AUTOMATED REFACTORING OF DESIGN PATTERN IMPLEMENTATIONS TO ASPECT ORIENTED COUNTERPARTS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

ALİ BUĞDAYCI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

DECEMBER 2007

Approval of the Graduate School of Informatics

_____
Prof. Dr. Nazife Baykal
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

_____
Assoc. Prof. Dr. Yasemin Yardımcı
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

_____
Dr. Aysu Betin-Can
Supervisor

Examining Committee Members

Assoc. Prof. Dr. Ali Doğru	(METU, CENG)	_____
Dr. Aysu Betin-Can	(METU, IS)	_____
Dr. Ali Arifoğlu	(METU, IS)	_____
Dr. Cengiz Çelik	(BILKENT, CS)	_____
Dr. Alptekin Temizel	(METU, IS)	_____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this wok.

Name, Last name : **Ali Buğdaycı**

Signature : _____

ABSTRACT

AUTOMATED REFACTORING OF DESIGN PATTERN IMPLEMENTATIONS TO ASPECT ORIENTED COUNTERPARTS

BUĞDAYCI, Ali

MSc. , Department of Information Systems

Supervisor: Dr. Aysu Betin-Can

December 2007, 62 pages

In this thesis, automation of refactoring Design Pattern implementations to their Aspect Oriented Programmed(AOP) counterparts is studied. A recent study has shown that Aspect Oriented implementations of the Gang of Four design patterns lead to modularity improvements in 17 of 23 cases for the Java Programming Language. These improvements are manifested in terms of better code locality, reusability, composability, and pluggability. Using case studies, the effectiveness of automation and refactoring to AOP counterparts are shown. The results show that automation of refactoring Design Pattern implementations to their AOP counterparts can be applied for the already implemented software projects with ease. Our tool replaces the old object oriented pattern code with an automatically created AOP implementation. While automating the refactoring, we encountered some new problems that were not explored before. Hence with our tool different

object oriented pattern implementations can be automated, and no further design problems occur after the refactoring.

Keywords: Design Patterns, GOF, Aspect Oriented Programming, AOP, Automation

ÖZ

KODDAKİ TASARIM DESENLERİNİN GÖRÜNÜM YÖNELİMLİ PROGRAMLI EŞDEĞERLERİNE YENİDEN DÜZENLENMESİNİN OTOMATİZE EDİLMESİ

BUĞDAYCI, Ali

Yüksek Lisans, Bilişim Sistemleri Bölümü

Tez Yöneticisi: Dr. Aysu Betin-Can

Aralık 2007, 62 sayfa

Bu tez çalışasında tasarım kalıplarının İlgiye Yönelik Programlanmış eşdeniğine yeniden düzenlenmesinin otomatize edilmesine çalışılmıştır. Java yazılım dilinde, İlgiye Yönelik Programlamanın temel tasarım kalıplarından (patterns of Gang of Four) 23 tanesinden 17sinin modülerliliğini geliştirdiği görülmüştür. Bu gelişimeler daha iyi kod yerelliği, tekrar kullanılabilirlik, rahat düzenlenmesi ve çıkartılabilmesi şeklinde belirtilebilir. Örnek projeler üzerinden otomasyonun ve İlgiye Yönelik Programlanmış tasarım kalıplarının kullanışlığının etkinliği sorgulanmıştır. Sonuçlar göstermektedirki tasarım kalıplarının İlgiye Yönelik Programlanmış eşdeniklerine otomatik bir şekilde yeniden düzenlenmesi bu

çalışma sayesinde kolayca yapılabilmektedir.

Anahtar Kelimeler: Tasarım Kalıpları, GOF, İlgiye Yönelik Programlama, İYP, Otomasyon

ACKNOWLEDGEMENTS

I would like to thank Dr. Aysu Betin-Can , for her help, patience, professional advice, and valuable supervision during the development and the improvement stages of this thesis. This thesis would not exist without her guidance and support.

I would also like to thank my family, for their great encouragement and continuous morale support though they are far away. And I dedicate this work to my unborn children and their mother: I live to enjoy this life together in love and harmony.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS / ACRONYMS

AOL	: Abstract Object Language
AOP	: Aspect Oriented Programming
ASG	: Abstract Syntax Graph
AST	: Abstract Syntax Tree
DP	: Design Patterns
DP-2-AOP	: Our tool (Design Patterns to AOP)
EDP	: Elemental Design Patterns
FUJUBA	: From UML to Java And Back Again
IDE	: Integrated Development Environment
IDEA	: Interactive Design Assistant
JDT	: Java Development Tools
LOC	: Lines of Code
GoF	: Gang of Four
OO	: Object Oriented
OOP	: Object Oriented Programming
Ptidej	: Pattern Trace Identification, Detection, and Enhancement in Java
SPQR	: System for Pattern Query and Recognition
UML	: Unified Modeling Language
XML	: Extensible Markup Language

XSLT : Extensible Stylesheet Language Transformations

CHAPTER 1

INTRODUCTION

1.1 Motivation and Purpose of the Study

The motivation behind our work depends highly on the technologies that try to enhance the current software development. Our work brings together different software technologies: Design Patterns, Refactoring and Aspect Oriented Programming; and harmonizes them in an Integrated Development Environment.

Design patterns are abstract, reusable and proven solutions to common problems in software development. Since late 1990s, Design Patterns have been frequently used in almost every software project. The invention of the Aspect Oriented Programming (AOP) [1] concept changed the developers' perspective to the Design Pattern (DP) implementations. Hence using Refactoring, clarifying code without changing the functionality, and a widely used IDE as an environment, we automated the refactoring of regular DP implementations in Java [2] code to their AOP implementations. Since DP has been widely used through out the software projects, the automation enables revisiting, hence enhancing, already written code possible. All in all, the IDE

integration brings the usability idea to an extreme, since an IDE is a developers' necessity in today's world.

We were motivated for 4 reasons for working on this thesis subject:

- Bringing together different software technologies
- Automating the process of refactoring, hence finding a way to covering different possible implementation details of the DPs.
- Applying our work on finalized projects without much struggle using the automation process
- Design the tool so that it can mature as a product. So that it can be integrated into the widely used IDE's.

1.2 Outline

Our work depends on the different software technologies being applied; hence we start with building up the general knowledge for these technologies. In chapters 2 through 4, we explain the background and uses of these technologies and their specific relation with our work. In chapter 5, the implementation logic and our tool DP-2-AOP is presented. Finally in chapter 6 some discussion and concluding remarks are given.

CHAPTER 2

DESIGN PATTERNS

2.1 Design Patterns Overview

Design patterns are known effective solution “patterns” for widely encountered “design” problems. The general concept of the design patterns are introduced by a well known architect Christopher Alexander [3]. Christopher Alexander saw the patterns as an aid to design cities and buildings. So in order to solve widely encountered architectural problems, he brought together well known solutions together and named them as pattern language.

Although design patterns are introduced in the field of architectural sciences, collection of patterns in a pattern language is applied to many other fields. Like in computer science, a design pattern is a proven solution for a widely occurring software design problem. Gamma et. al. adapted the pattern language concept to the computer science field in 1995[4]. The design pattern solutions in software engineering are guides that aid developers to design more robust software systems. For this purpose, the pattern solutions they provide in general decrease the dependencies and increase the abstraction so that the

system can easily be maintained and developed without changing the functionalities.

The design patterns in Gamma et. al's work is known as GoF (Gang of Four) patterns. They worked on totally 23 design patterns that solve design issues related to class instantiation (Creational patterns), class and object composition (Structural patterns) and a class's objects communication (Behavioral patterns).

The design patterns solutions try to apply design principles in code as much as possible. Software design principles are guidelines that should be used in a software development.[5] Although applying design principles to full extent can be seen as a utopia, applying principles as much as possible leads to a better design. One of the well known design principle is open/closed principle, which depicts "open" for extension, "closed" for changes [6]. The open/closed principle explains that the code should be written so that it need not be changed, but rather it should be extended when needed [7]. Likewise Liskov Substitution Principle explains how inheritance contracts should be: a subclass can not have stronger preconditions and weaker postconditions than its superclass[8].

One thing that people get confused with is that they think algorithms are a part of design patterns. Algorithms are not classified as patterns; they solve computational problems, whereas design patterns solve widely encountered design issues.

2.2 Finding Design Patterns in the Code

After GoF introduced software design patterns to the computer science community, the community understood the importance of the design patterns. Researchers used GoF's work for studying different aspects of design patterns.

One of the new studies concerning design patterns is finding design patterns in the code. There are two different aspects regarding this work: finding the already used design patterns in code (i.e. reverse engineering) and searching for the design pattern that should have been applied to the system.

Shull et. al. introduced an inductive method for discovering design patterns from object oriented systems[9]. They gave a set of procedures and guidelines for repeatable and usable reverse architecting processes.

A procedure used for finding DP's uses the class structure and the relations between the classes (e.g.: delegation, uses). One of the works that depicts this procedure is that the design and the code are mapped into an intermediate representation, then a multi-stage search run on the intermediate representation to determine patterns in code [10]. OO software metrics are used to determine pattern candidates. Software metrics, and structural properties extracted from Abstract Syntax Tree (AST) is combined with the intermediate representation: Abstract Object Language (AOL). Then a pattern can be seen as a graph in which nodes are classes and edges correspond to relations. However, this approach may report more patterns than actually used [11].

Another reverse engineering approach to detect design patterns is using metrics and a machine learning algorithm to fingerprint design motifs [12]. Yann et.al. define fingerprints as sets of metric values characterizing classes playing a

given role. They devise fingerprints experimentally using a repository of "micro-architectures" similar to design motifs.

Smith et.al.'s work on design pattern recognition is a pioneer[13,14]. In the previous approaches, discovering design patterns in code used static descriptions of structural and behavioral relationships, which leads to finite library of variations of pattern implementations. Their approach differs from the conventional ones with encoding OO concepts in a formal denotational semantics as a small number of fundamental components (elemental design patterns), encode the rules by which these concepts are combined to form patterns (reliance operators), and encode the structural/behavioral relationships among components of objects and classes (rho-calculus). Then they use a logical inference system to reveal the patterns using these elemental design patterns, and find patterns inferred dynamically during code analysis by a theorem prover. Their discussion about use in composition and decomposition of existing patterns, identification of pattern use in existing code to aid comprehension, refactoring of designs, integration with traditional code analysis techniques, and the education of students of software architecture brings a value to the design pattern community. Smith et. al. matured their approach as a System for Pattern Query and Recognition(SPQR), in which the system finds patterns that were not explicitly defined, but instead are inferred dynamically during code analysis by a theorem prover, which provides practical tool support for software construction, maintenance, and refactoring[15,16,17].

