ATHABASCA UNIVERSITY

Analyze Aspect-Oriented Software Approach and Its Application

By

Artur Krupa, Id: 2247378

An essay submitted in partial fulfillment
Of the requirements for the degree of
MASTERS OF SCIENCE in INFORMATION SYSTEMS

Athabasca, Alberta
June, 2010

©Artur Krupa, 2010
DEDICATION

I would like to dedicate this essay to my wife who supported me throughout.
The ultimate goal of software development is to create software systems, usually for a von Neumann architecture of instructions, registers, and memory locations. [7] First computers were created to perform huge number of calculations fast and efficiently. The earliest programming languages were machine oriented languages such as an assembler. The growing power of computers and better operating systems allowed computers to perform more complex tasks such as multitasking and to run complicated applications such as weather simulation software. Initial computer languages were imperative and examples of such language are Algol and FORTRAN. Imperative language has the abstractions of imperative programs such as procedures, statements, and record structures corresponding to a von Neumann architecture. The next step in evolution of programming languages was functional programming, a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data and examples of computer language using functional programming are APL or Lisp. After functional programming, computer languages started to use mathematical logic introducing logic-based programming. The examples of logic-based languages are ALF and Leda. After logic-based programming the Object-oriented Programming was introduced and become very popular. Object-oriented programming is a programming paradigm that uses objects together with their interactions to design applications and computer programs and examples of Object-oriented languages are Java and C++. In order to solve complex problems computer languages should be able to simulate complex systems. This process of evolution of computer languages can be described as moving from a machine oriented view to a problem
oriented view. The problem oriented view allowed creating programming models with better adaptability, reusability, efficiency, and easy to implement. [7]

In 1990s, Object Oriented Programming was a very popular and efficient way of creating computer software. Despite of success of the Object Oriented Programming, it had problems relating to the difficulties with the maintenance of software code. Any change to the code required updates to a large number of unrelated modules.

In 1995, a group scientist at the Xerox Palo Alto Research Center (PARC), led by Gregor Kiczales, created a theory for a new paradigm called Aspect-Oriented Programming. The new paradigm was designed to address the Object Oriented Programming problems. The main idea behind the new paradigm is separating crosscutting concerns into single units. During the development process a developer deals separately with crosscutting concerns by implementing aspects. Both code and aspects are combined into an executable form using weaving mechanism. Many scientists consider an Aspect-Oriented Programming as a complement to Object-Oriented Programming by facilitating another type of modularity that pulls together the widespread implementation of a crosscutting concern into a single unit called an aspect. The new paradigm can be implemented in many programming languages, such as Java, C++, Smalltalk, C, and C#. The process of establishing the theory of Aspect-Oriented Programming is still not completed.
ACKNOWLEDGEMENTS

Special thanks and appreciation are owing to the following individuals:

- Richard Huntrods, my advisor and paper supervisor, for all his generous and clear advices and guidance throughout the preparation of this master’s essay
- The always helpful staff and faculty of Athabasca University’s Master of Science in Information Systems program
# TABLE OF CONTENTS

CHAPTER I - INTRODUCTION ........................................................................................................1

  STATEMENT OF PURPOSE ...................................................................................................... 2
  CONTENT ................................................................................................................................. 2
  SIGNIFICANCE ......................................................................................................................... 3

CHAPTER II – THE THEORY BEHIND ASPECT-ORIENTED PROGRAMMING ...... 5

  BACKGROUND ......................................................................................................................... 5
  BASIC CONCEPTS .................................................................................................................. 7
  ASPECTUAL DECOMPOSITION AND CROSSCUTTING CONCERNS .............. 11
  MODULARITY AND SEPARATION OF CROSSCUTTING CONCERNS .......... 12
  WEAVING OR INTEGRATION ................................................................................................. 13

CHAPTER III – ASPECT-ORIENTED PROGRAMMING IMPROVEMENTS .......... 16

  SECURITY ............................................................................................................................... 16
  REAL-TIME CONCERNS ......................................................................................................... 19
  ERROR AND FAILURE HANDLING ..................................................................................... 22

CHAPTER IV – ASPECT-ORIENTED PROGRAMMING IMPLEMENTATIONS .... 25

  DEVELOPMENT IN JAVA USING ASPECTJ ................................................................. 25
  DEVELOPMENT IN JAVA USING JMANGLER ............................................................... 31
  DEVELOPMENT IN JAVA USING ASPECT WERKZ ................................................... 37
  DEVELOPMENT IN JAVA USING JIAZZI ................................................................. 45
  DEVELOPMENT IN .NET AND AOP# ............................................................................. 48
  IBM IMPLEMENTATION OF ASPECT-ORIENTED PROGRAMMING ............ 56

