Taming Aspects with Managed Data

Theologos A. Zacharopoulos
theol.zacharopoulos@cwi.nl

April 10, 2016, 29 pages

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

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

Contents

Abstract

Chapter 1

Introduction

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

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

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

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

1.1 Initial Study

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

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

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

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

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

1.2 Problem statement

1.2.1 Problem Analysis

Predefined data structuring mechanisms

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

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

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

Crosscutting concerns

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

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

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

Aspect Oriented Programming problems

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

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

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

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

1.2.2 Research Questions

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

1.2.3 Solution Outline

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

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

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

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

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

The Singleton Pattern, which as presented [HMK05] [HK02], can easily be abstracted as an aspect and replace the OOP usage in JHotDraw.

The Template Method, which as presented [HMK05] [HK02], scatters code by introducing roles such as those of *AbstractClass* and *ConcreteClass*.

This inventory is implemented using data managers that have modularity as a main characteristic and is evaluated in a new JHotDraw implementation. We compared those aspects with the original version of JHotDraw, and the aspect version, AJHotDraw. Since our solution is a refactoring of the JHotDraw framework we needed a way to ensure the behavioral equivalence between the original and the refactored solution [Fow09]. However, JHotDraw comes with no tests. Thus, we use TestJHotDraw, which is a subproject of the AJHotDraw development team, and is developed in order to contribute to a gradual and safe adoption of aspect-oriented techniques in existing applications allowing for a better assessment of aspect orientation. Since we use our JHotDraw implementation

for the functional evaluation of our solution, we can use some already researched criteria [HK02], which consist of *Locality*,*Re-usability*, *Composition Transparency*, and *(Un)pluggability*, in order to present metrics of our solution.

1.2.4 Research Method

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

Managed data implementation in a static programming language. In order to answer the question if managed data could be implemented in a static language, we've implemented managed data in Java using Java's reflection capabilities[5], Java interfaces for schemas definition and dynamic proxies[6] for the data managers. An extensive presentation of the implementation is given in Chapter 4.

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

Use case evaluation. In order to show if our managed data solved the issues of AOP in terms of modularity, we have gathered a number of metrics for each of the three implementations the results are presented extensively in Chapter 5.

1.3 Contributions

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

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

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

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

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

1.4 Related Work

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

Meta-Object Protocol

Managed data can be implemented using reflection and the MPO. The authors of Enso [LvdSC12] implemented it in Ruby using the meta-programming framework and more specifically, the **method_missing** mechanism. In other languages (such as Java, JavaScript or C#) that support dynamic proxies, they can be used for the managed data implementation, which is the way we've implemented it. The MPO [KDRB91] was first implemented for simple OOP capabilities of the Lisp language in order to satisfy some developer demands including compatibility, extensibility and developers experimentation. The idea was that the languages have been designed to be viewed as black box abstractions without giving the programmers the control over semantics or the implementation of those abstractions. MPO opens up those abstractions to the programmer so he can adjust aspects of the implementation strategy. Providing an open implementation can be advantageous in a wide range of high-level languages and thus MPO technology is a powerful tool for providing that power to the programmer [KDRB91]. Furthermore, MPO provides flexibility to the programmer because as a language becomes more and more high level and it's expressive power becomes more and more focused, the ability to cleanly integrate something outside the language's scope becomes more and more difficult. Thus, both MPO and managed data allow the programmer to be able to control the interpretation of structure and behavior in a program. However, MPO focuses on behavior of the objects and classes, while in managed data the focus rests solely on the data management. One could conclude that managed data is a subset of the MPO approach since managed data have a more narrow scope.

Adaptive Object Model

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

Model Driven Software Development

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

The Enso Language

Enso project[7] is the first implementation of managed data, it is open source[8] and is used for EnsoWeb, a web framework written with managed data. Although Enso is implemented in Ruby, which is a dynamic language, the source code was a very helpful resource for our static implementation in Java. The design of Enso was an inspiration for our implementation even

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

though some parts have changed completely in order to conform to our needs and support Java's static type system. Additionally, examples presented in Enso, are also implemented in our case and are presented in Chapter 3.

Aspect Oriented Programming

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

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 is discussed, providing explanation of our issues and implementation details. Next, in Chapter 5 an evaluation of our implementation is illustrated, by applying it in JHotDraw. Additionally, some metrics, claims and results are presented. Finally, a conclusion is given in Chapter 6 followed by further work in Chapter 7.