Recent approaches on design pattern recognition include using inherent parallelism of bit-wise operations to derive an efficient bit-vector algorithm for finding occurrences of design patterns in a program [18]. Kaczor et. al used the

traditional approach of identification of design patterns in a program, which is class structure and organization matching. Additionally they just expressed the problem of design pattern identification with operations on finite sets of bit-vectors.

Another recent approach includes combining static analysis with dynamic analysis. Wendehals' design recovery process is based on an Abstract Syntax Graph (ASG) representation of the source code. A structural and a behavioral pattern are defined for each design pattern to enable a tool-based recognition. The process starts with the static analysis which takes the source code of the system and a catalog of structural pattern specifications as input. The source code is parsed into the ASG representation which is searched for the structural patterns. Each match of a structural pattern is annotated as a design pattern candidate. The purpose of the dynamic analysis is to check for each design pattern candidate if the collaboration of its elements at runtime conforms to the behavioral description of the design pattern. The design pattern candidates and behavioral pattern specifications are used to record method call traces during the program's execution [19].

2.3 Tool Support and Automation Work on DP

Reverse engineering design patterns with Unified Modeling Language (UML) tools is another research area in design patterns community. Sunye et. al. worked on the integration of UML and design patterns from a tool perspective. They found some vague parts and tried to clarify ambiguities and propose solutions for handling these ambiguities. They point out that a tool should be capable of the following for a design pattern recognition: Recognition, the tool recognizes that a set of classes, methods and attributes corresponds to a design

pattern application and points this out to the designer; Generation: Here, the designer chooses a pattern she wants to apply, the participant classes and some implementation trade-offs and receives the corresponding source code; Reconstruction: The former approaches can be merged into this third approach [20].

Bergenti et. al. presents a system called IDEA (Interactive Design Assistant). IDEA is an interactive design assistant for software architects meant for automating the task of finding and improving the realizations of design patterns. Basically, IDEA is capable of automatically finding the patterns employed in a UML diagram and producing critiques about these patterns [21]. They point out that a case tool capable of design pattern recognition should be able:

- to find pattern realizations
- to propose pattern-specific critiques for improvement
- to suggesting alternatives of the patterns realizations
- to propose a design pattern
- to find base recurring solutions for finding new patterns.

Gueheneuc et. al.'s work showed solving the problem of automating the instantiation and the detection of design patterns. Their research depicts tools to evaluate and to enhance the quality of object-oriented programs, which promotes use of patterns at different levels [22]. Patterns-Box tool provides assistance in designing the architecture of a new piece of software, and Pattern Trace Identification, Detection, and Enhancement in Java (Ptidej) [23] tool identifies design patterns used in an existing one. Using these tools in combination offers the users maintenance of the project by highlighting defects

in an existing design, and by suggesting and applying corrections based on design patterns solutions. In their recent work Gueheneuc shows that Ptidej has matured and uses Ptidej to share their experience about pattern claims, choices, uses, pattern definition, formalization, use for reverse-engineering and for implementation[24].

Arcelli et. al. states that many tools have been proposed and developed for reengineering, restructuring and re-documenting large systems, but most of them do not work properly due to scalability problems. In their work they tried two well known reverse engineering tools, CodeCrawler and Ptidej, and showed that a pattern identification tool like Ptidej can work well on a large system. They state that although Ptidej does not detect the involved classes' roles with the same definition names. But by analyzing the UML package's structure and detecting interesting points in the source code, they state that 91% of the detections are correct by comparing with the micro-architectures [25].

Another work by Arcelli et. al. is porting Elemental Design Patterns(EDP) for recognizing design patterns in Java[26]. Although SPQR defines a complete catalog of EDPs and how EDPs can be described though rhocalculus, it is implemented for programs in C++. Another research they base their study on is "From UML to Java And Back Again" (FUJUBA) [27] FUJUBA uses decomposing design patterns into subcomponents which improves design pattern detection process and results. Hence they come up with their own EDP recognizer for Java: EDPDetector4Java.

CHAPTER 3

ASPECT ORIENTED PROGRAMMING

3.1 Aspect Oriented Programming Overview

The work on aspect-oriented programming emerged from the goal of making it possible to handle complex design structures in software implementations. Kiczales et. al's work is the originating study on AOP [1]. They focused on the issues of crosscutting concerns or aspects of a system, and offered a special-purpose AOP language. The attention shifted to making a general-purpose AOP language, and the same core team created the AspectJ: AOP implementation for the Java programming language [28].

AOP affect the software using two different approaches depending on the underlying programming language. The first is that a hybrid, combined program is produced, valid in the original language and indistinguishable from an ordinary program to the ultimate interpreter. The second way is: the ultimate interpreter or environment is updated to understand and implement AOP features. AspectJ started with a source-level weaving but shifted to bytecode weaving in a year. It modifies the generated middle man: byte codes using the

aspects. Hence produces a combined program obeying the first way of affecting the software.

In six years, AspectJ went from early research prototype to a production ready system with a large user base. With AspectJ being pioneer, today AOP has been ported to most of the programming languages varying from dynamically typed languages (e.g. Python, Ruby) to scripting languages (e.g. JavaScript, Flash Action Script) and even mark up languages (e.g. XML) and modeling languages like UML. Lesiecki et. al's book on AspectJ [29] is one of the major works that software community sees as a de facto and we based our knowledge on.

Object-oriented programming is the motherhood of the AOP. Object-oriented technologies came with its benefits such as: reusability of components, modularity, less complex implementation, and reduced cost of maintenance to software development. OOP allows for encapsulation of data and methods specific to an object. In other words, an object should be a self-contained unit with no understanding of its environment, and the environment should be aware of nothing about the object other than what the object reveals. The goal of the class is to fully encapsulate the code needed for the concern.

Object Oriented Programming (OOP) comes with its own problems. There are cases where a class does not only handle its concern, but also must fulfill the requirements of another concern. The class has been "crosscut" by concerns in the system. Crosscutting is the situation when a requirement for the system is handled by placing code into many objects throughout the system, but the code does not directly relate to the functionality defined for those objects. Adding global requirements like timing information, authentication, or logging

introduce indirect code that does not relate to the functionalities of the class. Hence, OOP modularization fails in handling crosscutting concerns. This mixing of concerns leads to a condition called code scattering or tangling.

Aspect-oriented programming emerged to address the croscutting problem faced in the Object Oriented Programming. This paradigm introduces a new modularity unit, called aspect, to encapsulate the cross cutting functionality. Hence, Aspect-oriented programming is introduced with two fundamental goals:

1) Allowing separation of concerns as appropriate for a host language.

2) Provide a mechanism for the description of concerns that crosscut other components.

AOP is not meant to replace OOP or other object-based methodologies. Instead, it supports the separation of components, typically using classes, and provides a way to separate aspects from the components.

AOP model introduces new terms joinpoint, pointcut, advice and aspect for handling crosscutting concerns. A joint point is a point in the control flow of a program. Any key points in dynamic call graph can be a joint point. Some of the joint points are:

- method and constructor calls
- method and constructor execution
- field get and set
- exception handler execution

- static and dynamic initialization

A pointcut matches a set of join points and puts a predicate on them. Pointcuts can match or not match any given join point and can pull out some of the values at that join point. Pointcuts are composed of joint points with predicates, using and (&&), or (||) and not (!).

An advice is the procedure that is to be applied at a given join point of a program. Whenever the program execution reaches one of the join points described in the pointcut, a piece of code associated with the pointcut, the advice, is executed. An advice procedure can run before or after the pointcut execution. Furthermore the advice can run around the pointcut execution: it runs in place of the join point it operates over, rather than before or after it.

An aspect is the unit that consists of a number of advices declared on pointcuts, and structures of the programming language it is implemented on. In AspectJ, aspects can have fields, classes, interfaces and many other Java like structures inside. Hence an aspect in AOP can be correlated to a class file in Java. Another element of an aspect is an open class(i.e. inter-type declaration) declaration. Open class declarations provide a way to express crosscutting concerns affecting the structure of the module, hence an aspect can insert Java elements (e.g. fields, methods, implements declaration) to a Java class.

An aspect realizes crosscutting concerns using the advices declared in them. Using joint points, pointcuts, advices and aspects, AOP model gives the user to apply cross cutting concerns on the projects.

3.2 Introduction to Aspect Oriented Programming implementations of

Design Patterns

AOP methodology brings the new concept of crosscutting concerns which modularizes the general behavior that is scattered and removes the additional code that does not relate to the direct behavior of the class. Like every code, design pattern code can benefit from what the AOP methodology brings.

Hannemann et. al.'s work on AspectJ implementations of the GoF design patterns show that in 23 of the GoF patterns, 17 of them benefit from the modular improvements. The improvements vary from one pattern to another but in general:

- better code localization: all dependencies between patterns and participants are localized in the pattern code.
- increased reusability: less scattering in the project code that's using a design pattern
- composability: multiple patterns can have shared participants, which are again modularized
- pluggability: existing classes can use a pattern instance without any modifications; all the changes are made in the pattern instance. This makes the pattern implementations relatively pluggable [30].

Aside from the general improvements on the patterns, there are some specific improvements that apply to the Java language. Java does not allow multiple-inheritance, but some of the design patterns use multiple-inheritance in their implementations. Although these patterns can still be implemented via interfaces, multiple-inheritance brings flexibility. The open class mechanism in

AspectJ allows to attach code to both interfaces and implementations, hence enables multiple inheritance in Java.

In addition to code benefits, the modularity of the design pattern implementation also results in ease on documentation. A design pattern code is contained in the same module which results in the localization of the description of a pattern instance. Hence the programmer can easily document the classes using a pattern.

CHAPTER 4

REFACTORING

4.1 Refactoring Overview

The Etymology of the word “refactoring” comes from the notion of factoring in mathematics. In the polynomial factorization, $x^2 - 2x - 8$ can be factored as $(x + 2)(x - 4)$, revealing an internal structure that was previously not visible (such as the two roots at −2 and +4) [31]. Similarly, in software refactoring, changing the visible structure often reveals the "hidden" internal structure of the original code.

Martin Fowler's book written in conjunction with Kent Beck is the classic reference for the refactoring [32]. Although that Smalltalk Refactoring Browser was well known and widely used by smalltalkers at that time, the Fowler’s work on gathering well known code smells and refactorings was the first major work.