CHAPTER V – CONCLUSIONS AND RECOMMENDATIONS ............................. 65
REFERENCES ..............................................................................................................69
APPENDIX A ...........................................................................................................75
APPENDIX B ...........................................................................................................78
APPENDIX C ...........................................................................................................86
APPENDIX D ...........................................................................................................90
APPENDIX E ..........................................................................................................103
APPENDIX F ..........................................................................................................110
APPENDIX G ..........................................................................................................115
APPENDIX H ..........................................................................................................122
APPENDIX I ..........................................................................................................125
APPENDIX J ..........................................................................................................131
APPENDIX K ..........................................................................................................136
APPENDIX L ..........................................................................................................142
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>The Extension definition template implementation using AOP</td>
<td>57</td>
</tr>
<tr>
<td>2.</td>
<td>Examples of AOP examples and RUP equivalents and examples</td>
<td>58</td>
</tr>
<tr>
<td>3.</td>
<td>A sample of hierarchical representation of the RUP meta-model</td>
<td>61</td>
</tr>
<tr>
<td>4.</td>
<td>Examples of terms in which developers think and program</td>
<td>75</td>
</tr>
<tr>
<td>5.</td>
<td>Brief summary of comparison of Aspect-Oriented and Container Managed Security</td>
<td>84</td>
</tr>
<tr>
<td>6.</td>
<td>Change of C&amp;K metrics measures after implementing AOSim application</td>
<td>100</td>
</tr>
<tr>
<td>7.</td>
<td>Combination of Individual Results for System Properties after implementing AOSim application</td>
<td>101</td>
</tr>
<tr>
<td>8.</td>
<td>Average change over all system properties for real-time concerns after implementing AOSim application</td>
<td>102</td>
</tr>
<tr>
<td>9.</td>
<td>Results of measurement of different RPG implementations</td>
<td>108</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Implementation of a complex software system as a set of concerns</td>
<td>5</td>
</tr>
<tr>
<td>2.</td>
<td>Concern identifier prism</td>
<td>6</td>
</tr>
<tr>
<td>3.</td>
<td>Aspect-oriented Programming development steps</td>
<td>10</td>
</tr>
<tr>
<td>4.</td>
<td>Weaving process</td>
<td>15</td>
</tr>
<tr>
<td>5.</td>
<td>AspectJ compilation process</td>
<td>29</td>
</tr>
<tr>
<td>6.</td>
<td>A typical AspectJ’s browser session</td>
<td>30</td>
</tr>
<tr>
<td>7.</td>
<td>Methods of intercepting classes at load-time</td>
<td>32</td>
</tr>
<tr>
<td>8.</td>
<td>JManipulator’s open architecture schema</td>
<td>33</td>
</tr>
<tr>
<td>9.</td>
<td>JManipulator nested classes loading process</td>
<td>34</td>
</tr>
<tr>
<td>10.</td>
<td>JManipulator implementation of BCEL weaver</td>
<td>35</td>
</tr>
<tr>
<td>11.</td>
<td>Coverage tool implemented with JManipulator</td>
<td>37</td>
</tr>
<tr>
<td>12.</td>
<td>A simple aspect with a couple of advices</td>
<td>40</td>
</tr>
<tr>
<td>13.</td>
<td>Execution Model in .NET framework</td>
<td>49</td>
</tr>
<tr>
<td>14.</td>
<td>.NET compilation process</td>
<td>49</td>
</tr>
<tr>
<td>15.</td>
<td>The structure of an assembly</td>
<td>50</td>
</tr>
<tr>
<td>16.</td>
<td>AOP# architecture</td>
<td>51</td>
</tr>
<tr>
<td>17.</td>
<td>An example of an aspect in AOP#</td>
<td>53</td>
</tr>
<tr>
<td>18.</td>
<td>AOP# engine architecture</td>
<td>54</td>
</tr>
<tr>
<td>19.</td>
<td>An example of element dependencies in RMC</td>
<td>61</td>
</tr>
<tr>
<td>20.</td>
<td>The Process Tailoring discipline</td>
<td>63</td>
</tr>
<tr>
<td>21.</td>
<td>An actual decomposition example</td>
<td>76</td>
</tr>
</tbody>
</table>
22. Basic components of an Aspect-Oriented Programming system..........................77
23. An example of a system descriptor providing declarative security ....................78
24. Authentication example of an application created with Java language .................80
25. Authentication example of an application created with AspectJ .......................81
26. Access control example of an application created with Java language ...............82
27. Access control example of an application created with AspectJ .......................83
28. Accountability example of an application created with AspectJ .......................84
29. The class diagram of Personal Management System.........................................86
30. Aspect-oriented implementation of the system...............................................87
31. Sentient Traffic Simulator Class Diagram created with RTJava .........................91
32. Sentient Traffic Simulator aspect-oriented design diagram...............................92
33. The constructor for RealtimeThread..............................................................93
34. AOSim application code handling threads ......................................................93
35. AOSim application code handling memory management .................................94
36. AOSim application code handling synchronization and resource sharing ............95
37. AOSim code handling asynchronous events.....................................................96
38. AOSim code handling asynchronous transfer of control ....................................97
39. AOSim code related to asynchronous thread termination ...............................98
40. AOSim code related to physical memory access ..............................................98
41. Example of the object-oriented RG application implementation .....................105
42. Example of the memoization aspect code.....................................................107
43. Example of the fusion aspect code..............................................................107
44. Example of the memory management aspect code.........................................108
45. Duplicated code in Eclipse plugin ................................................................. 110
46. UML diagram of the solution to exception handling related problems .......... 111
47. Detailed structure of the patterns ................................................................ 112
48. Steps in case 1 when handling an exception .................................................. 113
49. Steps in case 2 when handling an exception .................................................. 114
50. A class OrderDAO ....................................................................................... 116
51. The class diagram for a new code design ..................................................... 117
52. LogArgumentsThrowsAdvice code .............................................................. 118
53. GetProxy method code ................................................................................ 118
54. Exception Handling Aspect interaction ....................................................... 119
55. SerializeArgumentsThrowsAdvice code ...................................................... 120
56. Design of Health Watcher application ......................................................... 122
57. Step 1 implementation code ......................................................................... 126
58. Step 2 implementation code ......................................................................... 128
59. The ThreadPooling aspect ........................................................................... 129
60. The Source Code for the Code Coverage Tool ............................................. 134
61. The structure of classes described by the package signature “mzbase” .............. 136
62. The source code of the unit “driver” and the package signature “program” .......... 137
63. The source code of “base” and “game” units ................................................. 137
64. Connections between packages included in the unit “game” ......................... 138
65. The package signature “mzmagic” ............................................................... 138
66. The source code for the atom unit “opmagic” .............................................. 139
67. The source code for the package signature “mzlocked” and the atom unit “locked” 140
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>68.</td>
<td>The source code for the atom unit “opbase” and the compound unit “game2”........</td>
</tr>
<tr>
<td>69.</td>
<td>Connections established in linking of “game2” and “opmagloc” units...............</td>
</tr>
<tr>
<td>70.</td>
<td>Object-oriented class diagram........................................................................</td>
</tr>
<tr>
<td>71.</td>
<td>Object-oriented code: deposit into account....................................................</td>
</tr>
<tr>
<td>72.</td>
<td>Aspect-oriented class diagram ..........................................................................</td>
</tr>
<tr>
<td>73.</td>
<td>The AspectTransaction aspect class methods....................................................</td>
</tr>
<tr>
<td>74.</td>
<td>Aspect-oriented code: deposit into account....................................................</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