Chapter 2

Background

2.1 Cross Cutting Concerns

A lot of research in software engineering focuses on the importance of software modularity. The number of advantages of modular systems are outrageous, with one of the most significant, the extensibility and evolution of a system [Par72].

During the development of a system though there are many concerns that they have to be considered and implemented into the system. In order to follow the modularity principles, those concerns have to be implemented in a separate modules, this way the program will be extensible and it's evolution easier. However, many of those concerns can not fit into the existing modular mechanisms of any of the programming paradigms including both OOP and PP. In those cases, the concerns are scattered through the modules of the system, resulting to scattered and tangled code, those concerns called Cross Cutting Concerns [HMK05]. CCC considered a significant issue for the evolution of a system and the affects of tangled and scattered code disastrous for a system's extensibility.

The reason is that, the code that it is related to one concern now is scattered in multiple modules, while the concern code is now tangled with the module's logic. This results to a system that does not follow the *Single Responsibility Principle* principle and consequently the system will be hard to maintain.

Some examples of those CCC are the following: persistence, caching, logging, error handling [LL00], access control and many more, as well as some design patterns that scatter "design pattern code" thought the application, such as the *Observer Pattern*, *template Pattern*, *command Pattern* etc. [HK02] [Mar04].

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

2.2 Aspect Oriented Programming

Kiczales et al. present [KLM+97] using an example of a simple image processing application, that in general, whenever two properties being programmed must compose differently and yet be coordinated (in the example filters and loop-fusion), they **crosscut** each other. Because general purposes languages provide only one composition mechanism, and those mechanisms lead to complexity and tangling, the programmer must do the co-composition manually. According to their theory, a property that must be implemented can be either a *component*, if it can be cleanly encapsulated in a generalized procedure, or an *aspect*, if it can not be cleanly encapsulated in a generalized procedure. AOP supports the programmer in cleanly separating components and aspects from each other, by providing mechanisms that make it possible to abstract and compose them when producing the overall system. This is in contrast to general purpose programming, OOP or PP, which support programmers in only separating components from each.

However, implementing AOP programs is not that easy and several tools are needed. More specif-

ically, while a general purpose language needs a language, a *compiler* and a *program* written in the language that implements the application, the AOP based implementation of an application 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 one or more *aspect programs* that implement the aspects using the aspect languages. Essential to the function of the aspect weaver is the concept of join points, which are those elements of the component language semantics that the aspect programs coordinate with.

2.2.1 AspectJ

There is a lot of work in the area of aspect oriented languages but one the most important contribution is the AspectJ[1] project. Kiczales et al. introduce AspectJ [KHH+01]. AspectJ is a simple and practical aspect-oriented extension to Java. 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 otherwise would lead to tangled 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 characteristic of AspectJ is that it is not a DSL but a general purpose language that uses Java's static type system. Something that it holds for our managed data implementation as well.

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 GoF design patterns [Gam95] in AspectJ, where 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 and objects. There are several problems that the OOP design patterns introduce in respect of CCC, and 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 tailored to the instance of use, and this often leads them to disappear into the code and lose their modularity [HK02]. Even worst, 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 with a detailed implementation. As they mention, in a observer pattern implementation, both the *Subject* and the *Observer* have to know about their roles in the pattern and consequently have "pattern related code" in them, so that adding and removing code from a class requires changes in that class.

In their implementation[6] of the observer pattern in AspectJ, they separate abstract aspects for:

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

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

- Which classes can be Subjects and which Observers.

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

- A set of changes on the Subjects that triggers the Observers.
- The specific means of updating each kind of Observer when the update logic requires it.

The modularity properties which this implementation relates are:

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

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

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

(Un)pluggability: Because Subjects and Observers dont 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.

More design pattens in Aspect Oriented Programming

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

2.2.3 Evolvability issues

Since modularization and separation of concerns makes 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, but paradoxically they are not [TBG03]. AOP technologies deliver applications that are as hard, and sometimes even harder to than was the case before.

According to Tourwe et al. [TBG03] the problem is that aspects have to include a crosscut description of all places in the application. Consequently it is much harder to make such crosscuts unaware 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 of course this affects negatively the evolvability of a system. As Tourwe et al. [TBG03] propose a solution for the 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 is something that we tried to show in our thesis, a new language that implements CCC in a modular way.