Fowler describes refactoring as making the software easier to understand and modify. One thing to note is that the behavior of the project does not change

while refactoring the code. The software still carries out the same functionality as it did before. Kent Beck describes this with a hat analogy [33]. The programmer wears three different kinds of hats: a tester hat which the user writes the unit tests of the requirement that possibly will fail at first; a coder hat which the programmer implements the code that should make all tests pass; and a refactoring hat which the programmer refactors and makes the software easier to understand and modify. So during the development the programmer continuously switches his hat in this order.

Fowler points out the code that should be refactored as a stinking code. So bad smells adds up and make the code stink. We can describe bad smells as procedures that are violated in the project. Code smells are the symptoms of the disease. Some of the bad smells are: duplicated code, long method, feature envy, primitive obsession, and switch statements.

Fowler brought together the refactorings under the hood as catalog of refactorings. He categorized refactorings as follow:

- Composing Methods: composing methods to package code properly. e.g.: Extract Method, Inline Method.
- Moving Features Between Objects: deciding where to put responsibilities. e.g.: Move Method, Move Field, Extract Class.
- Organizing Data: refactorings that make working with data easier. e.g.: Self Encapsulate Field, Replace Data Value with Object.
- Simplifying Conditional Expressions: e.g.: Decompose Conditional, Introduce Null Object

- Making Method Calls Simpler: explores refactorings that make interfaces more straightforward e.g.: Rename Method, Add Parameter, Remove Parameter.
- Dealing with Generalization: mostly dealing with moving methods around a hierarchy of inheritance. e.g.: Pull Up Field, Pull Up Method, Push Down Method.

Automatic code refactoring is now widely implemented in the most of the current integrated development environments (IDE). The first IDE that included automated refactorings is the Smalltalk's refactoring browser. Actually the refactoring feature of the IDE's are still called refactoring browsers.

One thing that goes along with the refactorings is the tests. It is advised that the tests for the code should be written beforehand to avoid hatching bugs. The written tests will enthusiast the programmer for refactoring, whereas if tests are absent, then generally the programmers avoid refactoring not to break any code that is already working. Hence as eXtreme Programming (XP) emphasizes test driven development [33] should go along with the refactorings.

4.2 Refactoring to Design Patterns

Refactoring has been widely used so that possible new refactoring approaches for specific fields had been emerged, like refactoring of databases [34]. One of them is Joshua Kerievsky's work on refactoring to patterns [35].

In his book Kerievsky describes refactoring of patterns with three options: refactoring to, towards, and away from patterns. Refactoring to patterns is the

case in which the pattern is applied to the full extent on the code. Whereas refactoring towards depicts the case where pattern is not fully applied, but rather applying to some level is sufficient. Refactoring away from patterns is used when the project is over-engineered, meaning that the pattern is used extensively which just complicates the design.

Similar to the previous works on refactoring and design patterns, Kerievsky tried to gather the refactoring to design patterns in a catalog. He used the conventional format for describing the refactorings: name, summary, motivation, mechanics, example, and variations. He organized the patterns as follows:

- Creation: targets design problems in the code varying from constructors to overly complicated construction logic. e.g.: Replace Constructors with Creation Methods, Introduce Polymorphic Creation with Factory Method.
- Simplification: presents different solutions for simplifying methods, state transitions, and tree structures. e.g.: Compose Method, Replace Conditional Logic with Strategy, Move Embellishment to Decorator
- Generalization: transforms specific code into general-purpose code like to removing duplicated code. e.g.: Form Template Method, Extract Composite, Replace Hard-Coded Notifications with Observer
- Protection: improves the protection of existing code without altering the behavior. e.g.: Replace Type Code with Class, Limit Instantiation with Singleton, Introduce Null Object
- Accumulation: targets the improvement of code that accumulates information within an object or across several objects. e.g.: Move Accumulation to Collecting Parameter, from Move Accumulation to Visitor

- Utilities: contains low-level transformations used by the higher-level refactorings in the catalog. e.g.: Chain Constructors, Unify Interfaces

CHAPTER 5

DP-2-AOP System

5.1 Implementation Environment and Structure

In this thesis we automate refactoring of the design patterns implemented in a project to their Aspect oriented counterparts. By AOP counterparts, we mean the same design patterns implemented using aspect oriented programming, and by refactoring we mean removing the old design pattern implementation and injecting the design pattern implementation with aspects. For this automation process we implemented a tool called DP-2-AOP, which is an abbreviation of design patterns to AOP.

We focused on Java programming language for implementing our tool, since Java is one of the most widely used programming languages. Java is also the pioneer programming language in the aspect oriented programming paradigm with AspectJ.

We also used Eclipse [36] platform as a base environment for our tool. Eclipse is an integrated development environment (IDE) that supports extensibility using plug-ins: a wrapped component which conforms to Eclipse's plug-in

contract. The basic mechanism of extensibility in Eclipse is that new plug-ins can add new processing elements to existing plug-ins, and work in harmony with the other plug-ins.

In this work, we implemented our tool, DP-2-AOP, as a plug-in to the Eclipse framework. Hence we can encompass more users and benefit from the integration with other plug-ins. Also we used the "project" concept of the Eclipse: for each implementation a new project is generated. Our tool works on project base; hence we can distribute the transformations locally for each project and separate the context for DP-2-AOP refactoring.

Our tools correlation with other plugins and where it fits in the Eclipse framework can be seen in the figure below.

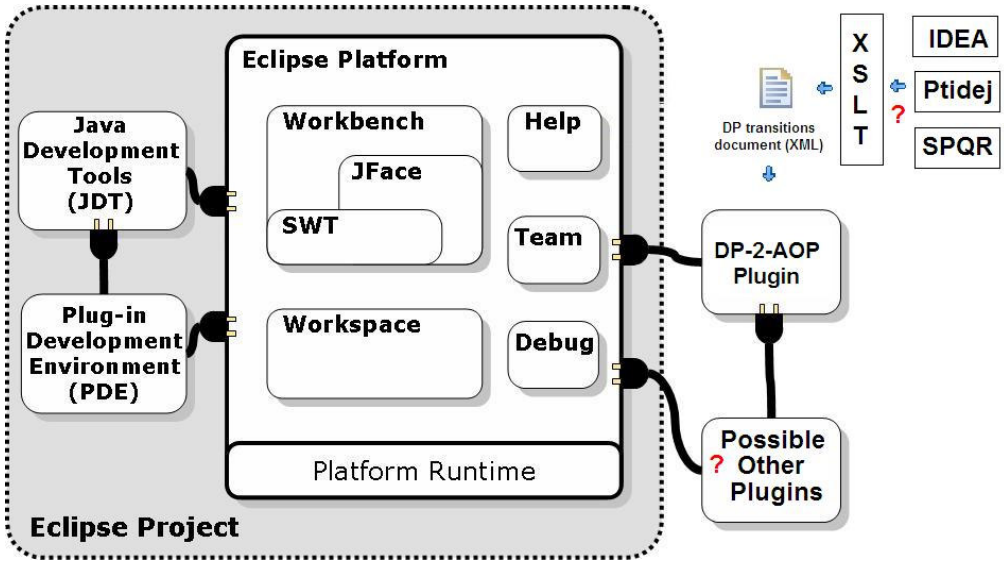

Figure 1: Correlation of DP-2-AOP with other Eclipse plugins

DP-2-AOP communicates with the Java Development Tools plug-in to understand the structure of a Java project (e.g. sources, class path), for analyzing the Java constructs (e.g. classes, methods, fields, signatures) and modifying them. DP-2-AOP uses Plug-in Development Environment for communicating with the other plug-ins.

The refactoring of design pattern implementations to Aspect Oriented implementations of a project can be partitioned into sub-tasks:

1. Finding design patterns implemented in the project
2. Documenting them so that it can be interpreted by the tool
3. Constructing the AST of the project
4. Construct the data objects mapping to design patterns, the classes roles, their interactions
5. Using the constructed data objects applying refactorings:
   a) Removing the old design pattern implementations and related code
   b) Constructing the AOP implementations of the Design Patterns

The first step requires detection of design patterns. As we presented in Chapter 2, there are tools in the literature for reverse engineering and understanding the unimplemented DP's in a project. Using those works as a base point, and dumping their results, one can apply XSLT transformations and generate the document stated in the second step. We call this document the XML configuration file (i.e DP transition document) (see Figure 1) in our project and it is an input to our tool DP-2-AOP.

For step two, we currently create the configuration file manually since we mainly focused on the job of automating the refactorings, such as constructing

the transformations and applying them successfully. But we one can use Java Development Tools (JDT) [37] to implement another plug-in for generating the XML configuration file. Using input from the user, like the observer interface as an input, JDT infrastructure can be used to extract the methods, fields, signatures of a class. So the user can map the roles and generate the configuration file.

We left the configuration file generation as an extension, and focused on the hard work. But we designed our tool in a manner which these extensions can be implemented and injected to our system with ease.

We used Java Development Tools (JDT) for overcoming step three and four. Using the attributes in XML configuration file, we found the objects and classes mapping to the DPs' role. Then we generated the data classes that hold onto this mapping information. So using the outcome of step four we can apply the modifications.

Using the mapping information constructed in step four, we finally apply our refactorings in step five. We used JDT for finding the elements of the design pattern in code and used conventional text search/replace/modification techniques to apply our refactorings.

As explained we skip the finding design patterns phase, not to reinvent the wheel, since there is enough work done in the literature and there are tools that already reverse engineer and find the implemented or the unimplemented but could be applied DP's in a project. The other sub-tasks will be inspected individually for each DP we automated refactoring. The DP, the participants and transformations will be studied in detail.

To sum up, our tool replaces the object oriented pattern implementations with their AOP counterparts. The structure of the tool will be explained in detail in the following sections with the class diagrams. Using DP-2-AOP we worked on two, Observer and Singleton, design patterns. We will get into detail on how we handled specific issues about the patterns in the following sections.

5.2 The Observer Pattern

Observer design pattern is one of the most widely used design pattern. Observer Pattern reduces coupling between a data producer and its consumers. This separation is achieved trough a notification mechanism upon a change in the data. Observer pattern is also known as Publish-Subscribe, because a publisher object opens up a subscription desk (a method), where subscribers can register for a change and in return notified when a change occurs in the Publisher.

There are four participants in an observer pattern:

- Subject: The abstract interface for attaching and detaching "Observer" objects. The abstract producer of the data or data change.
- Observer: The unifying abstract interface for updating when a notification occurs. The abstract consumer of the data.
- Concrete Subject: The concrete class extending Subject which defines where and when a notification will be send. This is the "Publisher" object that stores the state and data that the observers interested in.
- Concrete Object: The concrete object that obeys the "Observer" contract that modifies its state upon "Subject" notification.