The ultimate goal of software development is to create software systems, usually for a von Neumann architecture of instructions, registers, and memory locations. [7] The modern computer language should help to develop a system that is easy to maintain and makes efficient use of memory. Traditional procedural and object-oriented languages do not handle those tasks very well. Procedural programming can be used to implement the computer system clearly, concisely, and in good alignment with the domain model. [14] The attempt to implement the system that has optimized memory usage often results in creating a tangled code, which is difficult to maintain. Therefore, a new paradigm for software development was required. Some software engineers already formulated certain ideas that could be used when creating a new paradigm for computer programming. For example, in 1972, D.L. Parnas, a Canadian software engineer, who developed the concept of information hiding in modular programming argued that to create systems that are easy to implement, verify, and evolve, one must decompose the system into modules in such a way, that each module hides an aspect of the system that can evolve independently of other aspects; this leads to modules that are loosely coupled and can be implemented, understood, verified, and evolved independently. [30] It was a brilliant idea formulated about 20 years before a paradigm called Aspect-oriented Programming was created by scientists at the Xerox Palo Alta Research Center (PARC). The scientist at the Palo Alta also created the first Aspect-oriented language called AspectJ, as an extension to Java language. The new paradigm has great potentials to create reliable and easy to maintain computer software.
Currently, Aspect-oriented Programming is implemented into several different languages and platforms. Aspect-oriented software development is becoming more popular among software developers in the world. The first International Conference on Aspect-oriented Software Development was held in year 2002 in Enschede, Netherlands and it has become an annual international event.

STATEMENT OF PURPOSE

The purpose of the paper is to 1) present the basic concept behind the Aspect-oriented Programming paradigm 2) describe its features such as aspectual decomposition and crosscutting, modularity and separation of concerns, and weaving 3) explore how the Aspect-oriented Programming paradigm can improve several issues related to application development such as security, real-time concerns, and error and failure handling 4) research Aspect-oriented Programming development in Java using AspectJ, JMangler, AspectWerkz, Jiazzi and in .NET using AOP# as well as IBM implementation of Aspect-oriented programming 5) try to predict the future of this new paradigm as well as available alternatives.

CONTENT

In order to achieve these goals, it is important to understand what Aspect-oriented Programming is and what the advantages of its implementation are. During my research, I completed a review of scientific papers published on Internet as well as books about Aspect-oriented Programming.

