Taming Aspects with Managed Data

Theologos A. Zacharopoulos
theol.zacharopoulos@gmail.com

April 6, 2016, 19 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 for which 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) presented [KLM+97] in order to provide a solution by introducing the notion of *aspects*.

AOP results in a modular and *single-responsibility* 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 a process called *weaving*. However, relying on AOP, paradoxically, does not improve the evolution of a project even with the modularity that it provides since it introduces tight coupling between the aspects and the application. As a result the way to tackle with this problem we need a more sophisticated and expressing crosscut language. Consequently, CCC could be handled in a higher level of the language such as the data structuring and management mechanisms.

Managed data [LvdSC12] allows programmers 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 Rubys reflection capabilities. Furthermore, managed data are considered less able to be supported in static languages directly which makes it more challenging for this thesis since it is going to be implemented in Java. In this thesis I am going to use the Java reflection capabilities to implement managed data and focus on specific aspects and design patterns implementations using the data managers concept of managed data.

1.1 Initial Study

In their study on managed data, Loh et al. [LvdSC12] present an implementation of managed data in Ruby and they use as a case study a web development framework from the Enso project 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 implementation of the meta-object protocol [KDRB91], which can be used though the Java Reflection API [FFI04]. Additionally, this project will focus on aspects and will provide a solution to the CCC problem by using managed data.

[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

1.2 Problem statement

The problem we study regards the CCC that are scattered around the application, resulting to a hard to maintain system by tangling implementation logic and concerns code together. 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, Friedrich Steimann [Ste05] argues that modeling languages are not aspect ready. The problem that arises is located at the level of software modeling. More specifically, in *roles modeling*, whereas in OOP roles are tied to the collaborations, collaborations rely on interactions of objects, and aspects on the other hand 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 describe the order rather 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.

In order to solve the aforementioned problems, we implement manage data, lifting the data management up to the application.

1.2.1 Problem Analysis

1.2.2 Research Questions

Managed data has not been practically implemented in a static language before, therefore my first research questions states "Can managed data be implemented in a static language like Java?". Based in the previous argumentation about the relevance of AOP and the solutions that managed data can provide in CCC, my 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], I am going to 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, in contrast with the original and the AOP implementation."

1.2.3 Solution Outline

Our solution is consisted of an implementation of managed data in Java. This framework can be used by applications in order to deal with CCC.

To validate our hypotheses we are going to implement managed data in Java using the Java Reflection API and Dynamic Proxies. More specifically we are going to use Java interfaces for *schemas* and dynamic proxies for *data managers*. Furthermore, we are going to provide as a proof of concept the examples given in [LvdSC12] but this time developed in Java. As mentioned in [LvdSC12] to stack data managers I am going to use the Decorator Pattern [Gam95].

In order to prove that managed data solves the problems that AOP introduces, we are going to implement 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 Singleton Pattern, which as presented [HMK05] [HK02], it can easily be abstracted as an aspect and replace the OOP usage in JHotDraw.

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

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

This inventory is implemented using data managers that have modularity as a main characteristic and is been evaluated in a new JHotDraw implementation. We compare those aspects with the original version of JHotDraw, and the aspect version, AJHotDraw. Since our solution is a refactoring of the JHotDraw framework we need a way to ensure the behavioral equivalence between the original and the refactored solution [Fow09]. However, JHotDraw comes with no tests. Thus, we use the TestJHotDraw, which is a subproject of the AJHotDraw development team, and it is developed in order to contribute to a gradual and safe adoption of aspect-oriented techniques in existing applications and allow for a better assessment of aspect orientation. Since we use our JHotDraw implementation for the functional evaluation of our solution, we can use the presented criteria [HK02], which are *Locality*,*Re-usability*, *Composition Transparency*, and *(Un)pluggability*, in order to present metrics of our solution.

1.2.4 Research Method

The answers for the research questions have been extracted from the background material, in our Java managed data implementation and an evaluation of our implementation in an existing use case system the JHotDraw.

