functionality of a managed object 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.

```java
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 explained 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 interfaces used 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 also represented as managed data. 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 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`, without `Fields`, and a `Klass`, with a set of `Fields`. Additionally, those `Fields` may have some extra meta data attributes that are explained in Section 4.1.1.

A schema like this can describe itself since every concept used in the explanation is de facto 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.

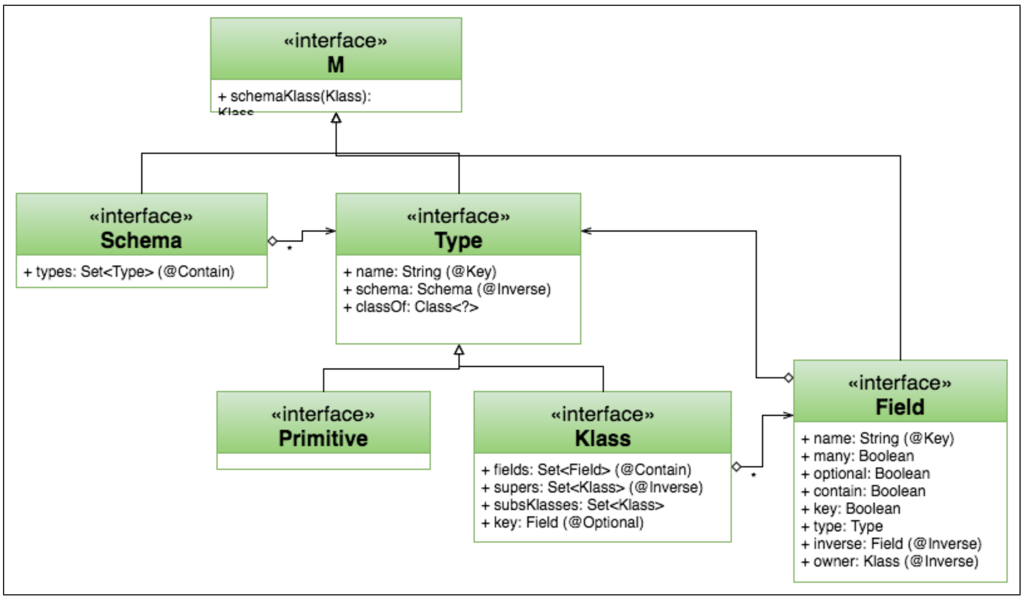

Figure 4.2: The schema of schemas

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 with the normal schemas, we use a schema factory. However, this time it is a *schemaSchemaFactory* that defines constructors of all the schema klasses that are needed to describe our *SchemaSchema*. Listing 4.17 shows its definition.

```java
public interface SchemaFactory {
    Schema Schema();
    Primitive Primitive();
    Klass Klass();
    Field Field();
    Field Field(
      Boolean contain, Boolean key, Boolean many, String name, Boolean optional);
}
```

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 an 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.

```java
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 will be explained further in Section 4.3. 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 it has been given. A simple description of that process is shown in Listing[8] 4.19. As illustrated we first implement the instances and following that we use setters to wire them up. The reason for this is that not everything exists at the time that it needs to be set.

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

```java
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 our implementation.

4.3 Bootstrapping

Considering that SchemaSchema is managed data itself, 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 exclusively self-describing, as it must manage itself [LvdSC12], and hardcoded in its own class, `BootSchema`.

4.3.1 Cutting the umbilical cord

Having a `BootSchema` in place we can now create "real" `SchemaSchemas` [9]. 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 no longer need the `BootSchema`, which leads to a process that we call "Cutting the umbilical cord". An example of "Cutting the umbilical cord" is shown in Listing 4.3.1, where 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,
         Primitives.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, making the Schema Schema independent and managed data itself. Thus, it can be used to create other schemas like the Machine schema or even itself.

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

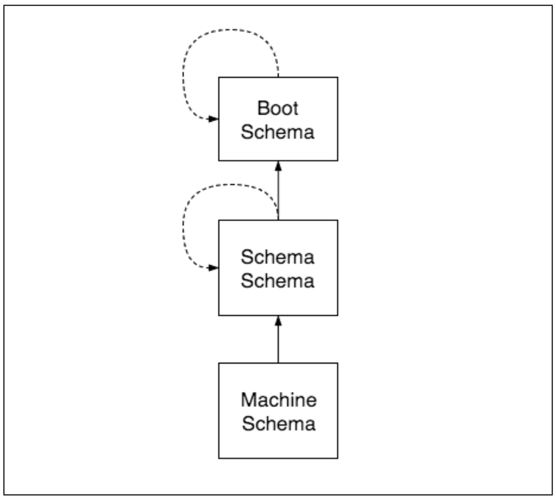

Figure 4.3: Boot Schema models

4.3.2 Primitives Definition

Since the Bootstrap Schema defines the primitive types for its description, the real schema schema needs a way to include them as well. These initial Java primitives supported in our implementation are shown in Table 4.1.

	Class<?>	Name	Default Value
Integer	Integer.class	“Integer”	0
int	int.class	“int”	0
Boolean	Boolean.class	“Boolean”	false
boolean	boolean.class	“boolean”	false
String	String.class	“String”	“”
Double	Double.class	“Double”	0.
Float	Float.class	“Float”	0.f
Class	Class.class	“Class”	null
Object	Object.class	“Object”	null

Table 4.1: Primitives Table

To define those primitives we use an interface called *Primitives*, introduced during the loading of the real schema, as seen in Line 19 of Listing 4.20. The definition of this interface is shown in Listing 4.3.2 which is a simple `Class/Name` mapping [10].