There are two different implementation ways of Observer Pattern that we studied. The push model implementation of Observer Pattern pushes the Subject to the notification method as a parameter, and the Observers use the parameter for their needs. Whereas in the pull model, the Observers save the Subject as a class' field and pull the information using this field.

The pull model might make observers less reusable, because Subject classes make assumptions about the Observer classes that might not always be true. On the other hand, the pull model may be inefficient, because Observer classes must ascertain what changed without help from the Subject [4]. The push and pull model will be investigated in detail in the section 5.2.2.

5.3 Singleton Pattern

Singleton design pattern is the next pattern we automated. Singleton design pattern is used when we need to ensure that only one object is initialized and that object is used by the whole system. It is important that some classes should have only one instance, since synchronization problems can be seen otherwise, like only one file system should exist in an operating system or there is only one session variable that holds onto the session specific attributes. Using one and only global variable through out the system works, but it does not ensure that only one copy of an instance exists. A better solution is that during the first object initialization the first and only instance is created. When another object is tried to instantiate that object instance is returned.

Singleton design pattern lets the class itself deal with for ensuring that it is a singleton. Contrary to observer design pattern, singleton design pattern deals with one class only. Yet the class itself deals with its number of instances.

5.4 The DP-2-AOP System

We implemented DP-2-AOP system to automatically refactor existing Java design pattern implementations to their aspect oriented counterparts. In this respect, we implemented automation of two of the well known and used design patterns. We designed DP-2-AOP as flexible as possible so that we can add other design pattern automations easily. The figure below shows the UML diagram of our system:

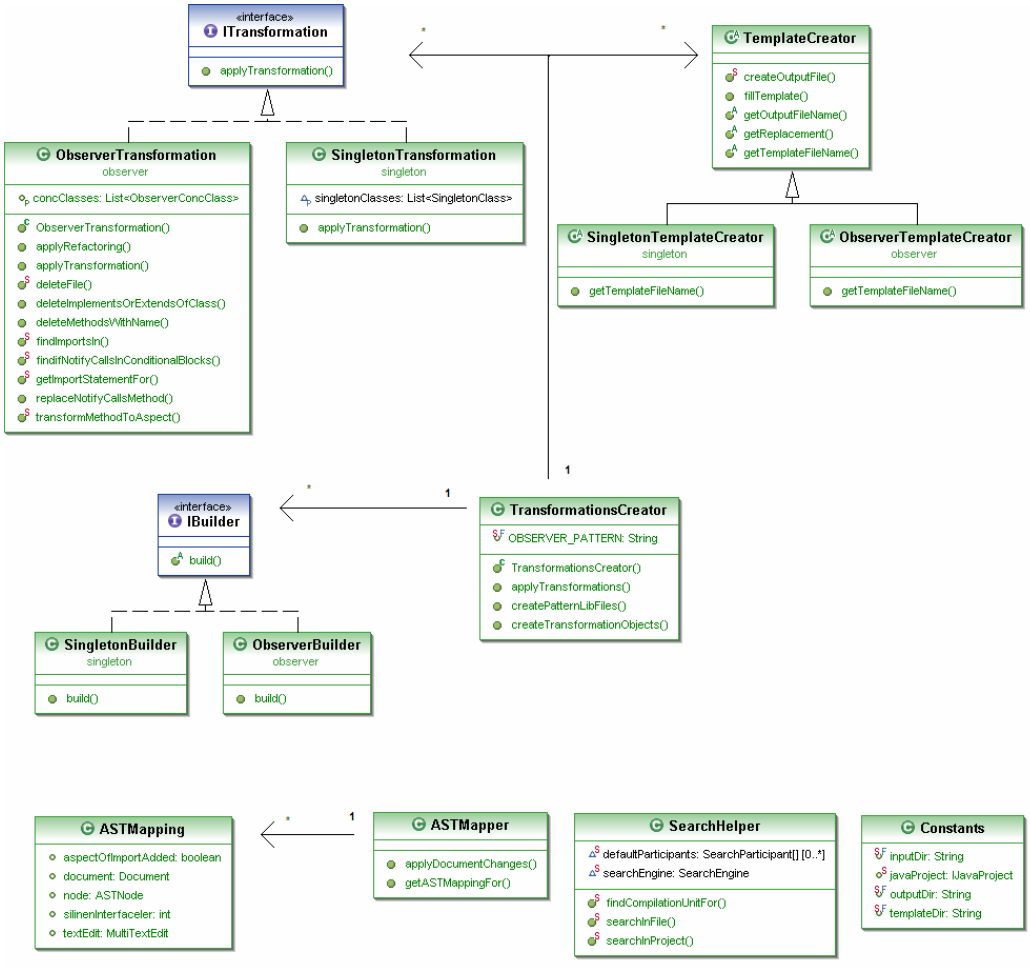

Figure 2: The UML diagram of the DP-2-AOP

As you can see from the UML diagram, TransformationCreator class creates builders and calls their build method with transformations document as a parameter. The builders create the objects that map to the design pattern and the roles of it. The template creator creates the template aspect files with the variables in them. Transformation classes traverse over these variables and fill them in. ITransformation classes do the transformations and apply the changes that should be done on the Java and AspectJ classes.

For adding another design pattern transformation to our system, first we add a builder class. The builder classes build the objects that map to the state of the pattern. Then we implement a template creator which creates a template aspect file that reflects the AOP implementation of the DP. Last we add a transformation class to fill in the template aspect file and modify the source Java files that removes the DP code from them. Below UML diagrams shows the class diagram of the Observer package. The diagram depicts all of the elements we talked about.

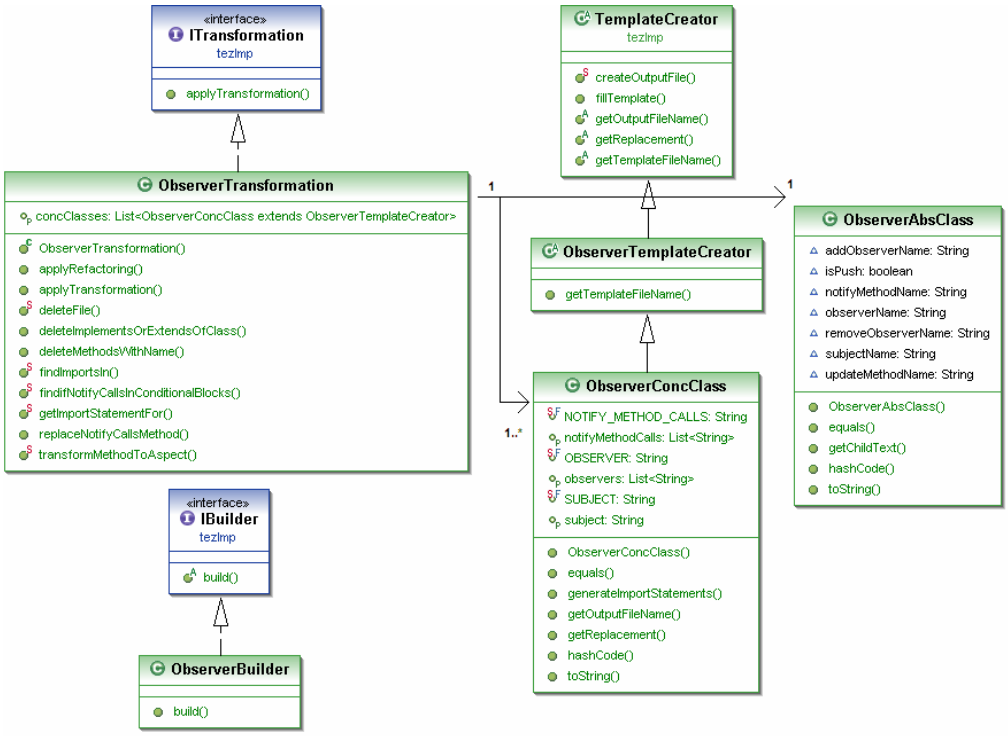

Figure 3: The UML diagram of the observer package which handles the functionalities of DP-2-AOP for the Observer DP

As you can see, the ObserverAbsClass and ObserverConcClass are the classes that map to the DP's roles. The ObserverConcClass objects fills in the corresponding aspect template files. The reason is that we create an aspect file

for each observer concrete class in the AOP implementation. Hence ObserverConcClass extends from ObserverTemplateCreator.

The builder builds these representation classes and generates their state using the information in the transformation XML document. The transformation class uses these built classes to apply the refactorings in the existing code and fills in the corresponding aspects.

5.5 Observer Design Pattern Refactoring

Refactoring Observer Pattern to an AOP implementation requires generating the AspectJ files and removing the old object oriented Observer Pattern implementation from the code. Using a case study we implemented which is about 1500 lines of code (LOC), we will describe the refactoring of an Observer Pattern implementation below.

Consider a weather station system. In this system there are several interceptors around that collect data about Humidity, Pressure, Wind, Temperature and many other weather related data. There are also display units which interpret these data and display them accordingly (e.g.: Current Condition, Forecasting). Hence when a modification occurs in the data, the data classes inform the display units. Upon notification, the display classes evaluate the data change and modify their displays accordingly. We will show the transformation to the implementation of observer with aspects through this example project, which we also used as a test project input for our tool. A possible UML diagram that focuses on Observer pattern application of the system can be seen in the figure below:

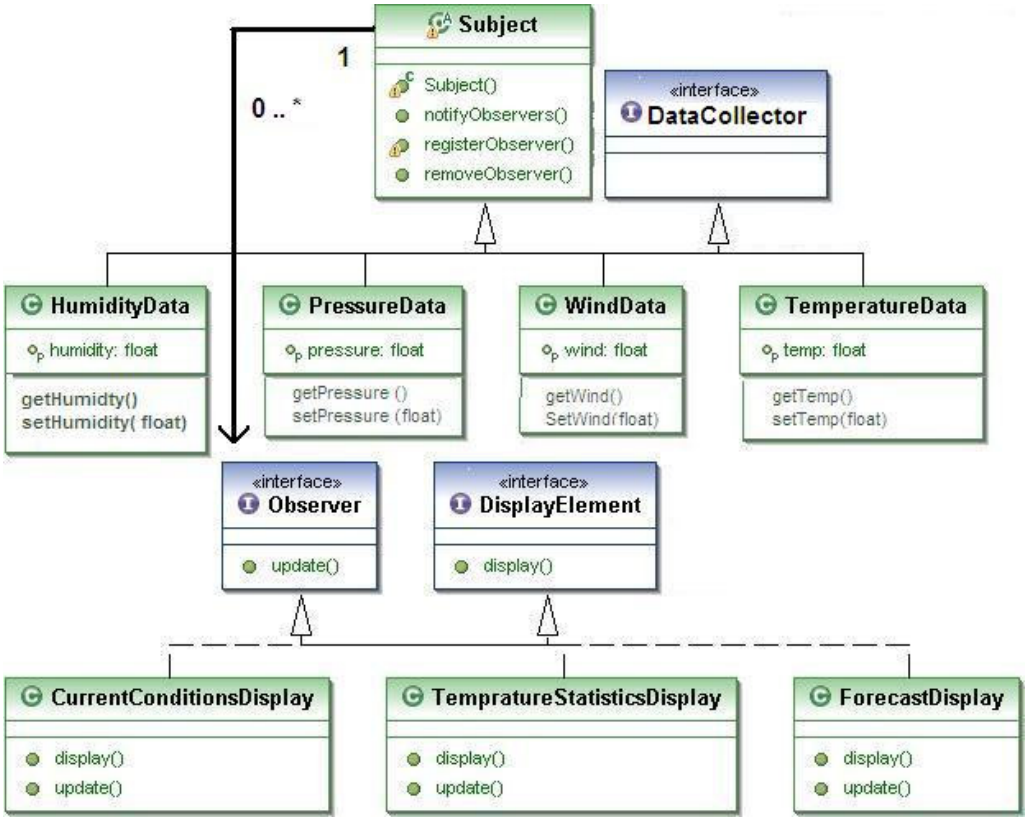

Figure 4: The UML diagram of the Observer pattern applied on weather station system

Upon a data change in the setter methods of the weather data objects, the notiftyObservers() method is invoked, which in return triggers the observers' update that are interested in that publisher.

Observer pattern brings the separation of objects such that any object that obeys the "Observer" contract (an object implementing Observer interface) can attach itself to any Subject that obeys the "Subject" contract at runtime. Although observer pattern is a breakthrough in the computer science era, aspect oriented programming brought a new point of view to observer pattern. In the

object oriented observer pattern implementations the classes ought to do more than their job. The job of the HumudityDataCollector class is organizing the hardware and collecting humidity data. Collecting observers and informing them is not the real functionality of the class. So the class tries to lift more than its weight, as you can see in the example code below:

```
class CurrentConditionsDisplay{
…
      public void update(MySubject data) {
            if (data instanceof HumidityData) {
                  humidity = ((HumidityData)
                  data).getHumidity();
            }
            else if (data instanceof TemperatureData)
            {
                  temperature = ((TemperatureData)
                  data).getTemp();
            }

            display();
      }
}
class HumidityData {
…
      public void setHumidity(float humidity) {
            this.humidity = humidity;
            notifyObservers();
      }
}
abstract class MySubject {

…
      public void notifyObservers() {
            for (int i = 0; i < observers.size(); i++) {
                  MyObserver observer =
(MyObserver)observers.get(i);
                  observer.update(null);
            }
      }
}
```

Figure 5: Code from the Observer pattern applied project

From the UML and the example code, you can see that code for implementing observer pattern is spread across the classes. All participants should know about their roles in the pattern and consequently have pattern code in them. Adding or removing a role from a class requires change in that class. Changing the notification mechanism (such as switching between push and pull models) requires changes in all participating classes. Where as in the AOP implementation, the classes has no knowledge about the pattern implementation and any modification related to the pattern implementation (such as adding/removing observers, changing push/pull model) can be done in the AOP code.

Our tool, DP-2-AOP, uses an XML template for applying the refactoring. In the XML document, one can enter as many Observer pattern transformations as he wants. DP-2-AOP searches for the "transformations.xml" document in the root of the Eclipse project that the refactoring will be done. Hence for each project, different transformation documents can be prepared and easily separated. For the given example, a transformation document for the observer pattern can be seen below. As you can see from the transformation document, the roles are mapped to the classes in the document. For each observer pattern there is one abstract class interface and any number of concrete classes that the user maps to. User also documents the model (push or pull) they use for the observer pattern.

```
<AJimpofDP>
        <project> tezTestProj </project>
        <transformations>
            <!-- Observer Transformation-->
            <transformation type="observerPattern">
                     <abstractClasses>
                             <subject>MySubject</subject>
                             <addMethod>registerObserver</addMethod>
                             <removeMethod>removeObserver</removeMethod>
                             <notifyMethod> notifyObservers </notifyMethod>

                             <observer>MyObserver</observer>
                               <updateMethod
                                       isPush="true">update</updateMethod>
                     </abstractClasses>

                        <concreteClasses>
                                <subject> HumidityData </subject>
                                <observer> CurrentConditionsDisplay </observer>
                                <notifyMethodCalls> public void
                                HumidityData.setHumidity(float)
                                </notifyMethodCalls>
                        </concreteClasses>

                        <!—Other concrete classes-->
                        <concreteClasses>
                        …
                        </concreteClasses>
            </transformation>

            <!—Other Transformations-->
            <transformation>
                …
            </transformation>
        </transformations>
```

Figure 6: The transformations XML document Observer pattern part for the example project

Using the data in the transformations document, we apply the following tasks:

- ObserverBuilder class builds the abstract and concrete observer class objects and assigns the attributes right attributes.
- ObserverTemplateCreator class generates the AspectJ template (see figure below) using built abstract observer classes, and creates the variables that will be filled in by the transformations

```
package aspectImpOfDP;

$imports$
/**
 * Concretizes the observing relationship for <code>$subject$</code> (subject)
 * and <code>$observer$</code> (observers). Subject changes trigger updates.
 */

public aspect $subject$Observer extends ObserverProtocol{
        /**
         * Assings the <i>Subject</i> role to the <code>$subject$</code> class.
         * Roles are modeled as (empty) interfaces.
         */
        declare parents: $subject$  implements Subject;
        /**
         * Assings the <i>Observer</i> role to the <code>$observer$</code> classes.
         * Roles are modeled as (empty) interfaces.
         * This can be written more than once
         */
        protected interface $MyObserverName$ extends Observer {
                public void $UpdateMethodName$($UpdateMethodSignatureParam$
$UpdateMethodCallParam$);
        }
        //i.e: declare parents: $observer$ implements $MyObserverName$;
        $observerList$
        /**
         * The if expressions before notifyObservers() are extracted into methods.
         * The pointcut
         * Adding a point cut for the methods
         * the preceding if expression is Extracted as a Method
         * put a point cut for extracted methodlara:
         * @param subject the <code>$subject$</code> acting as <i>Subject</i>
         */
//i.e.: pointcut notifyMethodConditionChange(Subject s):
//              ... && target(subject);
        $IfPointCuts$
        /**
         * Specifies the join points that represent a change to the
         * <i>Subject</i>. Captures calls to <code>$NotiftMethod$</code>.
         * @param subject the <code>$subject$</code> acting as <i>Subject</i>
         */
        protected pointcut subjectChange(Subject subject):
                ($notifyMethodCalls$) && target(subject);
        /**
         * Defines how <i>Observer</i>s are to be updated when a change
         * to a <i>Subject</i> occurs.
         * @param subject the <i>Subject</i> on which a change of interest occured
         * @param observer the <i>Observer</i> to be notifed of the change
         */
        protected void updateObserver(Subject subject, Observer observer) {
                $MyObserverName$ myObserver = ($MyObserverName$) observer;
                myObserver.$UpdateMethodName$($UpdateMethodCallParam$);
        }
}
```

Figure 7: Observer Template generated and filled in. The variables are the ones starting and ending with '$' sign

- ObserverTransformation class applies the transformations. It applies the following automatic transformations:
    - For each concrete class fill in the generated AspectJ template with the built abstract and concrete observer class objects' attributes
    - Generate the "import" statements for the aspect template files using the project AST
    - Generate an UpdateMethodsCollection aspect that will include the update methods of the concrete observers.
    - Search for the update methods in the project using AST, then transform the methods to AspectJ format, apply the push/pull model correctly on the aspect
    - Move the update methods to the generated UpdateMethodsCollection aspect
    - Find add/remove observer methods in the project using AST and replace it with the aspect call : Observer.aspectOf(), add imports statements for the aspectOf() call
    - Search for the notify method calls and remove them from the pure Java classes
    - Search and remove the notify, add/remove observer, and update methods
    - Delete the implements or extends of the abstract subject and observer classes
    - Delete the abstract subject and observer classes.

5.5.1 Observer Pattern after Refactorings

AOP solves the problem of scattering by extracting the Subject and Observer roles out into aspects. The scattered notification calls are extracted and unified in the aspects. The following figure shows the sample code after applying the

refactoring.

```
class CurrentConditionsDisplay:
// update(MySubject data) method removed
```

```
class HumidityData {
      public void setHumidity(float humidity) {
            this.humidity = humidity;
            // notifyObservers() call removed
      }
}
```

```
abstract class MySubject : class removed
```

```
public aspect HumidityDataObserver extends ObserverProtocol {
      declare parents: HumidityData  implements Subject;
      protected interface IHumidityDataObserver extends
      Observer {
            public void update(Subject subject);
      }
      declare parents: CurrentConditionsDisplay implements
IHumidityDataObserver;
      protected pointcut subjectChange(Subject subject):
            (call(public void HumidityData.setHumidity(float)))
&& target(subject);
      protected void updateObserver(Subject subject,Observer
observer){
            IHumidityDataObserver myObserver =
      (IHumidityDataObserver) observer;
            myObserver.update(subject);
      }
}
```

```
public aspect UpdateMethodsCollection {
public void CurrentConditionsDisplay.update(Subject data) {
            if (data instanceof HumidityData) {
                  humidity = ((HumidityData)
                  data).getHumidity();
            }
            else if (data instanceof TemperatureData)
            {
                  temperature = ((TemperatureData)
                  data).getTemp();
            }
}
            display();
      }
```

Figure 8: The code after the refactoring

As you can see from the code, the observer pattern code is removed from the Java classes, and it is collected in the aspects. This relocation brings separation of concerns, which means that every class only deals with its job. The following figure shows the UML diagram of the system after the refactoring.