The entire essay can be divided into the three main topics. First topic is the theory behind Aspect-oriented Programming that includes basic concept and specific features such as
aspectual decomposition and crosscutting, modularity and separation of concerns, and weaving. Second topic explores how Aspect-oriented Programming can help to improve software development issues such as security, real-time concerns, and error and failure handling. When discussing this topic, I will point out any possible weaknesses of Aspect-oriented Programming. The third topic evaluates implementation of Aspect-oriented Programming in different language such Java using AspectJ, JMangler, Aspect Werkz, and Jiazi and .NET using AOP# as well as IBM implementation of Aspect-oriented Programming. 

The last chapter summarizes all advantages and possible disadvantages of Aspect-oriented Programming drawing a conclusion about its importance for software development. The last question that is answered is to predict the future of Aspect-oriented Programming. The prediction of the Aspect-oriented Programming is also discussed.

**SIGNIFICANCE**

Aspect-oriented Programming is a new paradigm that was designed to complement the Object-Oriented Programming in attempt to fix all related problems. The theory of this new paradigm was initially defined by scientist at the Xerox Palo Alta Research Center. It was expected that this new theory may revolutionize software development. So far it did not happen, but the new paradigm was already implemented in several computer languages such as Java, .NET, and Smalltalk. The most important factor is that the implementation of this new paradigm is not limited to the object-oriented languages.

The most significant idea of this paper is to introduce the reader to the new paradigm in software development. As the latest paradigms in software development are slowly reaching
their limitations, the future development demands new theories to handle more complicated problems and to better simulate complicated models such as domain models.

Another important idea, discussed in this paper, is the possible implementation of Aspect-oriented Programming in computer languages and the advantages of this for software development.

Next chapter will provide an overview of the theory behind Aspect-oriented Programming.
CHAPTER II
THEORY BEHIND ASPECT-ORIENTED PROGRAMMING

BACKGROUND

We can view a complex software system as a combined implementation of multiple concerns. [20] Complex software systems may be composed of many different concerns such as business logic, performance, data persistence, logging, debugging, authentication, security, multithread safety, and error checking. During the development of such a complex system a developer should create a comprehensive and easy to maintain, debug, and expand software system.

Figure 1 shows the implementation of a complex software system as a set of concerns.

Figure 1. Implementation of a complex software system as a set of concerns

Based on multiple requirements, a developer has to identify and separate all concerns related to the developed system. Figure 2 shows this process often described as a concern.
**identifier prism.** The process is using an analogy of the prism separating white light passed through into a spectrum to the process of identifying and separating concerns.

Categories of requirements can be either core module-level or system-level. Core-module requirements often crosscut many modules creating crosscutting concerns in a system.

Figure 2. Concern identifier prism.

How do the current software development and implementation techniques address multiple requirements and crosscutting concerns?

The current implementation techniques implement multiple requirements using one-dimensional methodologies. [20] In other worlds multi-dimensional requirements are mapped into a single-dimensional methodology resulting in code tangling and scattering. Code tangling is happening when a module, in software system, implementing one concern is mixed with modules implementing other concerns. Code scattering happens when the implementation of multiple concerns is spread over all modules in software system. Both code tangling and code scattering create many problems during design, development, and maintenance of software systems.

Those problems include:
Poor mapping between concerns and their implementation that makes software system complex and its logic difficult to understand. Because of that it is easy to introduce bugs in software system.

Poor productivity that affects development process when developers have to pay attention to many concerns instead of concentrating on the core functionality.

Poor reusability of modules in software system that means software system may contain several modules with similar functionality. Those modules cannot be reused because each of them also includes part of a crosscutting concern.

Poor code quality that means some concerns may receive not enough attention creating hidden problems.

Poor maintainability that means any expansion or modification of software system affects many unrelated modules and therefore it is difficult to perform.

Problems related to crosscutting concerns affected many software systems and it created a need to develop a completely new programming theory.

In 1995, a new paradigm called Aspect-Oriented Programming was developed at Xerox PARC Palo Alto Research Center by a team of researchers led by Gregor Kiczales. Two years later in 1997, this new paradigm was presented at the European Conference on Object-Oriented Programming (ECOOP) in Finland.

BASIC CONCEPTS

Aspect-oriented Programming (AOP) was defined by Gregor Kiczales as a programming technique that makes possible to clearly express programs using aspects and including appropriate isolation, composition, and reuse of the aspect code. In 1997, Gregor Kiczales mentioned that the Aspect-oriented Programming research was analogous to that of
Object-Oriented Programming 20 years ago. [14] Aspect-oriented Programming is often referred as Aspect-oriented Software Development (AOSD). The Gregor Kiczales’ definition is not very clear and it does require further explanation and clarification. Another definition, taken from a.osd.net Web site, says that Aspect-oriented software development is a new technology for separation of concerns in software development. The new technology makes possible to modularize crosscutting aspects of a system.

It looks like Aspect-oriented Programming was created to fix problems related to crosscutting concerns in software systems.

The term aspect is something that differentiates Aspect-oriented Programming from Object-Oriented Programming. An aspect is one kind of concern in software development. An aspect can be compared to a class, but an aspect is more abstract and less concrete than a class [28]

Another definition of an aspect says that aspects, in Aspect-oriented Programming, package advice and point cuts into functional units in much the same way that Object-Oriented Programming uses classes to package fields and methods. [28]

The second definition tells more about aspects, but it also introduces new terms related to Aspect-oriented Software Development.