```java
public interface Primitives {
  Integer Integer();
  int _int();
  Boolean Boolean();
  boolean _boolean();
  String String();
  Class Class();
  Float Float();
  Double Double();
}
```

Listing 4.21: Primitives Definition

[10] We use the “_” prefix convention in order to define names of primitives that are reserved words in Java.

The benefits of such a definition is that the `Primitives` interface is extensible. By extending it one can add more primitives in the schema as long as it is introduced during the schema loading.

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, apart from the `bootstrapSchema`, they are managed data and not normal Java objects, we need a way 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 managed objects, we have omitted it. We do not depend on the ordinary `hashCode` for managed objects, we do not call it and therefore we have not implemented it. If it is a collection field type, then the field has to have a `Key` field. In this case, we obtain the value of the key field and index it into a `HashMap`.

Using the `Key` field as the key of the hashmap works whether 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 the `hashCode` method to the real object so that it would never 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[11], default methods enable the programmer to add new functionalities to the interfaces and can be used as method implementation in abstract classes. We use Java 8 default methods in order to add functionality to our schema definitions. 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 the default method invocation in the MObject is `protected`, which makes it possible for the derived data managers to “monitor” when a default method is invoked.

4.5 Benefits and Limitations

One of the advantages of this language is the simplicity of its usage. A programmer simply 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

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

uses Java concepts, which makes it safer in terms of type checking and definitions making it easier for Java developers to adapt. Furthermore, by being a self-describing language it is no longer bounded to the Java constructs transforming everything into managed data. Finally, the effortless mechanism of stacking data managers makes it significantly modular on every level, meta or not.

However, in addition to the implementation issues described in the previous section, there are significant performance implications since we use Java reflection and dynamic proxies to dynamically interpret the schemas. This makes it unfavorable for applications that focus on performance and are based on JVM optimizations.

Another issue that arises is that integration in existing systems is complicated considering every model has to be redefined as a schema and every functionality has to be 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 programmer control over fundamental mechanisms of data creation and manipulation [LvdSC12]. Those mechanisms are traditionally predefined by the programming languages. Managed data gives control over them by using data managers. Moreover, we claim that managed data introduces a modular way to define data and aspects of data. In Chapter 5 we present how to *aspect refactor* an existing application using managed data.

Chapter 5

Taming Aspects of JHotDraw with managed data

5.1 Crosscutting Concerns Identification

Our managed data framework addresses the problem of CCC by capturing them in modular data managers. Yet, to solve the problem of CCC it is first required to identify them in the source code. This leads to a process called *aspect mining*. *Aspect mining* is a reverse engineering process that aims at finding CCC in existing systems [MVDM04]. The aspect mining topic has been addressed in previous research that include methods such as clone detection [BVDVET05], machine learning [SGP04], IDE tools [RM02] and more. Marin et al. [MVDM04] introduced a technique constructed by spotting methods that are invoked from many different places (high fan-in), in order to identify candidate aspects in open-source Java systems. One of these projects include the JHotDraw. In this thesis we focused on their concern findings in refactoring JHotDraw. In particular, we focused on the *FigureSelection*, concern, which is an observer pattern implementation.

5.2 Aspect Refactoring in Managed Data

In order to evaluate the ability of managed data to tame aspects, we have refactored the aforementioned concerns of JHotDraw. More specifically, in this chapter we present the refactoring of the *FigureSelectionListener* observer pattern. The choice of those concerns has been made on purpose, since those are the concerns that AJHotDraw refactors using AspectJ and AOP techniques. For the refactoring we used our implementation of managed data in Java, presented in the previous chapter. Therefore, by having three versions of the same application (JHotDraw) and by solving the same concerns we will be able to perform a comparative evaluation. The three systems included in our assessment are: **JHotDraw**[1], the original OOP version, **AJHotDraw**[2], the AOP refactored version and our **ManagedDataJHotDraw**[3], the managed data refactored version. We focused on those concerns because they were also identified, solved, analyzed and presented in AJHotDraw. Note that, for compatibility and comparison reliance, we used the version *JHotDraw v.5.4b1* since AJHotDraw also refactors the same version.

In order to refactor JHotDraw, we first had to migrate it in managed data. The result of this migration is available on an open-source project, the ManagedDataJHotDraw. We claim that this is the first aspect refactoring of an application using managed data to date, since this project aims on showing how managed data can deal with CCC in existing systems.

[1] http://www.jhotdraw.org/
[2] https://sourceforge.net/projects/ajhotdraw/
[3] https://github.com/TheolZacharopoulos/ManagedDataJHotDraw

5.3 Migration Process

The refactoring of an application of JHotDraw's size required a significant amount of time to study and familiarize with, yet, its well-designed OOP code, made it easy to grasp. We solely focused on the parts that were going to be refactored, based on refactorings that AJHotDraw developers [MM] performed. Thanks to their fan-in analysis [MVDM04], we targeted the same concerns in order to make a fair comparison. Furthermore, during the implementation of ManagedDataJHotDraw we focused on maintaining behavioral coherence and the original design.

5.3.1 DrawingView

One of the main components of JHotDraw is the *DrawingView* interface. As Figure 5.1 illustrates, the *DrawingView* is responsible for rendering `Drawings` and listening to its changes. Additionally, it is responsible for receiving the user input and delegating it to the current tool.

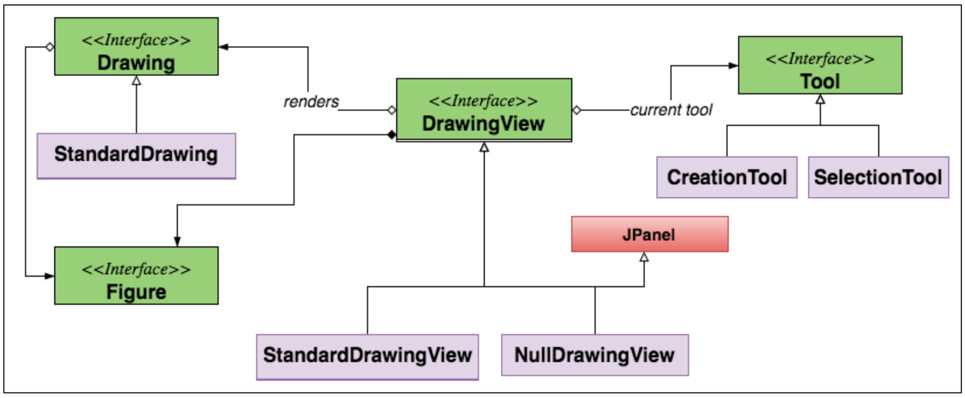

Figure 5.1: DrawingView of JHotDraw

Conclusively, `DrawingView` makes a good candidate for managed data migration. The reason is that the specifications of that class can be implemented in data managers and dynamically added to it.

5.3.2 Managed Data DrawingView

To support sub-typing on the `DrawingView` interface, we have implemented the `MDDrawingView`, namely Managed Data DrawingView, which replaced the `DrawingView` in JHotDraw. Having this interface for super type, we still needed the actual managed data schemas. As Figure 5.1 shows, there are two implementations of the `DrawingView`. In particular, the `StandardDrawingView`, which is the implementation that is used when a new drawing view is created in the application and the `NullDrawingView`, which represents a null drawing view as for the *null-object* pattern.

Following their original design, we have implemented two schemas, one for the `StandardDrawingView` and one for the `NullDrawingView`, namely `MDStandardDrawingView` and `MDNullDrawingView` respectively. The instances of those schemas have been used in the same way their counterparts are used in JHotDraw. A snippet of the `MDStandardDrawingView` is shown in Listing 5.22 [4].

[4]Most of the implementation has been omitted for brevity.

```java
1  public interface MDStandardDrawingView extends M, MDDrawingView {
2    ...
3    // Composition over inheritance, the original inherits the JPanel
4    JPanel panel(JPanel... panel);
5
6    default JPanel getPanel() {
7        return panel();
8    }
9
10   default void setPanel(JPanel _panel) {
11       panel(_panel);
12   }
13   ...
14   Rectangle damage(Rectangle... damage);
15   Drawing drawing(Drawing... drawing);
16   ...
17   default FigureEnumeration selectionZOrdered() {
18     List result = CollectionsFactory.current().createList(selectionCount());
19     FigureEnumeration figures = drawing().figures();
20
21     while (figures.hasNextFigure()) {
22       Figure f= figures.nextFigure();
23       if (isFigureSelected(f)) {
24         result.add(f);
25       }
26     }
27     return new ReverseFigureEnumerator(result);
28   }
29   ...
30   default void repairDamage() {
31     if (getDamage() != null) {
32       panel().repaint(damage().x, damage().y, damage().width, damage().height);
33       setDamage(null);
34     }
35   }
36   ...
37 }
```

Listing 5.22: MDStandardDrawingView schema

Listing 5.22 shows that the `MDStandardDrawingView` interface extends both `M` interface, defining that this is a schema definition, and `MDDrawingView`, for sub-type support. Additionally, all the functionalities implemented in methods of the original `DrawingView`, in managed data they are implemented in default methods of the schema interface. The fields of a schema can provide those methods with the managed object's current state. As Lines 17 and 30 show, the fields of the schema can be used to query their values inside the default methods. Note that the code in the default methods is identical to the original `DrawingView`. Furthermore, for consistency with the legacy code, we have implemented setters and getters, Lines 10 and 6, for field values accessors. This way we maintained consistency across in accessing values of the managed object.

A notable issue is that the original `StandardDrawingView` extends the `javax.swing.jpanel` class as Figure 5.1 shows. However, such a structure is not supported in managed data. Schema definitions can not extend classes. To overcome this issue we defined the `JPanel` as a field in the schema, namely *panel*. To support the `JPanel` as a type of a field though, it is needs ti be defined as managed data. By all means, the same holds for the remaining fields, such as `Rectangle` and `Drawing`.

As explained in Section 4.3.2, our framework provides external primitives definition by inheriting the `Primitives` interface. The JHotDraw primitives definition is shown in Listing 5.23.