2.3 Managed Data

Managed data [LvdSC12] is a data abstraction mechanism that allows the programmer to define the data and their manipulation mechanisms. It provides a modular way to control aspects of data. Managed data is an approach to data abstraction that helps the programmer by giving them control over the structuring mechanism. Until now those data structuring mechanisms were predefined form the programming languages and the developers could not take control on the data structuring and management mechanisms, but only to create data of those types.

Managed data provide flexibility and lifts data management up to the application level, by allowing the programmer to build data managers that handle the fundamental data manipulation primitives that are normally hard-coded into the programming language.

The input to a data manager is a schema, which describes the structure and behavior of the data to be managed. Managed data has three essential components:

Data description language, that describe 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 created and manipulated.

In the traditional approach, the programming language includes a process and data definition mechanism, 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 in managed data are the way to describe the structure and the behavior of the data to be managed. Schemas can be just a simple data description language which programmers can use describe simple kind of data. For example Loh et al. [LvdSC12] used Ruby hash for the data description on a simple example where the hash was an object that represents a mapping from values to values. However, a simple schema format like this can not be used to describe itself, because a simple schema is not a record. We need a self-describing schema that can be used to describe schemas. Self-describing allows schemas to be managed data themselves. A Schema-Schema is also managed data with its own data manager. This process is called *Bootstrapping* and it is needed in order to jumpstart this process, this extends the benefits of programmable data structuring to their own implementation. Schemas can be interpreted in many different ways to create different kinds of records based on the manipulation provided by the data managers.

2.3.2 Data Managers

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

2.4 Java Reflection and Dynamic Proxies

The Java programming language provides the programmer with a Reflection API[2] and with it offers the ability to examine or modify the runtime behavior of applications running in the Java Virtual Machine (JVM). Also, 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 software environment, and to change what it does depending on what it finds [FFI04].

In order for this self-examination to be done, the program needs to have a representation of itself which is called *metadata*. In a Object Oriented programming language this metadata is organized

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

into objects, called *metaobjects*. Finally, the runtime self-examination of these metaobjects is called *introspection*.

Java supports reflection with it's reflection API since the version 1.1. Java provides a operations for using metaobjects performing intercession. Using Java reflection a running program can learn a lot about it self, this information may include: 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 many 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 the runtime. Using dynamic invocation, a `Method` metaobject can be commanded to invoke the method that it represents.

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

Security Since metaobjects give a developer the ability to invoke and change underline data of the program, that gives access to places that are supposed to be secure.

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

Runtime performance Of course the runtime dynamic examination and introspection introduce significant overhead on most language implementations in case of Java's dynamic proxies it has been observed 6.5x overhead [MSD15] However, this is not something to consider in this thesis.

2.4.2 Dynamic Proxies

Since the version 1.3 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 *proxy* the same interface as the *target* so that it can be used in exactly the same way. The proxy *delegates* some or all of the calls that it receives to its target and thus acts as either an intermediary or a substitute. As a result, a programmer has the capability to add behavior to objects reflectively and dynamically. The Java reflection API contains a dynamic proxy-creation facility, `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 of interfaces for which that proxy was created. The corresponding proxy class extends class `Proxy` and implements all interfaces. Thus, conforming to all those interfaces, a proxy can be casted to any of them, and any method defined in those interface 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.

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

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), a the method itself that was invoked on the proxy, and the arguments of that method. Therefore, the `invoke` method is capable of handling any method invocation.

Issues

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

The methods `equals`, `hashCode`, and `toString` 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 proxy classes, and invocations to them are forwarded to the invocation handler of the proxy. Other methods defined in Object are not overridden by proxy classes, as they are `final` [Eug06].

Logging Example

Previously we mentioned the *logging* CCC. In case of every method invocation of an object has to be logged into the console we need to write logging code on each of the methods of that class. Hence, this would lead to scattered logging code and tangled code with the method's logic. This is a problem that can be solved with dynamic proxies, and it's can be seen into the following source code.

```
1
2  import java.lang.reflect.*;
3  import java.io.PrintWriter;
4
5  public class TracingIH implements InvocationHandler {
6      public static Object createProxy( Object obj, PrintWriter out) {
7          return Proxy.newProxyInstance(
8              obj.getClass().getClassLoader(),
9              obj.getClass(). getInterfaces (),
10             new TracingIH( obj, out));
11     }
12
13     private Object target;
14     private PrintWriter out;
15
16     private TracingIH(Object obj, PrintWriter out) {
17         target = obj;
18         this.out = out;
19     }
20
21     public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
22         Object result = null;
23
24         try {
25             out.println( method.getName() + "(...) called" );
26             result = method.invoke( target, args );
27         } catch (InvocationTargetException e) {
28             out.println( method.getName() + " throws " + e.getCause() );
29             throw e.getCause();
30         }
31         out.println(method.getName() + " returns" );
32         return result;
33     }
34 }
```

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 is a showcase for software quality techniques provided to the OOP community.