An advice is a piece of code that executes when point cut is reached and the point cut matches a join point. Join points are well-defined places in the structure or execution flow of a program where additional behavior can be attached. The most common elements of a join point model are method calls. [7] A point cut is a place of execution in the source code, meaning that given a certain dynamic join point, a point cut can either match join point or not. Both point cuts and advices form an aspect. The exact meaning of those definitions can
be different when talking about implementation of Aspect-oriented Programming in a specific programming language.

There are two main properties for any Aspect-oriented Programming distinguishing it from other paradigms:

- **Quantification** is the idea that a single or separate statements may affect many places in programming system
- **Obliviousness** means that you cannot tell that the aspect code will execute by examining its content.

Aspect-oriented Programming can be understood as the desire to make quantified statement about the behavior of programs and to have these quantifications hold over programs that have no explicit reference to the possibility of additional behavior. [7] What does it mean to a program’s code? It means that the code, realizing a concern, executes it whenever specific condition arises. We may describe the programming statement in English as “In program Pr, whenever condition Co arises, perform action Ac”. Quantification or quantified statements allow organizing a program in the form most appropriate for coding and maintenance. The weaving mechanism takes care of how and when the primitive directions are to be performed.

Obliviousness is very useful because it permits greater separation of concerns during the system creation process. The crosscutting behavior is intermixed by higher level specification, not by low level programming like in traditional languages. Non-oblivious systems require a program to address some concerns as well as to invoke the code for those concerns. It means that a developer of non-oblivious systems has to remember to include, in the code, all calls for designated routines.
Regardless of its implementation Aspect-oriented Programming comprises of three developments steps:

- Aspectual decomposition that include decomposition of requirements and identification of crosscutting and common concerns. During this step the module-level concerns are separated from crosscutting-level concerns.
- Concern implementation that includes implementation of each identified concern separately.
- Aspectual re-composition that includes creating modularization units called aspects based on re-composition rules defined by an aspect weave (also called integrator). This step is also called weaving or integrating. During that step the final software system is created.

Figure 3 shows Aspect-oriented development steps.

Figure 3. Aspect-oriented development steps

Aspect-oriented Programming implementation is not limited to aspect-oriented languages. It can be implemented not only with any object-oriented language, such as C++ and Smalltalk, but also with general purposes languages such as C.
Let’s talk about important Aspect-oriented terms or features such as crosscutting concerns, modularity, and weaving.

**ASPECTUAL DECOMPOSITION AND CROSSCUTTING CONCERNS**

The aspectual decomposition is the first step in the aspect-oriented development. This step is very important because based on multiple requirements a designer has to identify all areas of interest or focus in a system. Those areas of interest or focus are called concerns. Concerns can be categorized into module-level and system-level concerns. System level concerns are usually spread over many modules and those concerns require special attention. The term aspect in decomposition is referred to crosscutting concerns. It is very important to create relations among the aspects of concern precise because it enables:

- The proper work of the weaving process
- Reasoning about the code
- Debugging the code.

The key to make Aspect-oriented Programming work is to design basic functionality and use an aspect description language that supports an aspectual decomposition as well as weaving.

Identified concerns are captured using appropriate special purpose languages (Chapter IV talks about aspect-oriented development using different languages).

The Appendix A presents an example of a prototype of Aspect–Oriented Programming including aspectual decomposition and short description of weaving process.

The example illustrates how Aspect-oriented Programming requires less line of code than similar Object-Oriented Program. The reason for that are crosscutting concerns. Object-Oriented Program such as Java requires duplicated code all over the program.
MODULARITY AND SEPARATION OF CROSSCUTTING CONCERNS

Modularity is a characteristic of a system that has been divided into smaller subsystems which interact with each other. The concept of modularity was introduced by D.L. Parnas, a Canadian software engineer, who wrote that modularity enables better comprehensibility, changeability, and independent development. In Aspect-oriented Programming, the concept of modularity is used when crosscutting concerns are separated. The separation of crosscutting concerns is a big benefit of Aspect-oriented Programming comparing to limitation called the **Tyranny of the Dominant Decomposition** faced by traditional languages.

The Tyranny of the Dominant Decomposition means that program can be modularized (divided) into smaller subsystems in only one way at a time. It creates the situation when many kinds of concerns are not aligned with that modularization. As a result we end up with concerns scattered across many modules and tangled with one another.

In Aspect-oriented Programming, a developer defines aspects and each aspect defines one or more point cuts. Point cuts are point where the crosscutting concern hooks into the software system. The implementation of any functionality in software systems is done by advices applied to point cuts. The crosscutting concerns can be defined in a modular way at a central point that simplifies any changes to specification of the system.