```
public interface JHotDrawPrimitives extends Primitives {
  javax.swing.JPanel JPanel();

  java.awt.Color Color();
  java.awt.Cursor Cursor();
  java.awt.Point Point();
  java.awt.Dimension Dimension();
  java.awt.Rectangle Rectangle();

  CH.ifa.draw.framework.DrawingEditor DrawingEditor();
  CH.ifa.draw.framework.Drawing Drawing();
  CH.ifa.draw.framework.Painter Painter();
  CH.ifa.draw.framework.PointConstrainer PointConstrainer();

  ...
}
```

Listing 5.23: JHotDraw Primitives Definition

This has been proven very helpful since we did not need to re-implement every field as managed data during the refactoring. Especially, classes that are provided by libraries such as `javax.swing` and `java.awt`.

Limitations

However, extending our framework's primitives with the JHotDrawPrimitives we lost the "pureness" of managed data. That led to an application that partly managed data. Generally, this may be the case when refactoring big applications like JHotDraw.

Another limitation is that some Java keywords such as "synchronized" can not be supported on default methods. Instead, as future work, we could use annotations that define these properties to the default methods and add them during the interpretation of the schemas. Moreover, privacy and visibility is another an issue. All default methods are `public`, which means that the encapsulation is violated. Finally, private classes definition is not possible inside schemas, although they can be defined outside as managed data.

5.3.3 MDDrawingView Schema Factories

In order to create instances of the defined `MDStandardDrawingView` and `MDNullDrawingView` schemas, we needed their factories. Besides the schema factories, which is as simple as Listing 5.24 shows, we still needed a way to give initialization values to the schema instances the same way that the original `StandardDrawingView` does during construction. Additionally, this factory should be used like Java's `new` keyword in the source code. This factory just replicates the original `StandardDrawingView` constructor and is used from the program to create new instances of the schemas. The code of the `MDStandardDrawingView` *instances factory* is illustrated in Listing 5.3.3, in comparison to the original constructor, illustrated in Listing 5.3.3.

```java
public interface DrawingViewSchemaFactory {
  MDStandardDrawingView DrawingView();
  MDNullDrawingView NullDrawingView();
}
```

Listing 5.24: DrawingView Schema Factory

```java
public StandardDrawingView(DrawingEditor editor, int width, int height) {
  setAutoscrolls(true);
  fEditor = editor;
  fViewSize = new Dimension(width,height);
  setSize(width, height);
  fSelectionListeners = CollectionsFactory.current().createList();
  addFigureSelectionListener(editor());
  setLastClick(new Point(0, 0));
  fConstrainer = null;
  fSelection = CollectionsFactory.current().createList();
  setDisplayUpdate(createDisplayUpdate());
  setBackground(Color.lightGray);
  addMouseListener(createMouseListener());
  addMouseMotionListener(createMouseMotionListener());
  addKeyListener(createKeyListener());
}
```

Listing 5.25: Original StandardDrawingView Constructor

```java
public static MDDrawingView newDrawingView(
  DrawingEditor editor, int width, int height) {
  final MDStandardDrawingView drawingView = drawingViewSchemaFactory.DrawingView();
  MyJPanel jPanel = new MyJPanel();
  jPanel.setAutoscrolls(true);
  jPanel.setSize(width, height);
  jPanel.setBackground(Color.lightGray);
  drawingView.panel(jPanel);
  jPanel.setDrawingView(drawingView);

  drawingView.editor(editor);
  drawingView.size(new Dimension(width, height));
  jPanel.setSize(width, height);
  drawingView.lastClick(new Point(0, 0));
  drawingView.constrainer(null);
  drawingView.setDisplayUpdate(new SimpleUpdateStrategy());
  drawingView.setBackground(Color.lightGray);
  drawingView.drawing(new StandardDrawing());

  jPanel.addMouseListener(...);
  jPanel.addMouseMotionListener(...);
  jPanel.addKeyListener(...);
  return drawingView;
}
```

Listing 5.26: MDStandardDrawingView Instances Factory

5.3.4 MDDrawingView Integration

Finally, in order to integrate the `MDDrawingView` managed objects in the existing system, first we had to replace every instance of `DrawingView` with `MDDrawingView`, every `StandardDrawingView` with `MDStandardDrawingView` and every `NullDrawingView` with `MDNullDrawingView` accordingly. In addition, everywhere a new instance of these is created, we replaced it with our *instances factory*.

5.4 Aspect Refactoring of JHotDraw

Aspect refactoring usually refers to the refactoring of legacy code in aspect oriented code. However, in this section we present an aspect refactoring of JHotDraw legacy code in managed data.

5.5 FigureSelectionListener: Observer Pattern

The `FigureSelectionListener` observer pattern of JHotDraw is a concern first presented by Hannemann et al. [HMK05] in their role-based refactoring of design patterns in AspectJ. Later, Marin et al. used the same concern and migrated it into their AJHotDraw implementation [MMvD05a]. Likewise, we have also implemented the same aspect for our refactoring in order to compare our aspect solution with the existing one.

5.5.1 FigureSelectionListener in JHotDraw

The original `FigureSelectionListener` observer pattern of JHotDraw is illustrated in Figure 5.2.

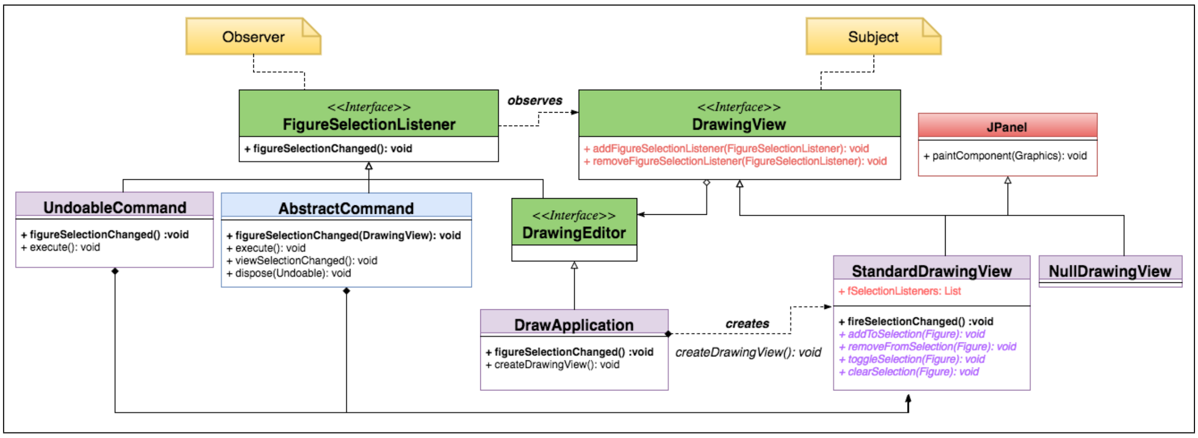

Figure 5.2: FigureSelectionListener in JHotDraw

As this figure illustrates, the `FigureSelectionListener` interface defines the *Observer* role. The classes that are interested in the changes of selection of figures in a `DrawingView` implement this interface. Accordingly, the `DrawingView` defines the *Subject* role, providing methods for adding and removing figure selection listeners. Practically, the only class that implements the *Subject* role is `StandardDrawingView`, while `NullDrawingView` has an empty implementation.

`StandardDrawingView` keeps the selection listeners in a list, the `fSelectionListeners`, and notifies them in the invocation of the `fireSelectionChanged` method. This method is called in the methods: `addToSelection`, `removeFromSelection`, `toggleSelection` and `clearSelection`, which indicate the change of figure selection. On the observers' side, the figure selection listeners implement the `figureSelectionChanged` method that is executed in case they have been notified by the subject.

Concluding, as described above, the "pattern code" of the observer pattern is scattered in many places, including the list of listeners on the subject, the add / remove methods, along with the pointcut methods that call the method which notifies the listeners.

5.5.2 Refactoring FigureSelectionListener in AJHotDraw

Marin et al. presented a refactoring of this concern in AJHotDraw [MMvD05a]. Their refactoring is illustrated in Figure 5.3.

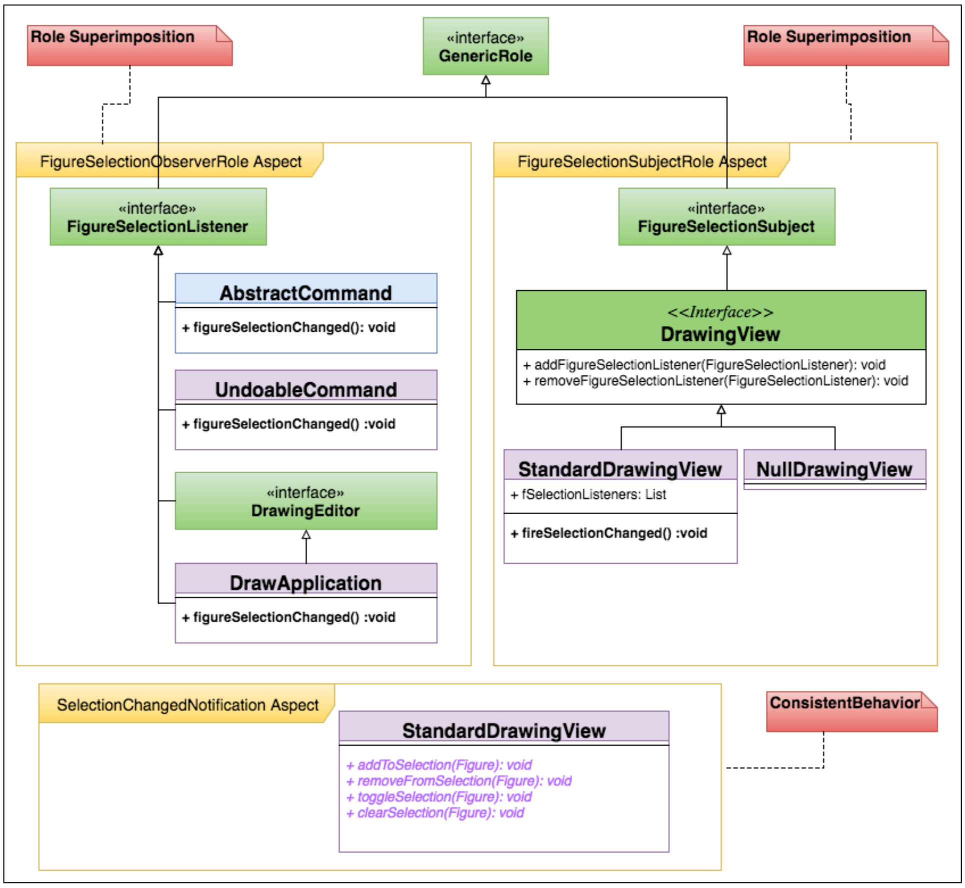

Figure 5.3: FigureSelectionListener in AJHotDraw

In their proposed type-based refactoring, they have used two crosscut sorts, namely *role superimposition* and *consistent behavior*.

Role Superimposition

As defined by the authors [MMVD05b], "the role superimposition refers to the implementation of a specific secondary role or responsibility". In the case of `FigureSelectionListener`, they used it twice, one for each of the roles. More specifically, they defined an abstract `GenericRole` and concrete roles, observer and subject which extend the abstract one.

Consistent Behavior

According to the authors[MMVD05b], "the consistent behavior sort implements a consistent behavior for a number of method elements that can be captured by a natural pointcut". In this case it is used to notify the *Observers* of the changes in the *Subject* object. More specifically, the methods `addToSelection`, `removeFromSelection`, `toggleSelection` and `clearSelection` are consistent behavior. They implement it as a pointcut in AspectJ shown in Listing 5.27.