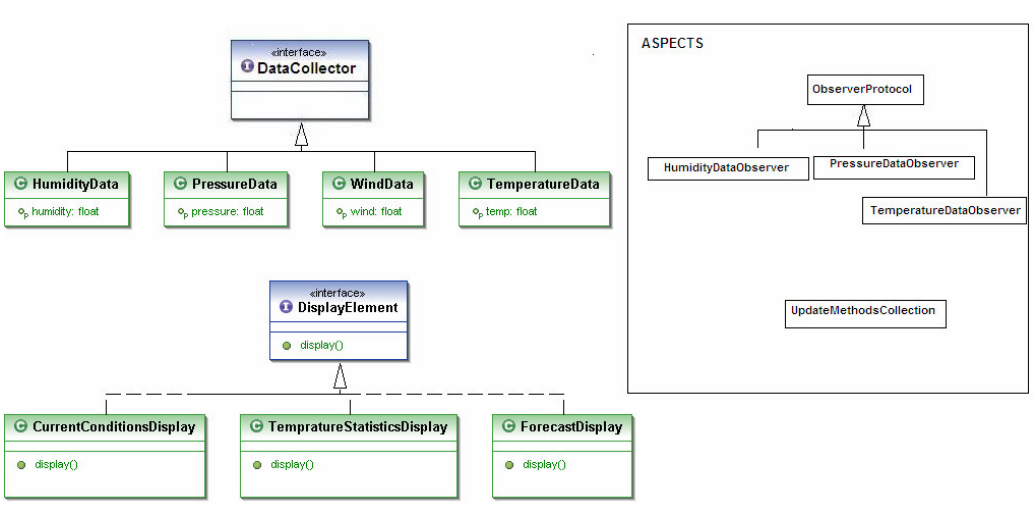

Figure 9: UML Diagram of the Observer Pattern Example after refactoring

5.5.2 Difficulties and Challenges

We used Hannemann et. al.'s [30] work as a basis in our work. In their work they implemented the design patterns in Java and AspectJ, and compared the enhancements per pattern. Although they implemented the patterns for both of the methodologies, their work is incomplete when we look from the automation perspective. For the automation process to work correctly, we should consider every possible implementation of the patterns. Whereas Hannemann et. al.'s work focuses only on a typical way of an implementation.

One of the challenges that we encountered is as follows. In some implementations of observer pattern a conditional may occur before the

notifications sent. Suppose that in the weather station system, the wind often fluctuates but the change is negligible. Hence the displays do not change if the wind change is within the negligible boundaries.

For this first challenge, AOP can not solve this problem as-is, since the notification method can be called anywhere in the code some of which cannot be picked by a pointcut. We have to shape the problem so that we can solve it with aspects. In order to do so, we used “extract method” refactoring to extract the conditional statement in the “if” or “switch” clause. Hence we get a method with the return statement as the conditional statement. Then we can use the joinpoint of calling this method to put an after advice in order to use the evaluated expression for our notification calls. We also modified the ObserverProtocol and created an abstract notifyMethodConditionChange pointcut, where the extending aspects define this pointcut and add the extracted conditional method calls to this pointcut.

Extracting the conditional checks for notification calls to the methods has another benefit. Usually the conditional statements that check the notification calls are similar, and most of the time the same. With the presented approach the user sees all the conditional statements together which can lead to better manual refactorings, such as the combining the conditional methods into one method.

Another challenge was that Hannemann et. al. moved the update method into the subjects’ aspect and used this code for updating the observers upon a notification. There is a drawback with this approach. If an Observer is observing more than one Subject then the update method is duplicated over the subjects’ aspect codes. This approach leads to code duplication smell, where

for an atomic modification the user should modify different places in the project. Hence the code becomes vulnerable to change.

For solving the second challenge, we extended the “Observer” that is defined in the abstract ObserverProtocol aspect. For each different observer, we created an interface extending “Observer” with an update method. All the update methods are then collected into a different aspect named as UpdateMethodsCollection, and the methods are transformed into aspects with AOP open class mechanism. With this approach the user now sees all the update methods in one aspect, and the code duplication is prevented. Also the place where the users modify upon a change is mostly the update methods, since the other parts are mostly automatically generated. So within one aspect we collect the code that will change most of the time. Another flexibility that this approach brings is that since for each observer we create an interface extending Observer, user can modify this interface and add different functionalities to it. Hence we make the system more robust and customizable.

The last challenge we faced with the observer pattern was the different kinds of Observer pattern implementations i.e.: push and pull models. There are two different kinds of model based observer pattern implementations:

- Push model: pushes the “subject” object as a parameter so that the update method can use this parameter to get the state of the subject. Push model should be used if the observer observes more than one subject; so that the observer can differentiate which subject triggered the notification.
- Pull model: pulls the state of the subject using the field accessor. The observer saves the subject as its field and modifies its state upon notification using this field.

Since automation involves every possible implementation, we should be able to handle both of the pattern implementations. We used an XML attribute to define the update method as push or pull model. We generate the joint points such that if implementation is push model, we give the subject as a parameter to the update method call.

Although it is not a challenge, one of the most difficult parts of our work is reverse engineering and the automation process. Although we used Eclipse as the environment and JDT (Java Development Tools) as the framework for generating the AST, we struggled with the scarce source of documentation and help.

5.5.3 Refactoring Observer Conclusion

When we compare AOP and OO methodology for Observer pattern, in the AOP implementation, the classes which have the subject role, do not have to know and hold onto the list of their observers anymore. The subjects do what they ought to do and do not deal with the observer code.

Another benefit of the AOP implementation is that the Observer pattern code is packed up in the same place, so that without making any changes in the Java code, one can add a class as a subject or observer, or remove a class from the observers vice versa. Hence separation of concerns is achieved.

5.6 Singleton Design Pattern Refactoring

We will show the transitions through a concrete example of an application singleton pattern. We used this example as a test project input for our tool as

well. Suppose that there is a drawing application, like Microsoft Paint or Paint Shop Pro. In such an application the user sees only one display and makes changes upon the same display, open a new project on the same display and such. Hence there should be only one Display object. Therefore, we should make it singleton so that the developers of our system do not make a mistake and create another Display instance. A possible UML diagram that focuses on Singleton class application of the system can be seen in the figure below:

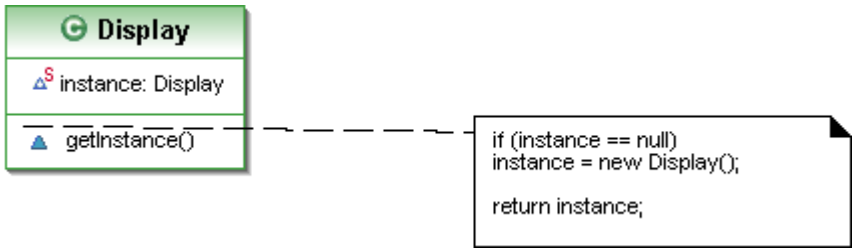

Figure 10: The Display class' Singleton pattern implementation part

To implement a class as a singleton:

- Make the constructor private or protected so that it can not be called.
- Create a static method which: creates the single instance if not initiated and then returns the single instance
- Make the single instance static so that it can be instantiated in the method returning singleton

The following figure shows source code how to apply singleton pattern with Object Oriented methodology:

```
public class Display {

	//The singleton instance:
	static Display instance;

	//A private constructor so that it can not be called from
outside
	private Display(){

	}

	//The static getInstance method for getting the singleton
	static Display getInstance(){
		//Instantiate the instance if first call
		if(instance == null)
			instance = new Display();

		return instance;
	}

	//Other code about drawing, refreshing, painting …
	…
}
```

Figure 11: Code from the Singleton pattern applied project

5.6.1 Singleton after Refactorings

Just like the observer pattern, in the singleton pattern the class lifts more than its weight. The class itself deals with ensuring that there is one and only one object of itself in the heap space. When all these new responsibilities add up, the class becomes a junk one which tries to handle all these features. The AOP solution again solves this problem gracefully. The following figure shows the code after applying the AOP solution:

```
public class Display {

      //The constructor is public and callable from outside
      public Display(){

      }

      //All the code about the singleton DP implementation is
removed
}


public aspect DisplaySingleton extends SingletonProtocol{

      //inject Display class to implement Singleton
      declare parents: Display implements Singleton;

      //add a pointcut on the constructor call
      protected pointcut protectionExclusions():
            call((DisplaySubclass+).new(..));

}

public abstract aspect SingletonProtocol {

      ...
    //The pointcut defined in the extending aspects
    protected pointcut protectionExclusions();


    //The around call to the constructor & return the
singleton instance
Object around(): call((Singleton+).new(..)) &&
!protectionExclusions()
      {
      Class
singleton=thisJoinPoint.getSignature().getDeclaringType();

      if (singletons.get(singleton) == null) {
                singletons.put(singleton, proceed());
            }
            return singletons.get(singleton);
      }
```

Figure 12: The code after the Singleton Pattern refactoring

As you can see after AOP transformation, all the singleton code is removed from the class, and it is inserted into an aspect. As a result, the class does not have to control making itself singleton. This leads to one of the main design principles: separation of concerns. Every class should do what it ought to do, no more no less. The following UML diagram shows the system after the refactoring:

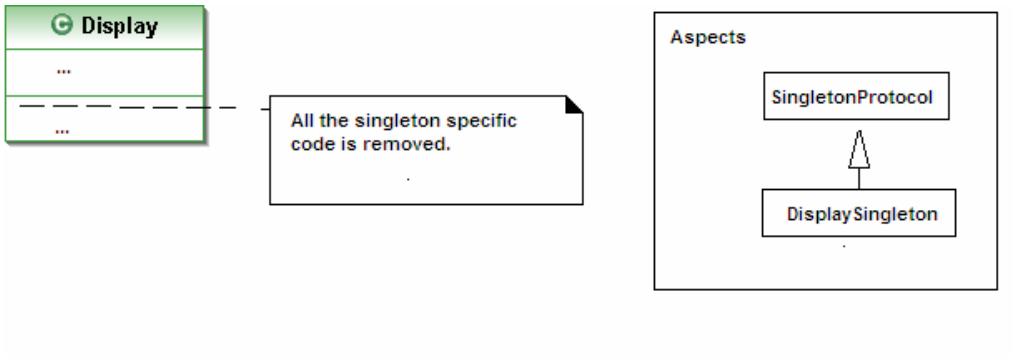

Figure 13: The UML diagram of the system after Singleton Pattern refactoring applied

5.6.2 Singleton Conclusion

When we compare AOP and OO methodology for Singleton pattern, AOP implementation removes the singleton code from the class and moves it into an aspect. Making the extending classes of the singleton class again singleton takes just one line of code modification in the corresponding aspect. An additional benefit of AOP solution is making a class or removing a class from being singleton can be done by modifying the aspect code only.