The fact that JHotDraw is appraised as a so well-designed application makes it an ideal candidate as a showcase for an aspect oriented migration. Marin and Moonen [MM] use this showcase for adoption of aspect-oriented techniques in existing systems. More specifically, they present AJHotDraw[5], which is an aspect-oriented version of JHotDraw developed in AspectJ 2.2.1. The goal of AJHotDraw is to take the existing JHotDraw and migrate it to a functionally equivalent aspect-oriented version. The contributions of AJHotDraw include:

1. Identification and documentation of crosscutting concerns in JHotDraw, which helps a lot to aware the developers of the existence of CCC in JHotDraw.
2. Building a common benchmark for testing aspect mining techniques and tools.
3. Associating aspect solutions to the identified crosscutting concerns and extra general solutions including design patterns, good practices etc.
4. Assessing reliability of proposed aspect solutions to known crosscutting concerns in the context of JHotDraw.
5. Building a model application based on an aspect-oriented solution.
6. Providing support for refactoring from an object-oriented solution to an aspect oriented solution.
7. Addressing the challenges risen by testing aspect oriented systems.

The first thing that AJHotDraw developers needed to do was to create a test subproject for the existing JHotDraw (called TestJHotDraw[6]) to ensure behavioral equivalence between the original and the refactored solution, since refactoring implies preserving the observable behavior of an application[Fow09] and the original JHotDraw comes without tests. There were several benefits taken from the aspect-oriented implementation approach[MM]. The authors believe that the project will contribute to a gradual and safe adoption of aspect-oriented techniques in existing applications and allow for a better assessment of aspect orientation.

In this thesis we have used JHotDraw and AJHotDraw as well as the test project, TestJHotDraw, in order to evaluate our CCC refactoring in managed data. The detailed evaluation is described in Chapter 5.

2.5.1 Refactoring of Cross Cutting Concerns

The refactoring of legacy code to aspect oriented code is known as *Aspect Refactoring*. During this process 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 for a type is needed in order to ensure behavior conservation, hence some coherent criteria to organize CCC are needed. Marin, Moonen and Deursen [MMvD05] organize the CCC into types, which are descriptions of similar concerns that share a number of 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.)

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

- An associated (desired) aspect language mechanism to support the modularization of the type's concerns (e.g., *pointcut* and *advice*, introduction, composition models).

During AJHotDraw implementation[MMvD05] [HMK05], the authors propose a type-based refactoring on the same *Observer* instance, `SelectionListener`, in JHotDraw.

The legacy code architecture of JHotDraw is displayed in figure 2.1.

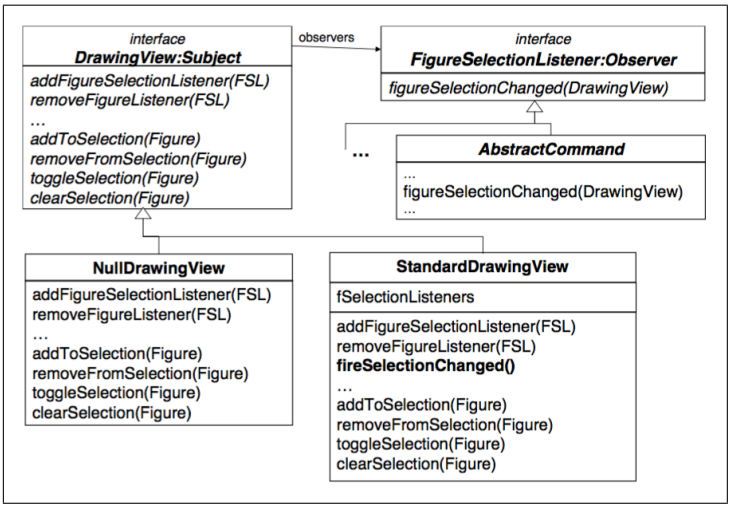

Figure 2.1: Observer pattern: Selection Listener [MMvD05]

The `FigureSelectionListener` interface defines the *Observer* role as its primary concern. The interface is implemented by all classes interested in changes of the selection of figures in a drawing view. The `DrawingView` interface partially defines the *Subject* role. The *Subject* role is a secondary concern for the `DrawingView` interface. Two classes implement this interface and only one, `StandardDrawingView`, contains a non-empty implementation of the *Subject* role.