```
public aspect SelectionChangedNotification {
  pointcut invalidateSelFigure(StandardDrawingView sdw) :
    (   withincode(boolean StandardDrawingView.addToSelectionImpl(Figure))
     || withincode(void StandardDrawingView.removeFromSelection(Figure)))
    && call(void Figure.invalidate())
    && this(sdw);

  pointcut clear_toggleSelection(StandardDrawingView sdw):
    (execution(void StandardDrawingView.clearSelection()) ||
     execution(void StandardDrawingView.toggleSelection(Figure)))
    && this(sdw);

  after(StandardDrawingView sdw): invalidateSelFigure(sdw) {
    sdw.fireSelectionChanged();
  }

  after(StandardDrawingView sdw): clear_toggleSelection(sdw) {
    sdw.fireSelectionChanged();
  }
}
```

Listing 5.27: AJHotDraw: Consistent Behavior in FigureSelectionListener

Benefits and Limitations

According to the authors [MMvD05a], such refactoring allows the crosscutting elements to be addressed individually, which leads to a modular solution, and any deviations from the pattern implementation can be addressed separately.

However, as they mention, the definition of pointcuts in order to capture the calls to the notifier is difficult when many consistent behavior instances occur. As Listing 5.28 shows, the original `clearSelection` method in JHotDraw calls `fireSelectionChanged` under specific conditions. Considering the AOP solution of AJHotDraw, Listing 5.27, this is not the case. In the pointcut definition, the pattern refactoring solution notifies the observers independently of the condition in the caller. Although, according to Marin et al. it is potentially harmless in this case, this implementation deviates from the behavior of the original JHotDraw, leading to a harmful, for the functionality, implementation. Finally, the problem of the unconditional call of a method in a pointcut is clearly a problem of the language. AspectJ mechanisms do not support such functionality. But can managed data solve this problem?

```
public void clearSelection() {
  if (selectionCount() == 0) {
    // avoid unnecessary selection changed event when nothing has to be cleared
    return;
  }
  FigureEnumeration fe = selection();
  while (fe.hasNextFigure()) {
    fe.nextFigure().invalidate();
  }
  ...
  fireSelectionChanged();
}
```

Listing 5.28: StandardDrawingView clearSelection Method

5.5.3 Refactoring FigureSelectionListener in ManagedDataJHotDraw

In JHotDraw's original code, the *observer* `DrawApplication` creates a new `StandardDrawingView` instance using the `createDrawingView` method. During the construction the `DrawApplication` passes itself to the constructor of the `StandardDrawingView` and this in turn adds it to its listeners list, using the `addFigureSelectionListener` method. This is shown in Line 7 of Listing 5.3.3. Likewise, the rest of the classes that implement the `FigureSelectionListener` interface, perform the same mechanism, adding or removing themselves from the `DrawingView` *Subject*. Consequently, the pattern code is scattered among its participants.

In this section we present our managed data refactoring of the `FigureSelectionListener` concern. Managed data implements aspects using data managers, by adding specifications to the data. For this case, we needed a similar mechanism to the *role superimposition* of AOP. This mechanism should be defined in a data manager that will produce managed data instances (managed objects) with a specific role. Additionally, the data manager has to support something similar to the *consistent behavior* as a pointcut.

In detail, since the `DrawingView` is managed data, and it is the *Subject* to the *Listeners* of the *FigureSelectionListener* case, we can implement a data manager that attaches the *Subject* `MDStandardDrawingView`. Therefore, no *Subject* role specific code will be tangled with the `DrawingView`, but a data manager will attach this role later.

More specifically, we needed a data manager that performs the following:

1. Attaches the *Subject* role to the `MDStandardDrawingView` since this object implements the pattern. Initially, `MDStandardDrawingView` has no *Subject* role related code.
2. Enables the *Subject* to *add* and *remove* listener objects to and from itself. In this case the `FigureSelectionListener` instances.
3. Defines an *Action* that will be executed on the listeners in case of the *Subject*'s notification. In this case the `figureSelectionChanged` method.
4. Finally, it defines a pointcut for the consistent behavior that executes the actions on the listeners. In this case the `addToSelection`, `removeFromSelection`, `toggleSelection` and `clearSelection` methods.

Data manager

First, we can abstract the *Subject* role concern code in a separate data manager. As it is mentioned in Chapter 4, the role of a data manager class is to create a `MObject`, which interprets and handles a managed object instance. Therefore, we first need this `MObject`, namely `SubjectRoleMObject`, to implement our subject role specifications.

SubjectRole specifications

We define the functionality of the *SubjectRole* in an interface, shown in Figure 5.29. The subject role simply needs to add and remove listener objects to and from a managed object.

```
public interface SubjectRole {
  void addListener(Object listener, Action action);
  void removeListener(Object listener);
}
```

Listing 5.29: SubjectRole Interface

Action

Additional to the listener object a *SubjectRole* has to define an *Action* for that listener. That *Action* determines the method which will be executed in that *Listener* in case a notification is retrieved from the *Subject*. As Listing 5.30 shows, this is simply a functional interface that represents an executable action.