Managed data implementation in a static 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], using 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 and evaluation. In order to argue about the contribution of our implementation and managed data for aspects handling in general, we've used an use case application (JHotDraw) which is considered as a good design use case for OOP, along with it's AOP implementation (AJHotDraw). Using these application as references we've implemented our own version of JHotDraw using managed data to tame some of it's aspects, the results are presented extensively in Chapter 5.

1.3 Contributions

Contribution 1: Managed data implementation in Java. Our first contribution with this thesis is the implementation of managed data in a static language like Java. 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 with which the developer can define and implement aspects as reusable modules and integrate them with an application without mixing the business logic with concern logic. More specifically, the schemas and the data managers have to be defined by the developer, as well as any additional functionality that may needed 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 evaluated our CCC solution. More specifically, we focused on the *Undo* concern, which is scattered around the implementation of the JHotDraw.

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

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

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 Meta-Object Protocol (MPO). The authors of Enso [LvdSC12] implemented it in Ruby using reflection and 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 wide range of high-level languages and that 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 focus on the data management only. 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. Additionally, 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 of 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 model represent abstract data that consisted 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 meta-data that describe an entity following by a code generation that adds functionality to that entity for it's manipulation.

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 informative place to start for our static implementation in Java. The structure of Enso was an inspiration of our implementation even though some parts have changed completely in order to follow our needs and support of Java's static system. Additionally, examples presented in Enso, are also implemented in our case and are presented in the Appendix B.

Aspect Oriented Programming

Although AOP is not directly connected to managed data, it allows a mechanism that is relative easy to be supported in managed data. This mechanism consist of the *weaving* of aspect code in specific join points. The way to support this mechanism in managed data it is through data managers, which it is the main topic of this thesis and is going to presented deeply 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 familiarizes with in order to follow the next chapters. In Chapter 3 a basic theory is presented, which is going to be used a a reference for our implementation. In Chapter 4 the implementation is demonstrated and analyzed, providing explanation of our issues and implementation details. Next, in Chapter 5 an evaluation of our implementation is presented, by applying it in JHotDraw. Additionally, some metrics, claims and results are presented. Finally, a conclusion is given in Chapter 6 along with some further work in 7.

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

Chapter 2

Background

2.1 Cross Cutting Concerns

2.2 Aspect Oriented Programming

2.2.1 AspectJ

2.2.2 Design Patterns in Aspect Oriented Programming

2.2.3 Aspect Oriented Programming Evaluation

2.2.4 Evolvability

2.3 Managed Data

2.3.1 Schemas

2.3.2 Data Managers

2.4 Internal DSLs

2.5 Java Reflection and Proxies

2.5.1 Reflection

2.5.2 Reflection and MetaObject Protocol

2.5.3 Dynamic Proxies

Chapter 3

Theory

3.1 Self Describing

3.1.1 Reuse

3.1.2 Malleability

3.1.3 Java runtime

3.2 Model Driven Development

3.2.1 Object and Schemas

3.3 Schema

3.3.1 Description of Schema

3.3.2 Schema Schema

3.3.3 Metadata

3.4 Factories

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

5.1.1 Refactoring of Crosscutting Concerns

Role-based Refactoring of Crosscutting Concerns.

Evaluation

5.1.2 The Undo Concern of JHotDraw

Evaluation

AspectJ Drawbacks in the Undo Solution

5.1.3 The Persistence Concern of JHotDraw

5.2 Research Questions and Answers

5.3 Evidence

5.3.1 Design Patterns

5.3.2 Undo Concern of JHotDraw

5.3.3 Persistence Concern of JHotDraw

5.4 Results

5.5 Claims

Chapter 6

Conclusion

Chapter 7

Further Work

Acknowledgments

Appendix A

How to Use the Framework

Appendix B

Example Application

B.1 Schemas definition

B.1.1 State Schema

B.1.2 Machine Schema

B.1.3 Transition Schema

B.2 Data managers definition

B.2.1 Basic Data Manager

B.2.2 Observable Data Manager

B.3 Aspects

B.3.1 Logging

Appendix C

Refactoring of JHotDraw's Undo Concern

Bibliography

[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.

[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.

[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.

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

[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.