The aspect refactoring would be described as the aspect construct comprises both the *Subject* and *Observer* roles definition, and maintains a list of associations between each *Subject* and its *Observer* objects. The type-based refactoring[MMvD05] distinguishes several crosscutting elements that occur in an implementation of the *Observer* pattern: role superimposition, applied twice, for each of the two roles and consistent behavior to notify the observers of the changes in the subject object. 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 the figure 2.2.

Role-based Refactoring of Cross Cutting Concerns

Evaluation

2.5.2 The "Undo" Concern of JHotDraw

Evaluation

AspectJ Drawbacks in the Undo Solution

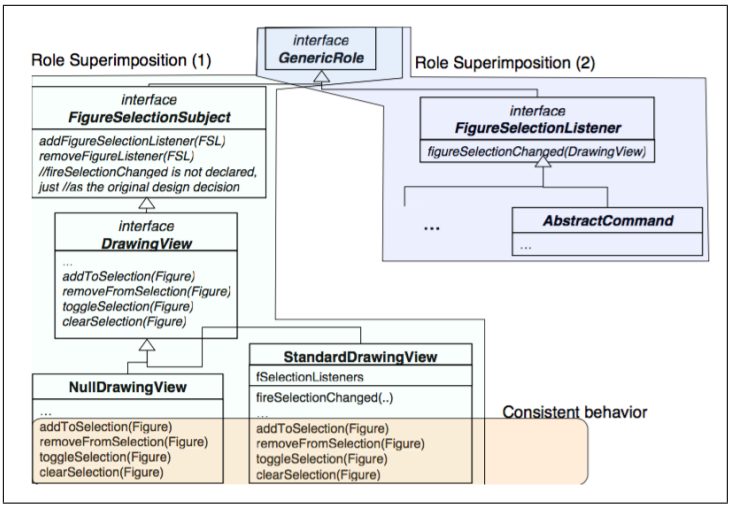

Figure 2.2: The concern types in Selection Listener [MMvD05]

Chapter 3

Example Application

3.1 Schemas definition

3.1.1 State Schema

3.1.2 Machine Schema

3.1.3 Transition Schema

3.2 Data managers definition

3.2.1 Basic Data Manager

3.2.2 Observable Data Manager

3.3 Aspects

3.3.1 Logging

Chapter 4

Implementation

4.1 Managed Data

4.1.1 Schema

Schema Definition

4.1.2 Data Managers

Data Managers Definition

4.2 Bootstrapping

4.2.1 Cutting the umbilical cord

4.3 Self-describing schema (SchemaSchema)

4.4 Schema Loading

4.4.1 Forward

4.4.2 Wire the Cross-References

4.5 Typing

4.5.1 Primitives

4.5.2 Collections

4.6 Implementation Issues

4.6.1 Methods ordering

4.6.2 Hash-code of Managed Objects

4.6.3 Default methods of Managed Objects

4.6.4 Collections of Managed Objects

4.6.5 Transparent equivalence

Chapter 5

Evaluation

5.1 JHotDraw and AJHotDraw

5.2 Research Questions and Answers

5.3 Evidence

5.3.1 Design Patterns

5.3.2 Undo Concern of JHotDraw

2.5.2

5.3.3 The Observer Pattern in JHotDraw

2.2.2 2.5.1

5.4 Results

5.4.1 Locality

5.4.2 Reusability

5.4.3 Composition transparency

5.4.4 (Un)pluggability

5.5 Claims

Chapter 6

Conclusion

Chapter 7

Further Work

Acknowledgments

Appendix A

How to Use the Framework

Appendix B

Refactoring of JHotDraw's Undo Concern

Appendix C

Refactoring of the Observer Pattern in JHotDraw

Bibliography

[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.

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

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

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

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

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

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

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

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

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

[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.

[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.