```java
@FunctionalInterface
public interface Action {
  void execute();
}
```

Listing 5.30: Action Interface

SubjectRole MObject

Having the specifications of a Subject in place we need a data manager that implements them. This data manager defines a "role superimposition" of the subject role.

Role Superimposition

The implementation of the `SubjectRoleMObject` is presented in Listing 5.31. First, the SubjectRole-MObject extends the MObject, inheriting the functionalities of the base data manager, followed by the implementation of the `SubjectRole` specifications. By implementing the `SubjectRole` interface, the MObject has to implement the `addListener` and `removeListener` methods that have been provided by the subject role specifications. Having added a listener object along with its `Action` to be executed on each notification, the method `executeListenerActions` executes all the actions for each of the listeners.

```java
public class SubjectRoleMObject extends MObject implements SubjectRole {

  protected Map<Object, Action> listeners;

  public SubjectRoleMObject(Klass schemaKlass, Object... initializers) {
    super(schemaKlass, initializers);
    listeners = new HashMap<>();
  }

  protected void executeListenerActions() {
    listeners.values().forEach(Action::execute);
  }

  @Override
  public void addListener(Object listener, Action action) {
    listeners.put(listener, action);
  }

  @Override
  public void removeListener(Object listener) {
    listeners.remove(listener);
  }
  ...
}
```

Listing 5.31: SubjectRoleMObject

Consistent Behavior

Since we have implemented a form of *role superimposition* what is left is the *consistent behavior* pointcut. However, this functionality is application specific, therefore we can extend the abstract

`SubjectRole` data manager with an application specific data manager that implements the consistent behavior. For practical reasons, we used an interface to define these specifications. As Listing 5.32 shows, a list of the methods that execute the `Action` for each listener are defined in the `FigureSelectionPointcut` interface.

```
public interface FigureSelectionPointcut {
  void addToSelection(Figure figure);
  void removeFromSelection(Figure figure);
  void toggleSelection(Figure figure);
  void clearSelection();
}
```

Listing 5.32: FigureSelectionPointcut Interface

FigureSelectionListenerSubjectRole MObject

Finally, having all of our specifications in place, we need to implement the actual MObject that uses them. Of course we need a new data manager that is application specific, namely FigureSelectionListenerSubjectRole. In particular, the FigureSelectionListenerSubjectRole MObject implements those specifications and provides its managed objects with the ability to use them. Additionally, this data manager extends the `SubjectRole` data manager, creating a stack of data managers. By stacking the data managers we have separated the application specific code, the consistent behavior in this case, from the more general subject role code.

Consistent Behavior Pointcut
Considering that an MObject is an Invocation Handler, every method invocation passes through that object first. By defining the pointcut in an interface and extending that interface in this MObject, we proxy the execution of the real object's methods, starting with MObject first. This allows the programmer to add functionalities in these methods which in other cases would scatter the real object's methods. Similarly to the AOP solution, the pointcut includes the three methods that call the `fireSelectionChanged` method. However, in managed data, we are not limited to a specific method of a class but to an `Action` passed for the specific listener. Invoking the `executeListenerActions` method on each of the methods defined by the pointcut, we have implemented the concern as a modular aspect. Listing 5.33 illustrates the code of the FigureSelectionListenerSubjectRole data manager.

Conditions in Pointcuts
As it has been seen from the AOP solution, during the pointcut definition, the language did not allow to add any kind of conditions or other functionalities based on the state of the object. However, since the MObject is proxied to a `MDStandardDrawingView` instance, the programmer can access the current state of the instance inside the data manager implementation. Therefore, the programmer can use the state of the program. In this case, the execution of the action on the listener is performed under a specific condition, (Line 27 of Listing 5.33), which is similar to the one defined on the original program 5.28.

```java
public class FigureSelectionListenerSubjectRoleMObject
  extends SubjectRoleMObject implements FigureSelectionPointcut {

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

  @Override
  public void addToSelection(Figure figure) {
    executeListenerActions();
  }

  @Override
  public void removeFromSelection(Figure figure) {
    executeListenerActions();
  }

  @Override
  public void toggleSelection(Figure figure) {
    executeListenerActions();
  }