CHAPTER 6

Analysis

6.1 Analysis Overview

Since the innovation of the software, the programmers questioned its maintainability and testing. As Tom DeMarco says so “You can not control what you can not measure" [38]. Software metrics measure the quality of the software. There are different scales for computing software metrics. One and probably the most general one is the complexity analysis of the software. Assessing complexity of software took the attention of the computer science community since it basically uses mathematical techniques for getting results.

McCabe's work is one of the first major works on software complexity. He named his metric scale as "Cyclomatic complexity". Cyclomatic complexity measures the linearly independent paths of software [39]. It is computed using control graphs which shows different control flows of the program. In the graph the nodes are the atomic commands of the program. The directed edges of the graph are transitions between commands which the software might execute one immediately after the other. Using this graph the cyclomatic number V (G) of a graph G with n vertices, e edges, and p connected components is v (G) = e -n + p. A simpler calculation of the cyclomatic

complexity can be computed by counting the predicates in the graph or by counting flow graph regions [40]. McCabe states that the cyclomatic complexity of software should be at most 10.

McCabe showed the control graph generation and cyclomatic complexity calculation using the FORTRAN software language. FORTRAN is a non-structural language and his work lacks the nesting level of the predicate nodes. In other words, in McCabe's work there is no distinction between two distinct if statements and two nested if statements, both of them increase the cyclomatic complexity by 2. Piwowarski [41] improved cyclomatic complexity by including the nesting level into play. Howatt et. al redefined the nesting level and add the structural language support to the cyclomatic complexity.[42] Chidamber et al later on generalized the cyclomatic complexity work for object oriented systems [43].

In this thesis, we worked on two different paradigms. Our input projects have two phases: an initial pure object oriented system and a final aspect oriented system. Hence we should use a metric system where a scale for comparing both of the paradigms exists.

Yann et.al's work is at the early stage for evaluating different paradigms' metrics. They proposed a methodology only for calculation [44]. Late works of Pataki et. al is promising works that we can use for assessing metrics in our project[45,46,47,48]. Their work is on Multi-paradigm Software Complexity Metrics, which is a metric system for systems that might bring together different paradigms in software. In their work, they used OO and AOP paradigms to discuss their method.

Pataki et. al's work uses complexity of nested control structures, as basis. But they also added complexity of the data components to the control graph. In a nutshell, they described new nodes for the control graph: data nodes. The nodes map to the data elements of a class. The directed edges outgoing from data nodes are "read" of the data, and incoming to the data nodes are "write" of the data. So a statement like "x = x +1" where "x" is an integer type generates both an incoming and an outgoing edge from the data node. They call this new graph the AV graph. This graph can be modeled to the Howatt's model by adding a reader control node before the node reading data and add a writer control node before the node writing data. The complexity of the data nodes (primitive e.g.: integer, char or an Object) does not affect the graphs they are used since even if it represents a complex data type, its definition should be included in the program and its complexity is counted there. The complexity of a class is the sum of the complexity of the methods and the data members (attributes).

The AV complexity of a program is a sum of the following three components:
a) The control structure of program. This is the general control structure calculation of software. They used Howatt's nesting level to weight the statement nodes. The control statements do no change according to the paradigm used.
b) The complexity of data types. This reflects the complexity of data used for classes.
c) The complexity of data access. This reflects the connection between control structure and data.

The code on the next page shows an example AV-graph mapping of a class and its elements:

```
class Date
{
public:
    void set_next_month() {
        if ( month == 12 ) {  month = 1; year = year + 1; }
        else                  { month = month + 1; }
    }
    void set_next_day() {
        if ( month == 1 || month == 3 || ... || month == 12 )
            if ( day == 31 )  set_next_month();
            else              day = day + 1;
        else
            if ( day == 30 )  set_next_month();
            else              day = day + 1;
    }
private:
      int year, month, day;
};
```

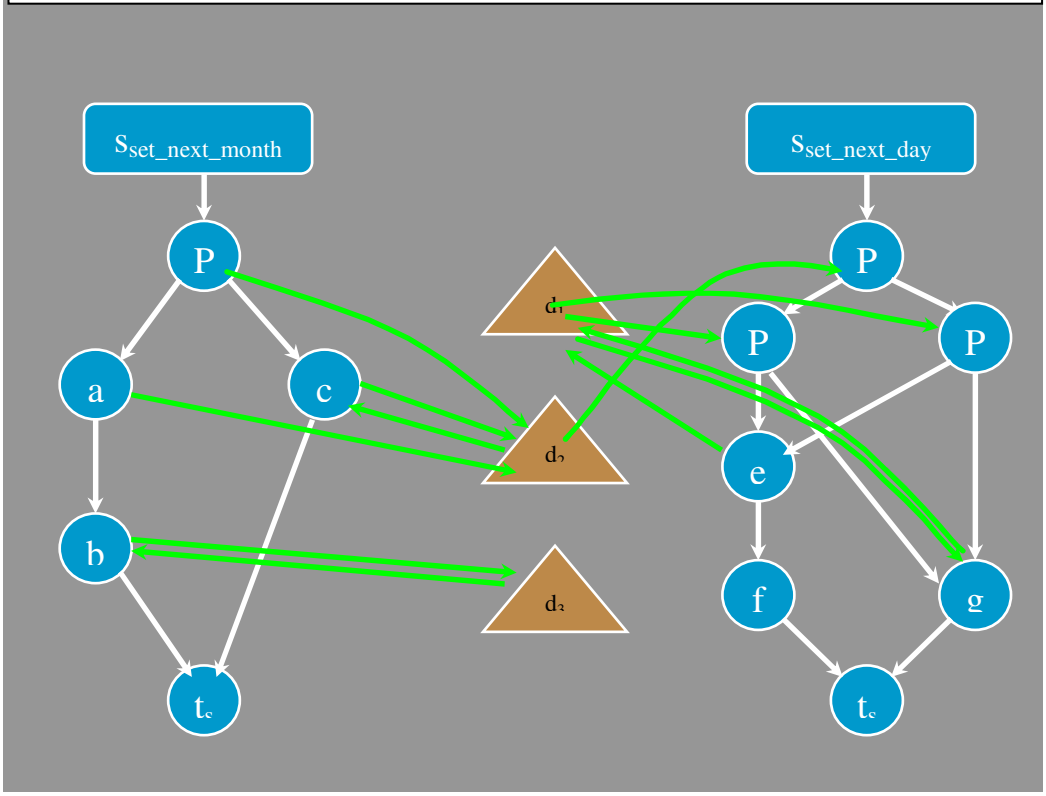

Figure 14: The Data class and mapping of its members on its AV graph

Pataki et. al. claim that by including data node calculation to the complexity issues, they sailed to the paradigm-independent notions. Hence they can apply their measure to procedural, object-oriented, aspect oriented or even mixed-style programs. They extended their metric to generate AV-graph for AOP elements.

Joint points are joints in the software where aspect oriented interception can be applied. Pointcuts are a collection of joint points brought together with conditionals. Advice is a pointcut and the functional part where applied upon a pointcut match in the software. Aspects are class like structures which includes other AOP elements and inter-type declarations.

As we pointed out in section three, the key elements of AOP are joint points, pointcuts, advices and aspects. They evaluated the key elements of AOP and mapped them to the elements of an AV graph. The signature of the joint points is like regular expressions. Therefore in the AV-graph, they mapped pointcut definitions to predicate nodes, and the pointcut type and the signature to input nodes. The complexity of a joint point is the sum of the complexities of the signature. Hence the complexity of a pointcut is the sum of the pointcut definitions' complexities. The pointcut part of an advice is calculated as described and the function parts complexity is measured the same way as Java methods. The complexity of an advice is the complexity of advice's body multiplied by the complexity of its pointcut. Aspects and classes have a lot in common from the complexity point of view. So the complexity of an aspect is the sum of the complexity of its elements.

6.2 Simple Approach

Pataki et al's work is one of the scarce works we can use for measuring and finding out the effectiveness for this thesis. Although their work is a good resource, absence of a tool support is a drawback. Drawing the AV graphs and measuring the multi-paradigm complexity for the whole projects is a hard task. Instead, we used a different approach to apply their work. Since our project is a refactoring process, both the initial OO implementation and the final AOP implementation share some code. We will not be calculating the multi-paradigm metric value of these overlapping code portions. Instead we will calculate the multi-paradigm metric of the OO code we removed and AOP code our tool add, and take the difference. If the difference is a positive value, our tool decreases the complexity, if it is 0 the complexity does not change, otherwise the complexity increases.

As stated in the recent work of McCabe, they state that in addition to counting predicates from the flow graph, it is possible to count them directly from the source code [40]. This often provides the easiest way to measure and control complexity during development, since complexity can be measured even before the module is complete. An "if" statement, "while" statement, and so on are binary decisions; therefore they add one to the complexity. Boolean operators (e.g.: && for Java) also add one to the complexity.

Using this method, the table below shows the multi-paradigm metrics of the refactored code. That is the metric for the removed OO code and the metric for the added AOP implementations:

	Java (Multi-paradigm metric complexity unit)	AOP (Multi-paradigm metric complexity unit)
Observer Pattern	788	734
Singleton Pattern	122	106

Table-1: The Multi–paradigm metric complexity difference for the presented projects

The results show that, our tool reduces multi-paradigm complexity of observer pattern by 54 and the singleton pattern by 16. The singleton design pattern reduction is minimal since the code change is not too much. Just the instance and method are deducted from the code, whereas a much more major refactoring occurs for the observer pattern.

Pataki et al used their metric scale on the Hannemann et. al's work [30]. When we compare with their results their observer pattern difference is about 70. The reason is that we extended and generalized their work, hence added new elements to the aspects.

As described in 5.5.2, we faced some difficulties when refactoring the Observer pattern. The Hannemann et. al's work did not offer any solution for some of these difficulties, and for the reduction of the code duplication, we offered a different approach. So we should justify that our approach offers a better solution. Hence the comparison table of their approach and our approach is below:

	Our approach (Multi-paradigm metric complexity unit)	Hannemann et. al's approach (Multi-paradigm metric complexity unit)
Observer Pattern	734	751

Table 2: The Multi-paradigm metric complexity difference between our and the traditional approach for Observer Pattern

Our approach reduces the complexity by 17 compared to the traditional way of refactoring. This is the result of the removal of code duplication that occurs in the Hannemann et. al's AOP implementation of the DP's.

CHAPTER 7

Conclusion and Future Work

7.1 Summary and Conclusion

In this work, we automated traditional design pattern implementations to their aspect oriented counterparts. Hannemann et. al. [30] implemented the Java design patterns in AOP. While automating process, we saw that there might be different kinds of design pattern implementations, and Hannemann et. al.'s work does not reflect all of them. Hence we generalized their work so that it encompasses all the design pattern implementations and automated refactoring of the DP's using the implementation specifications using a configuration file.