There are important rules and principles to ensure modularity such as:

- Direct Mapping rule
- Linguistic Modular Units principle. [10]

The **Direct Mapping rule** means that implemented modular structure reflects the modular structure design when the problem domain is modeled. Aspect-oriented Programming
follows the Direct Rule and as a result the design and the final structure of the software system reflect the components of the software system functionality. Aspect-oriented developer does not have to split crosscutting concerns across the entire software system, but, instead, may embed the code in one module called aspect and weave it into other modules using point cuts and advices.

The **Linguistic Modular Units principle** depends upon modules in software system correspond to syntactic language and how they can be compiled. Software system fulfills this principle when its modules correspond to syntactic language units in the programming language and can be compiled on their own. Aspect-oriented Programming fulfills the Linguistic Modular Units principle by allowing treating even crosscutting concerns as single modules and compiling them on their own.

**WEAVING OR INTEGRATION**

During the weaving process the aspect weaver accepts the component as input, and emits a C program as output. [14] This definition is right, but it is not implementation neutral. It also illustrates the fact that is difficult to talk about theory behind Aspect-oriented Programming without showing examples. Usually those examples are implementations of Aspect-oriented Programming in specific languages such as Java. More general definition of weaving says that weaving expresses how the system intertwines the execution of the base code and aspect. Key elements include the actual weaving mechanism (for example, compile-time weaving, altering the interpretation process, or meta- or reflective mechanism). [7] In other words, you may say that weaving coordinates aspects and components. A component is usually a unit of software system decomposition such as image filters, bank accounts, and GUI widgets. An aspect is a property that affects the performance or semantics of the
components in systemic ways. [29] Both components and aspects should be encapsulated in a
generalized procedure such as object, method, procedure, and API. We may also say that an
aspect provides functionality that crosscuts components in software systems.

Weaving process depends on the implementation elements such as:

- A component language in which a component program implementing components is
  written such as Java
- An aspect language in which aspects are implemented such as AspectJ
- An aspect weaver that is basically a meta-programming tool or an engine for the
  weaving process. An aspect weaver can be also called a compiler to make the term
  relevant to Object-Oriented Programming.

The weaving process can be done at runtime which is called RT weaving or at compile
time which is called CT weaving. The CT weaving generates executable code which is
executed at runtime. The RT weaving intercepts methods in classes and weaves the aspect
code through them during runtime. [29]

Before weaving is done the weaving rules need to be specified to know how to compose
separated and independently implemented concerns to form the final system.

During weaving the program should be able to:

- Find out what aspects should be weaved into which methods
- Weave an aspect into the code within a method at the right time, not just before and
  after it runs.

Figure 4 shows the weaving process.
The three development steps in Aspect-oriented Programming, described in this chapter, are implementation neutral. The specific implementation with specific languages may be slightly different because each computer language has its own specification, syntax, and limitations.

The next chapter will begin with overview of Aspect-oriented Programming improvements to specific areas of software such as security, real-time concerns, and error and failure handling.
CHAPTER III

ASPECT-ORIENTED PROGRAMMING IMPROVEMENTS

Security, real-time concerns, and error and failure handling are very important concerns to any software development. Let’s analyze if Aspect-oriented Programming makes any improvements related to application development in relation to those concerns.

SECURITY

Security is becoming a very important issue for many software applications, especially for an application handling very sensitive and important information. Web enabled applications, available either through Intranet or Extranet, are very popular. For example many banks are offering Web banking services for their customers. Using Web banking customers have access to their accounts with abilities to make financial transactions such as:

- Paying bills to previously defined payees
- Transferring money between accounts
- Creating new accounts
- Submitting requests for new cheques
- Managing investment portfolios.

Banking is just one example of areas where security of software application is extremely important. Other areas are related to government, military, intelligence services, and private industry. The list of areas is much longer, but my intention is to stress out the importance of software security.

Software systems contain many security related concerns such as:

- Identification and authentication as a process of verifying the true identity of a user
- Access control as a process of allowing only authorized users to access the software system and denying access to any intruders or hackers
- Accountability as a process of ensuring that any action done by an entity (a user or a system) may be traced to responsible entity
- Audit as a process of reviewing and examination of system transactions and records to test if the system is compliant with established policy, system controls, and operational procedures. An audit may result in changing specific recommendation any existing policy, system controls, and operational procedures. [32]

Unfortunately, the security concerns listed above are also crosscutting concerns spread throughout the software system. Separating security concerns was a problem that many computer scientists were trying to solve for years with limited success. The success was related to modularizing the implementation of security mechanism, but the problem where and when to call specific security mechanism in software system was not addressed yet. To make the problem even more complicated, security concerns may be related not only to the variety of specific places where security mechanisms are called, but also to the context of calls. [7] Sometimes the information required by security mechanisms is coming from the external source to software systems. An example of such a case is communication encryption within software system where the keys required to decrypt or encrypt the communicated message are linked to a particular user. The keys used for decryption and encryption depend on many factors such as a communication channel, host information, and the encryption algorithm. Another security problem is to protect data, which may also be spread all over software system. It looks like the separation of security concerns is a problem that has not
been fully resolved by the most popular programming paradigms such as Object-Oriented Programming.