  @Override
  public void clearSelection() {
    if (((MDStandardDrawingView) thisObject).selectionCount() > 0) {
      executeListenerActions();
    }
  }
}
```

Listing 5.33: FigureSelectionListenerSubjectRoleMObject

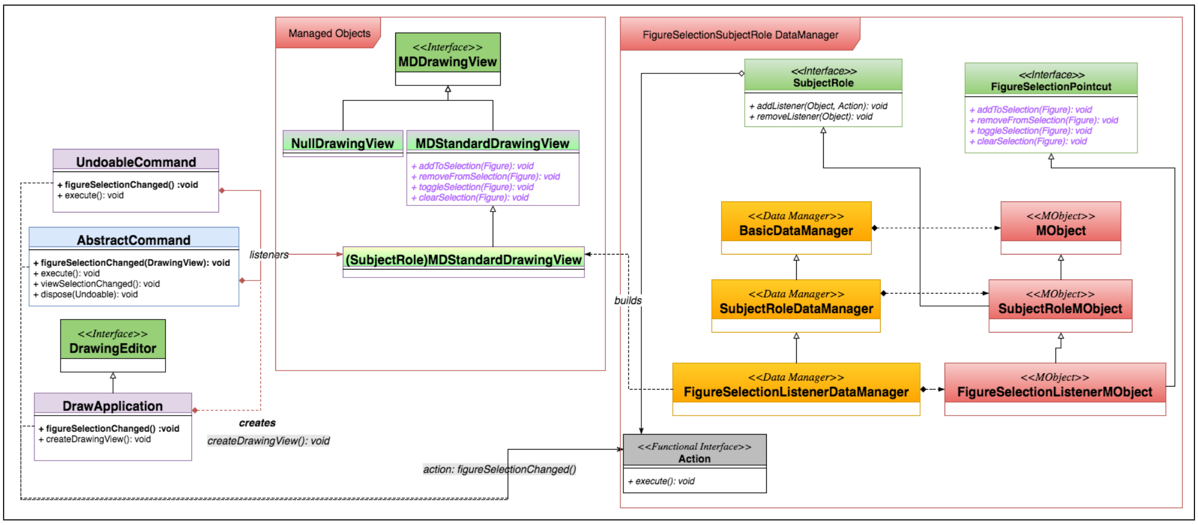

Figure 5.4: FigureSelectionListener in ManagedDataJHotDraw

5.5.4 FigureSelectionListener Results

Figure 5.4 illustrates the refactored version of the FigureSelectionListener concern in ManagedDataJHotDraw. Comparing it to the original, Figure 5.2, it can be seen that, first, the list of listeners has been removed from the `DrawingView`. Next, the `addListener` and `removeListener` methods have also been removed from the class. Every call of the `fireSelectionChanged` method in the pointcut methods has also been omitted. Finally, conditions on the pointcuts have been defined, something that is not supported by the AOP version, AJHotDraw.

The integration of the data manager was executed simply by using our schema factories and adding the listeners during construction. Most importantly, the behavior of the application remained equivalent to the original. ManagedDataJHotDraw conserved the behavior of JHotDraw, which we evaluated through its own test suite along with manual tests. Interestingly, by stacking data managers we manage to define aspects of our data in a modular way.

5.6 ChangeAttributeCommand: Undo Concern

The "Undo" functionality is used in several places in the original JHotDraw. Marin's fan-in analysis [MVDM04], identified about 30 undo activities defined for various elements of JHotDraw. For our assessment we focused on the refactoring of the undo concern in the *ChangeAttributeCommand* class. We choose the specific case since is the same that is used by Marin et al. on their undo refactoring in AJHotDraw [Mar04].

5.6.1 ChangeAttributeCommand in JHotDraw

The original Undo concern in *ChangeAttributeCommand* of JHotDraw is illustrated in Figure 5.5.

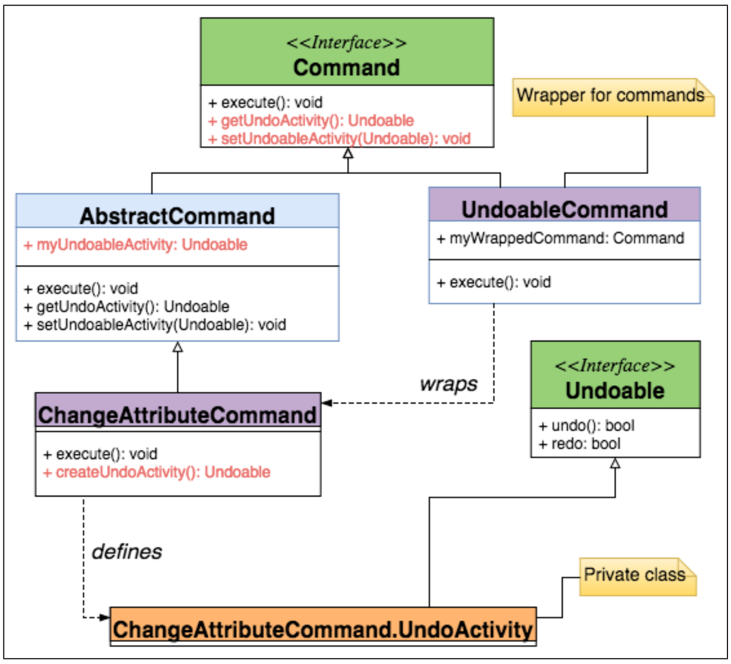

Figure 5.5: ChangeAttributeCommand in JHotDraw

A command of the *Command* pattern is represented by the `Command` interface in the figure. Some of the activities support the undo functionality, which is implemented in nested `UndoActivity` classes.

In the case of the *ChangeAttributeCommand*, the command is called when an attribute is applied to a figure. An attribute can be a color, a font, a url etc. When an attribute has been applied using a *ChangeAttributeCommand* object, the object defines its `UndoActivity` through the *UndoActivity* private class.

The role of the `UndoableCommand` wrapper class is to support repeated undo operations since it records the last executed commands of the wrapper class, in reverse order. In particular, this class acquires a reference to the undo activity associated with the wrapped command and it pushes it into a stack managed by an `UndoManager` [Mar04].

Therefore, as Marin et al. concluded [MVDM04] the "Undo" concern code is scattered in several places of the Command classes. First, the `myUndoableActivity` field in the `AbstractCommandClass` along with its accessors, `getUndoActivity` and `setUndoActivity`. Next, the private nested classes are implemented by the concrete commands that support undo. Moreover, the factory method, `createUndoActivity` creates instances of the private classes. Finally, the references to the before enumerated elements from non-undo related members.

5.6.2 Refactoring ChangeAttributeCommand in AJHotDraw

The refactoring that Marin et al. proposed can be seen in Figure 5.6.

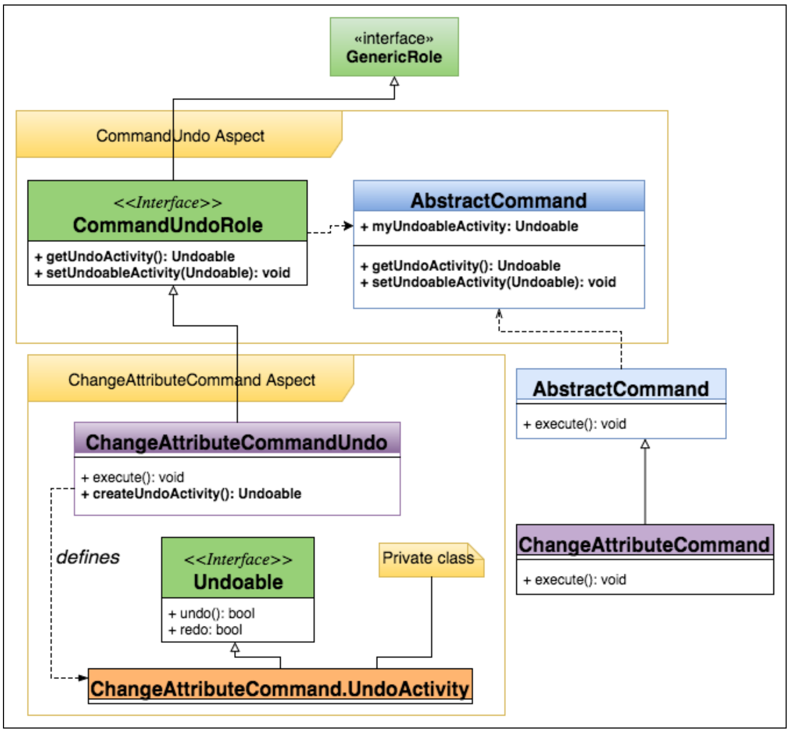

Figure 5.6: ChangeAttributeCommand in AJHotDraw

As the figure shows, a new aspect is created for the *ChangeAttributeCommand*. In this aspect the entire undo functionality is implemented and the undo code is removed from the actual *ChangeAttributeCommand* class. Additionally, the private class that implements the *UndoActivity* has moved to this aspect along with its factory method (*createUndoActivity*).