We chose two patterns to automate, but we worked on other GoF patterns as well. In that work, we understood that not all of the DP implementations in AOP enhance the design. Although in their work Hannemann et. al. points out that the 17 of the DP implementations in AOP enhances the locality, reusability, composition transparency and pluggability of the DP, we would not prefer the AOP implementations for all of them. For example, in the Command pattern, the command classes' role is to act as a command object and to be easily queued, and rolled back and so on. Hence removing this sole role from

the class, leaving an empty class as a hook, and moving all the functionalities to aspect part is not a good design.

We used Pataki et. al.'s multi-paradigm metric for calculating the complexities of OO and AOP implementations. We calculated the metrics for the systems before and after applying the refactoring. The analysis shows that our tool decreases the complexity, hence results in a more maintainable system. We also compared Hannemann et. al's AOP implementation and our refactoring approach. We used this metric comparison for evaluating whether our approach results in a better system. The comparison showed that our approach for removing the duplication leads to a less complex system.

From our point of view, aspect oriented implementations of DP's should be used to remove the burden of different functionality that the class should not deal with. It should not be used to remove the sole role from the class, and leave the class just like a shell as a hook. As Fred et. al. [32] remarks, AOP itself is not a silver bullet too. Similar to what object oriented programming was to a functional programming, aspect oriented programming is an enhancement to the object oriented programming. The right way to use it should be combining the powerful parts of both of the models, not to shift all of the object oriented implementations to AOP.

Observer and Singleton DP's are chosen from seeing aspect oriented programming as an enhancement to object oriented programming. In both of the design patterns, the aspect oriented programmed implementation removes the extra load from the classes, and organizes the patterns in a way such that, the DP is:

- Localized: meaning compact in one place,
- Reusable: The shared code is put into a protocol, and AspectJ extension mechanism is used to remove code duplication
- Pluggable: The methodology of the pattern, adding or removing another class to the pattern can be done easily without modification in the other Java sources. This also points out to the localization of the pattern.

7.2 Future Work

We had done some major work on automating and enhancing the design of the system. However there is a mass amount of work that can still be done in this context. We used Pataki et. al's work on multi-paradigm metrics for comparing our programs. But when applying their work, we used an easier differentiation method and counted predicates directly from the source code. This approach might be error prone since it is manual. We expected a reply for using their tool, if they have implemented any, but could not hear from them. Another point with the metrics is that the metric work on AOP is still quite new. Upon development, more robust and a safer metric tool should be applied for reassessing the metrics of the systems. A better way is using an automated tool which is accepted as the de facto complexity evaluation tool by the AOP community.

A different way to implement these AOP transformations would be using the new annotations mechanism for AOP. As of Java 1.5, it includes a new annotations mechanism, which using specific attributes metadata can be inserted to the Java source code. Likewise AOP took the same route, and added annotations mechanism to itself. When we were starting this work, AOP did

not have the annotations mechanism feature. As a future work, AOP's annotations mechanism can be used to implement DP's in AOP.

Additional work can be done for creating a better user interface. In our work we added a button to the eclipse framework using its own extension mechanism for testing our automation and refactoring results. In order to ship our work as a tool, a better user interface design with setting possible preferences can be built. Also adding different design pattern refactorings to our can be implemented.

REFERENCES

[1] Kiczales, Gregor; John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-Oriented Programming", *Proceedings of the European Conference on Object-Oriented Programming, vol.1241*, pp.220–242, 1997.

[2] Java, http://java.sun.com/

[3] Alexander, Christopher, "A Pattern Language: Towns, Buildings, Construction", New York: Oxford University Press, 1997.

[4] Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[5] Robert Cecil Martin, "Agile Software Development", Prentince Hall, 2003.

[6] Bertrand Meyer, "Object-Oriented Software Construction", IEEE Press pg 23. 1988

[7] Robert Cecil Martin, "The Open-Closed Principle", The C++ Report, http://www.objectmentor.com/resources/articles/ocp.pdf

[8] Robert Cecil Martin, "The Liskov Substitution Principle", The C++ Report, http://www.objectmentor.com/resources/articles/lsp.pdf

[9] Forrest Shull, Walcélio L. Melo, and Victor R. Basili, “An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems”, Technical Report, University of Maryland, 1996.

[10] G. Antoniol, R. Fiutem and L. Cristoforetti, “Design Pattern Recovery in Object-Oriented Software”, IEEE CS Press, 1998.

[11] G. Antoniol, R. Fiutem and L. Cristoforetti, “Using metrics to Identify Design Patterns in Object Oriented Software”, 1998.

[12] Yann-Ga¨el Gu´eh´eneuc, Houari Sahraoui, and Farouk Zaidi, “Fingerprinting Design Patterns”, 11th WCRE 2004. Working Conference on Reverse Engineering (pp. 172-181), 2004.

[13] Jason McC. Smith and David Stotts, “Elemental Design Patterns: A Logical Inference System and Theorem Prover Support for Flexible Discovery of Design Patterns”, Technical Report, 2002.

[14] Jason McC. Smith and David Stotts, “Elemental Design Patterns: A Link between Architecture and Object Semantics”, Technical Report, 2002.

[15] Jason McC. Smith and David Stotts, “SPQR: Flexible Automated Design Pattern Extraction from Source Code”, Technical Report, 2003.

[16] Jason McC. Smith and David Stotts, “SPQR: Use of a First-Order Theorem Prover for Flexibly Finding Design Patterns in Source Code” , Technical Report, 2003.

[17] Jason McC. Smith and David Stotts, “SPQR: Formalized Design Pattern Detection and Software Architecture Analysis”, Technical Report, 2005.

[18] Olivier Kaczor, Yann-Gael Gueheneuc and Sylvie Hamel, “Efficient Identification of Design Patterns with Bit-vector Algorithm”, Conference on Software Maintenance and Reengineering (CSMR'06), 2006.

[19] Lothar Wendehals, “Dynamic Design Pattern Recognition”, Dissertation Proposal University of Paderborn Warburger, 2005.

[20] Gerson Sunye, Alain Le Guennec and Jean-Marc Jezequel, “Design Patterns Application in UML”, Springer, 2000.

[21] Federico Bergenti and Agostino Poggi, “Improving UML Designs Using Automatic Design Pattern Detection”, Proc. 12th. International Conference on Software Engineering, 2000.

[22] Herv´e Albin-Amiot, Pierre Cointe, Yann-Gael Gueheneuc and Narendra Jussien, “Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together”, 16th IEEE International Conference on Automated Software Engineering (ASE'01) p. 166, 2001.

[23] Ptidej main page:
http://www.yann-gael.gueheneuc.net/Work/Research/Introduction/

[24] Yann-Gael Gueheneuc, “Ptidej: Promoting patterns with patterns. Wokshop on Building a System with Patterns”, Springer, 2005.

[25] Francesca Arcelli and Claudia Raibulet. “Program Comprehension and Design Pattern Recognition: An Experience Report”, ECOOP(European Conference on Object-Oriented Programming) WOOR(Workshop on Object-Oriented Reengineering), 2006.

[26] Francesca Arcelli, Stefano Masiero, Claudia Raibulet, Elemental Design Patterns Recognition In Java, 2005.

[27] U. Nickel, J. Niere, and A. Zündorf, “The FUJABA Environment”, In Proceedings of the 22nd International Conference on Software Engineering pp. 742-745, 2000.

[28] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, “An Overview of AspectJ”, ECOOP 2001.

[29] Nicholas Lesiecki, Joseph D. Gradecki, "Mastering AspectJ: Aspect-Oriented Programming in Java", Wiley Publishing, 2003.

[30] Hannemann, Jan and Kiczales, Gregor, "Design Pattern Implementation in Java and AspectJ", Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 161-173, 2002.

[31] Etymology Of Refactoring link:
http://martinfowler.com/bliki/EtymologyOfRefactoring.html

[32] Fowler, Martin, "Refactoring. Improving the Design of Existing Code", Addison-Wesley, 1999.

[33] Beck, Kent, "Test-Driven Development by Example", Addison-Wesley, 2003.

[34] Ambler, Scott W., "Refactoring Databases: Evolutionary Database Design", Addison-Wesley, 2006.

[35] Kerievsky, Joshua, "Refactoring to Patterns", Addison-Wesley, 2004.

[36] Eclipse, http://www.eclipse.org/

[37] Java Development Tools plugin, http://www.eclipse.org/jdt/

[38] DeMarco, T., "Controlling Software Projects: Management, Measurement & Estimation", Yourdon Press, New York, USA, p3, 1982.

[39] McCabe, T.J., "A Complexity Measure", IEEE Trans. Software Engineering, SE-2(4), pp. 308-320, 1976.

[40] Arthur H. Watson, Thomas J. McCabe, “Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric”, NIST Contract 43NANB517266
http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/235/title.htm , August 1996.

[41] Piwowarski, R.E., “A Nesting Level Complexity Measure”, ACM Sigplan Notices, 17(9), pp.44-50, 1982.

[42] Howatt, J.W., Baker, A.L., “Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting”, The Journal of Systems and Software 10, pp.139-150, 1989.

[43] Chidamber S.R., Kemerer, C.F., “A metrics suit for object oriented design”, IEEE Trans. Software Engeneering, vol.20, pp.476-498, 1994.

[44] Jean-Yves Guyomarc’h and Yann-Ga¨el Gu´eh´eneuc, “On the Impact of Aspect-Oriented Programming on Object-Oriented Metrics”, QAOOSE Workshop, ECOOP, Glasgow, pp. 42-47, 2005.

[45] Ádám Sipos, and Zoltán Porkoláb, “Comparison of Object-Oriented and Paradigm Independent Software Complexity Metrics”, ICAI’04, Eger, 2004.

[46] Ádám Sipos, and Zoltán Porkoláb, “Towards a multiparadigm complexity measure”, QAOOSE Workshop, ECOOP, Glasgow, pp.134-142, 2005.

[47] Norbert Pataki, Ádám Sipos, and Zoltán Porkoláb, “Measuring the Complexity of Aspect-Oriented Programs with Multiparadigm Metric”, ECOOP 2006 Doctoral Symposium and PhD Workshop, 2006.

[48] Norbert Pataki, Ádám Sipos, and Zoltán Porkoláb, “On Multiparadigm Software Complexity Metrics”, MaCS’06 6th Joint Conference on Mathematics and Computer, 2006.