Let’s present couple of examples on how Aspect-oriented Programming is handling security concerns.

The first example, presented in the Appendix B, compares Aspect-oriented and Container Managed security.

After comparing Aspect-oriented and Container Managed security, it looks that Aspect-oriented security implemented with AspectJ (aspect-oriented extension of Java) provides better separation of security concerns from the rest of an application. This advantage allows better maintenance of security code. Any changes to the security mechanism can be done only in aspects. Container managed security offers less flexibility. The functionality provided by a container is limited. Programmatic security allows exceeding limitations of a container, but the implementation becomes more complex and code is difficult to maintain.

The second example, presented in the Appendix C, analyzes security implemented for a Personal Information Management (PIM) system. The system developed in AspectJ (aspect-oriented extension of Java) is compared to object-oriented implementation.

After comparing both implementations, it looks like object-oriented implementation is more complex and does not handle crosscutting concerns very well. Aspect-oriented implementation separates crosscutting concerns very well and the successful separation offers several advantages over object-oriented implementation. The advantages include easier maintenance of the security code, ability to easy add new security concerns, and faster development of the system by developers.
REAL-TIME CONCERNS

Real-time concerns like memory management and thread scheduling are usually crosscutting concerns. One of the main features of Aspect-oriented Programming is separation of concerns. Let’s analyze if Aspect-oriented Programming provides any improvements when handling real-time concerns. The best way to explore such a problem is to compare aspect-oriented and object-oriented implementations of real-time concerns.

The first example, presented in Appendix D, compares a Real-time Java (RTJava) and AspectJ (aspect-oriented extension of Java) applications. In this comparison several real-time concerns are compared such as:

- Thread scheduling and dispatching
- Synchronization and resource sharing
- Asynchronous thread termination
- Memory management
- Physical memory access
- Asynchronous event handling
- Asynchronous transfer of control.

The comparison results were mixed, aspect-oriented implementation improved the application comparing to object-oriented implementation in respect to some real-time concerns, but provided no benefits in respect to other concerns.

Let’s analyze how aspect-oriented implementation affected application in respect of each real-time concern.

**Thread scheduling and dispatching**
Both individual and average results show that aspect-oriented implementation has provided a good improvement comparing to object-oriented implementation.

**Synchronization and resource sharing**

Both individual and average results show that aspect-oriented implementation has provided no benefits comparing to object-oriented implementation. The only measure showing some improvement has been ‘Lack of Cohesion of Methods’. There has been also some improvement in reusability area.

**Asynchronous thread termination**

Both individual and average results show that aspect-oriented implementation provided no benefits comparing to object-oriented implementation. The only measure showing some improvement has been ‘Coupling between Objects’.

**Memory management**

The memory management has been the concern in which aspect-oriented implementation has had the worst results providing no benefits at all with exception for Coupling between Objects measure. ‘Response For a Class’ measure has been the worst comparing to other concerns.

**Physical memory access**

The physical memory access has had mixed results. Aspect-oriented implementation has provided improvements in areas such as understandability, maintainability, reusability, and testability, but C&K metrics have shown improvement only for one measure: ‘Coupling between objects’. The other five measures have shown no improvements.

**Asynchronous event handling**
The asynchronous event handling has had mixed results. Aspect-oriented implementation has provided improvements in areas such as understandability, maintainability, reusability, and testability, but C&K metrics have shown improvement only for one measure: ‘Coupling between objects’. The other five measures have shown no improvements.

**Asynchronous transfer of control**

Individual results show that aspect-oriented implementation provided no benefit comparing to object-oriented implementation. The C&K metrics have shown some improvements for ‘Coupling between Objects’ measure and for other measures has shown no improvements. Average results for real-time areas have shown no improvements in reusability, understandability, maintainability, and testability areas.

The first example compared aspect-oriented and RTJava implementations. It was difficult to show benefits of aspect-oriented implementation since RTJava is an extension of Java handling real-time concerns much better than pure Java implementation.

The second example of comparison is presented in Appendix E. It compares object-oriented and aspect-oriented implementations of RG application developed at Xerox PARC for image processing.

The second example illustrates that aspect-oriented implementation does not improve the performance of the application, but it has other advantages such as code simplification.

Both examples have shown that aspect-oriented implementation performance has some weaknesses that should be improved such as implementation memory management, synchronization and resource sharing, and thread termination. The comparison in the first example has been done using the metrics designed to evaluate object-oriented programs and the comparison using different metrics may result in fair assessment of programs being
compared. Generally, both comparisons illustrate that aspect-oriented implementation simplifies application’s code without sacrificing performance.

**ERROR AND FAILURE HANDLING**

Error and failure handling is another example of a crosscutting concern. In majority of software applications, exception handling code is not separated from the rest of application code. Lack of separation prevents reusability of exception handling modules and makes exception handling code difficult to maintain. Also, exception handling code is often duplicated across software system. Those problems exist in many popular programming languages such as Java, C++, and C#.