However, by convention, each aspect will consistently be named by appending "UndoActivity" to the name of its associated command class, to enforce the relation between the two. All the abstract undo functionality has been defined in a *CommandUndo* aspect. This aspect defines the undo as a role, the *Undoable* field and its accessors of the *AbstractCommand*.

Additionally, the change of the visibility of the methods introduced from the aspects is an issue. The visibility declared in the aspect refers to the aspect and not to the target class. However, the same problem occurs in managed data.

5.6.3 Refactoring ChangeAttributeCommand in ManagedDataJHotDraw

Finally, the aspect refactoring of the `ChangeAttributeCommand` undo concern in JHotDraw using managed data is presented in Figure 5.7.

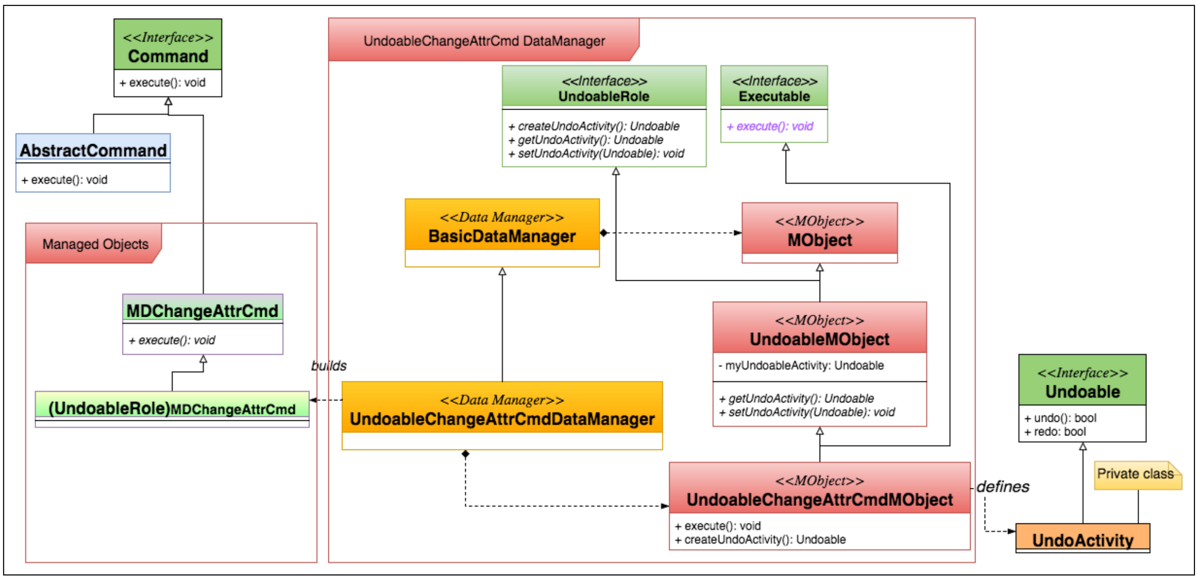

Figure 5.7: ChangeAttributeCommand in ManagedDataJHotDraw

To implement aspects using data managers in the `ChangeAttributeCommand` we first needed to migrate the original class to a managed data definition, namely *MDChangeAttrCmd*. The process was similar to the one described in Section 5.3.

In order to refactor the undo concern in managed data we needed a number of specifications that define the functionalities of our managed objects. As the AOP solution shows we needed something similar to *role superimposition*, but this time for the *Undoable* role. Additionally, we needed a way to define executables since our participant, in particular `ChangeAttributeCommand`, is a command.

In detail we needed a data manager that performs the following:

1. Attaches an *UndoableRole* to command instances. In this case the `MDChangeAttrCmd` instances.

2. Following the original design, this role should allow the objects to create their `UndoActivities` using a nested private class. *UndoableRole* should also provide accessors for its *UndoActivity*.

3. It defines an *Executable* specification, since every command has an `execute` method.

4. Finally, it defines a pointcut on the `execute` method since this is the place where the undo functionality for a specific command is implemented.

Data Manager

First, for abstraction's sake we have implemented the `UndoableRole` interface to an abstract class, namely `UndoableRoleMObject`. This data manager defines a `myUndoableActivity` field and implements its accessors given by the `UndoableRole` interface, including both `setUndoActivity` and `getUndoActivity`. However, no implementation for the `createUndoActivity` factory method is provided at this point. The reason is that this is a "command-specific" implementation and it has to be provided by the concrete classes. Since this is an abstract `MObject` no proxy factory is provided (data manager).

Next, we needed a `MObject` for the concrete command, in this case the `UndoableChangeAttrCmdMObject`. Since this has to inherit the `UndoableMObject`, it also has to implement the `createUndoActivity` method. As mentioned, this is "command-specific" code, and following the original design, this functionality can be implemented in a nested class. As a result, an `UndoActivity` private nested class is defined in the `UndoableChangeAttrCmdMObject` class file. Instances of this class can be created using the `createUndoActivity` method.

Furthermore, the command is an executable object; therefore, this `MObject` has to implement the `Executable` interface. By doing so, the pointcut of the specific undo functionality can be implemented inside this `execute` method of the `UndoableChangeAttrCmdMObject`. A comparison of the `execute` method between the original and the refactored code can be seen in Listings 5.1 and 5.2.

```
public void execute() {
  setUndoActivity(createUndoActivity());
  getUndoActivity()
    .setAffectedFigures(view().selection());
  FigureEnumeration fe =
      getUndoActivity().getAffectedFigures();
  while (fe.hasNextFigure()) {
    fe.nextFigure().setAttribute(fAttribute, fValue);
  }
  view().checkDamage();
}
```

Listing 5.1: Original execute method

```
default void execute() {
  FigureEnumeration fe = view().selection();
  while (fe.hasNextFigure()) {
    fe.nextFigure().setAttribute(attribute(),value());
  }
  view().checkDamage();
}
```

Listing 5.2: Refactored execute method

As the two listings show, all of the *Undo* related functionalities of the `ChangeAttributeCommand` command have been removed from the `execute` method. This code is located at `UndoableChangeAttrCmdMObject` and more specifically at its execute pointcut. Listing 5.34 shows the pointcut code.

```
public void execute() {
  setUndoActivity(createUndoActivity());
  getUndoActivity().setAffectedFigures(thisObj.view().selection());
}
public Undoable createUndoActivity() {
  return new UndoActivity(thisObj.view(), thisObj.attribute(), thisObj.value());
}
```

Listing 5.34: The execute Undo pointcut

5.6.4 ChangeAttributeCommand Managed Data Refactoring Overall

Comparing the original version with our refactored version several improved points can be observed. First, all of the *Undo* related functionalities of a command have been removed from the command's code and they have been externally attached by the *UndoRole* data manager. Next, following the original design, the creation of an `UndoActivity` instance is again defined inside a nested class; however, this time it is not in the command itself but it is inside a "command-specific" data manager. Finally, the `execute` method of a command is not aware of its undo functionality since it has been extracted to an undo dedicated pointcut inside a data manager.

5.7 Claims

We claim that our framework successfully performed aspect refactoring of the `FigureSelectionListener` concern in JHotDraw. Furthermore, it evolved the pointcut concept of AOP by adding conditions on them. This leads to a solution that focuses on behavior conservation, lost in the case of the AOP refactoring. Moreover, we claim that our framework also performed a successful aspect refactoring of the `ChangeAttributeCommand` undo concern in JHotDraw. Additionally, it solves the AOP's problem of useing naming conventions for the nested classes. Finally, although it is moderately easy to migrate an existing application in managed data, it is still time consuming with a lot of boilerplate code if one wants to make everything managed data.

Chapter 6

Evaluation

Having presented evidence that our framework is able to solve the problem of CCC, Chapters 3 and 5, we will now evaluate our claims. In this chapter we evaluate of our contributions in taming aspects with managed data, in relation to our research questions.

6.1 Research Questions and Answers

In Chapter 1 we stated three research questions which were the main focus of this thesis. These are answered as follows:

Can managed data be implemented in a static language? From our managed data implementation in Java, presented in Chapter 4, we conclude that it is possible to implement managed data in a static programming language. For the static implementation of managed data in Java, we used Java's reflection capabilities and Dynamic Proxies.

Can managed data solve the problem of crosscutting concerns? We argue that by using data managers, managed data can implement aspects of data in a modular way. As presented in both Chapters 3 and 5, managed data can tame aspects and solve the problem of Cross Cutting Concerns.

To what extent can managed data tame an inventory of aspects in the JHotDraw framework, compared to the original and the Aspect Oriented Programming implementation? The results of our aspect refactoring in Chapter 5 show that managed data can tame aspects of JHotDraw. In addition, we claim that our solution extends the capabilities of the AJHotDraw implementation. However, in this chapter, we provide further evaluation of our aspect refactoring and we compare it with the Aspect Oriented Programming solution.

6.2 Modularity Properties

In order to make a reliable assessment of our research, we apply the same software quality metrics presented by Hannemann et al. [HK02] in their design patterns refactoring. More specifically, the authors refer to those metrics as "closely-related modularity properties" and use them to analyze and evaluate their design pattern solutions in AspectJ.

Since our solution of the observer pattern concern in JHotDraw refers to the same issue that Hannemann et al. [HMK05] describes, we concluded that using the same modularity metrics will lead to a more reliable assessment of our solution and a more definitive assessment of our designs. The focal modularity properties are: *Locality*, *Reusability*, *Composition transparency* and *(Un)pluggability*.

6.2.1 Modularity Properties in the Observer Pattern

In their *Observer* pattern refactoring in AspectJ, Hannemann et al. [HMK05], along with Marin et al. [MMvD05a] who used the same refactoring method in AJHotDraw, assess their results using these four modularity properties.

Locality

As shown in the AOP solution, the code that implements the observer pattern is in concrete observer aspects, nothing is scattered in the participant classes. Thus, all the participants are free of the pattern context and there is no coupling between the participants [HMK05]. However, as the AOP evolution paradox suggests [TBG03], the aspects themselves are strongly coupled to the application's code.

Likewise, in the *managed data* version, *Subject* and *Observer* classes are pattern agnostic. Those classes are not aware of their pattern properties since they have been attached later by a subject role data manager.

Reusability

In the AOP version, the core pattern code is abstract and reusable. They have implemented abstract aspect code that explicitly refers to an observer pattern generalized implementation. This has been defined as an abstract aspect and it can be reused and shared across multiple observer pattern instances [HMK05]. That leads to a modular solution, although the concrete aspects have high-coupling with the application's code.

Similarly, in the *managed data* version, we have implemented the observer pattern as a reusable data manager. More specifically, the `SubjectRole` data manager is an abstract observer pattern that can be reused in any case. Moreover, the definition of this data manager is independent from the object's structure. No coupling between the application and the observer pattern implementation exists. The only "case-specific" functionality is the consistent behavior, which is similar to the one presented in the AOP solution, although, our solution supports conditional pointcuts. Additionally, the managed data version defines the pointcut in an explicit interface, separating the general observer from the specific consistent behavior.

(Un)pluggability

In the AOP version, since *Subjects* and *Observers* do not 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 [HMK05].

The *managed data* version, takes this a step further. It allows the programmer to "plug" or "unplug" specifications of the pattern by simply creating managed objects, using the preferable data manager with or without the specifications for the pattern. One can switch to the pattern implementation by creating objects provided by the subject role data manager, or simply by the basic data manager, which would lead to a non-observable object. Furthermore, this (un)pluggability can be enabled in data managers composition level, by stacking data managers, something we have applied in the subject role and pointcut case.

Composition transparency

In the AOP version, a pattern's participants are not coupled to the pattern, neither to the abstract aspects. Therefore, if a *Subject* or an *Observer* takes part in multiple observing relationships, their code does not become more complicated [HMK05]. However, the composition becomes more complicated in the join-points, since such an aspect implementation has to include multiple roles.

In the *managed data* version, we can implement such a thing in various ways. First, we can choose the AOP approach, by implementing a generalized data manager for the pattern and concrete data managers for each implementation. Alternatively, we could implement a data manager which implements a set of specifications that result to the composition of patterns. Such solution would lead to a separate specification definition that describes a composition of patterns. Consequently, the final system will be simple and (un)pluggable since the composition is transparent in a data manager.

6.2.2 Unpluggability 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 on two criteria. First, on the degree of *tangling* of the undo setup in the command's logic, specifically 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.

In the AOP version of the `ChangeAttributeCommand` the refactoring is considered successful. First, the `UndoActivity` nested class, accompanied by its factory method, is removed from the command's code. Next, the accessors of the `UndoActivity` are inherited from top level classes and not overridden locally. Finally, the undo related code in the nested classes is *unpluggable* and suitable for extraction and refactoring.

Likewise, the managed data version followed the same unpluggability principles as the AOP version. Following the similar design we managed to achieve the same level of unpluggability. However, in this case the methods related to the crosscutting functionality of undo have not been defined by top level classes but by a set of stacked data mangers. Additionally, in managed data the `UndoActivity` is not defined based on naming conventions like in AspectJ, but through a nested private class definition dedicated to the command-specific data manager.

6.3 Discussion

Conclusively, managed data is able to successfully tame aspects in a system. However, the following issues should be acknowledged:

6.3.1 Modularity

First, the modularity of data managers and their abstract way of data manipulation makes the implementation of aspects very simple and extensible. More specifically, implementing *components* (non-crosscutting concerns) as managed data and *aspects* (crosscutting concerns) by using specific data managers, one completely separates the two concepts. Contrary to the AOP version, managed data does not couple the aspects with the application's components code. The aspects are reusable data managers that can be plugged / unplugged and used in various concerns. Certainly, there are application specific parts, such as pointcut definitions; however, by *stacking* data managers we can support the application specific pointcuts at the lowest level. As a result, we can claim that managed data does not entirely solve the **evolution paradox** [TBG03], but it exceeds AOP system in modularity.

6.3.2 Flexibility

Managed data can be used similarly to AOP; however, the flexibility of the language takes it a step further. The design of the application is not bounded by the language, like in AspectJ's case.

AspectJ has many limitations that can be one overcome by using managed data. In particular, as shown, data managers allow to access the current object's state, while AspectJ's aspects implementation do not. An object's state can be used for implementing functionalities depending on the current state of an object, for instance conditions in pointcuts.

Furthermore, AOP tries to discriminate methods based on some common structural properties such as particular coding conventions. For instance, AspectJ's mechanisms do not allow introduction of nested classes; therefore in the case of the undo refactoring, the post-refactoring association was only an indirect one, based on naming conventions ("`UndoActivity`"). This is a weaker connection than the one provided by the original solution. Additionally, what happens in case the conventions followed do not agree with the rules, or in case developers do not follow the desired conventions? In other words, it is very difficult to capture the required join-points for the aspect weaver in a general and extensible way. Managed data on the other hand do not require such conventions. Since we have defined our data, its structure is accessible via the `schemaKlass` of the `MObject`, therefore we do not need any kind of conventions to determine the structure of properties. Moreover, private classes in managed data is as easy to be defined as in the original version since `MObject` is just a class. Therefore, no code conventions are needed.

6.3.3 Performance

In practice, since managed data has been implemented using Java reflection and Dynamic Proxies, it is unfavorable for applications that need performance to use it. AOP, and specifically the *aspect weaving* process, provides an oblivious way of dealing with aspects. The weaver produces static Java code, which is then compiled in JVM and can be optimized. On the contrary, data managers dynamically analyze the schemas through reflection, which makes it a lot harder for the compiler to optimize. More specifically, even though the HotSpot JVM has one of the best just-in-time compilers, Java's dynamic proxies introduce 6.5x overhead [MSD15]. Thus, AOP performance is much higher than managed data.

6.3.4 Migration and Integration

Finally, both the migration and the integration of an existing application in the two cases has some trade-offs.

On one hand, the integration of an AOP version requires a whole new language (e.g. AspectJ) in line with the new concepts of the AOP paradigm. A developer has to implement *aspects*, *advices*, *join-points* and *pointcuts*. In addition, the programmer has to setup AspectJ to the IDE of preference along with the programming environment. After the environment has been set up, the migration of an existing application to the AOP paradigm is relatively easy. Finally, a developer has to simply migrate the application using the aspect oriented language's concepts and the weaver will do the job.

On the other hand, the integration of managed data is very simple since it is pure Java. No new language is required, no any additional IDE setup nor programming environment. Additionally, although managed data introduces new concepts, it still uses the Java syntax, e.g. interfaces and annotations; therefore, the integration is a lot simpler. It is relatively easy for a developer, who is familiar with Java, to learn the managed data concepts. However, the migration of an existing application to managed data can be time consuming since a programmer has to migrate it from normal Java class definition to managed data schemas definition (interfaces). Additionally, some limitations arise from this migration because interfaces and classes do not explicitly support the same functionalities. As seen, the framework has some limitations such as Java keywords and encapsulation of the methods in interfaces. However, such definitions could be defined using annotations in future work.

6.4 Threads to Validity

In this section we present a set of validity criteria including *Construct*, *Internal*, *External* validity and *Reliability* [ESSD08]. First, regarding the construct validity of this research, one should keep in mind that our results are compared to another research's findings. The AJHotDraw implementation may not be indicative of the overall AspectJ's capabilities in aspect refactoring nor was altered by us. In other words our results focused on the comparison of our aspect refactoring in managed data with Marin's et al. AJHotDraw aspect refactoring.

The internal and external validity of this thesis is satisfactory since our focus was narrowed to a representative case and predefined and tested metrics. However, the AJHotDraw provides a more complete aspect refactoring of the original version. For instance, in the undo concern case, AJHotDraw refactors all of its instances inside JHotDraw. In our case we only refactored a part of the undo concern, in order to show how our framework will tame this aspect; a limited approach when compared to AJHotDraw's holistic implementation.

Finally, the reliability of this research can be problematic. More specifically, the refactoring is strictly dependent on the programmer's design and unless our design is used when replicating this research, a different assessment could be reached.

Chapter 7

Conclusion

TODO

Acknowledgments

TODO

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.

[BVDVET05] Magiel Bruntink, Arie Van Deursen, Remco Van Engelen, and Tom Tourwe. On the use of clone detection for identifying crosscutting concern code. *Software Engineering, IEEE Transactions on*, 31(10):804–818, 2005.

[ESSD08] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 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.

[MMvD05a] 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.

[MMVD05b] Marius Marin, Leon Moonen, and Arie Van Deursen. A classification of crosscutting concerns. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 673–676. IEEE, 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.

[RM02] Martin P Robillard and Gail C Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th international conference on Software engineering*, pages 406–416. ACM, 2002.

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

[SGP04] David C Shepherd, Emily Gibson, and Lori L Pollock. Design and evaluation of an automated aspect mining tool. In *Software Engineering Research and Practice*, pages 601–607. Citeseer, 2004.

[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.