One of the most important ideas behind aspect-oriented paradigm is separation of crosscutting concerns and it looks like applying this idea to exception handling is a very difficult task.

Let’s present couple of examples on how aspect-oriented implementation is handling error and failure handling.

The first example, presented in Appendix F, shows how AspectJ (aspect-oriented extension of Java) handles error and failure handling. The first example illustrates that aspect-oriented implementation has a lot benefits, but also some weaknesses.

Below are the following benefits of aspect-oriented implementation of error handling:

- Separation of error handling code
- Reduction of duplicated code
- Wide range of exception successfully handled with exception for checked exceptions
- Easy maintenance of exception code.
The weaknesses of aspect-oriented implementation of error handling are:

- Difficult to understand the interaction between exception code and base code
- Necessity for refactoring of the base code (in some specific cases)
- Difficulty to handle checked exception, and increase in exception code size.

As per checked exception, AspectJ provides a fix to this problem called exception softening that suppresses the checks performed by the Java compiler in certain join points.

The second example, presented in Appendix G, shows how C# and Spring.Net aspect-oriented deals with error handling.

This example demonstrates that aspect-oriented C# and Spring.Net provides the following improvements comparing to object-oriented application:

- Eliminates code duplication in exception handling code
- Separates exception handling code from the rest of application
- Reduces the size of exception handling code
- Provides detailed debugging information
- Increases the agility of the design
- Improves code maintainability.

Based on presented examples, we can see that both aspect-oriented implementations are using a pattern called Error Handling Aspect. Also, we may describe an exception handling mechanism in aspect-oriented programs. It includes four main concepts such as exceptions, exception signalers, exception handlers, and the exception model. An exception is usually raised by an element such a method or an advice when an unusual condition is detected. If the exception raised cannot be handled inside the method, it is signaled to the method’s caller. The exception signaler is the element that detects unusual state or condition and
raises the exception. An **exception handling** is the code invoked in response to a raised exception. The function of exception handlers is to restore the software to the normal state and log the exception or to stop the software function in safe way. The **exception model** defines how exception signalers and handlers are connected to each other. [4]

The third example, presented in Appendix H, analyzes AspectJ (aspect-oriented extension of Java) implementation in respect of possible problems related to the use of aspects.

Analysis of AspectJ application is done in two categories in which aspects are acting either as signalers or exception handlers. Based on analysis, several new problems were found. New problems are related to aspect-oriented exception handling code as well as to interaction among its elements. New problems related to exception handling code in AspectJ application may suggest weaknesses in AspectJ language, but there are solutions to fix all problems. See Appendix H for more details related to problems and solutions.

First two examples show several benefits to implementation of exception handling concern in an aspect-oriented language. The most important advantages are: clear separation of concern from the rest of code, reduction of duplicated code, and easy maintenance of exception code. However, one weakness related to understanding the interaction between exception code and base code is worth investigating.

The next chapter will describe and evaluate implementations of Aspect-oriented Programming in different languages such as Java using AspectJ, JMangler, Aspect Werkz, and Jiazi and .Net using AOP# as well as IBM implementation of Aspect-oriented programming.
CHAPTER IV

ASPECT-ORIENTED PROGRAMMING IMPLEMENTATIONS

Aspect-oriented paradigm is still relatively young idea not as popular as Object-oriented Programming. In 1997, Gregor Kiczales, one of scientists who created Aspect-oriented paradigm said that Aspect-oriented Programming research was analogous to that of Object-oriented Programming 20 years ago. Since 1997, several new implementations of aspect-oriented paradigm have been introduced. The first implementation of aspect-oriented paradigm is called AspectJ.

DEVELOPMENT IN JAVA USING ASPECTJ

AspectJ is more general-purpose language than other aspect-oriented language and it can be used for variety of applications. [11] It is also aspect-oriented extension of Java language that makes any valid Java program also a valid AspectJ program. AspectJ is a very attractive tool for any developer because it is freely available under an open-source license. The fact that AspectJ extends Java language makes it easier to learn by Java developers. Features such as an aspect weaver that is in the form of a compiler, an aspect-aware debugger, documentation generator, and aspect browser that visualizes on how an aspect crosscuts a system’s parts are additional benefits of AspectJ. [21]

AspectJ is using a compiler to implement weaving rules and this process is called crosscutting because the weaving rules cut across multiple modules to modularize crosscutting concerns. [19] There are two types of crosscutting such as:

- Static crosscutting that weaves of modifications which do not modify the execution behavior of the system into static structures such as classes, interfaces, aspects. An
example of a modification used in static crosscutting is adding a new data and
methods to classes and interfaces to define class states and behaviors. Another
example of a modification is defining compile-time warning and errors used by
multiple modules in an application. Generally, static crosscutting supports dynamic
crosscutting.

- Dynamic crosscutting that weaves of new behaviors into the execution of a program.
Dynamic crosscutting may expand or even replace the core program execution flow
by modifying the system behavior. Example of dynamic crosscutting is when a
certain action needs to be executed before the execution of certain methods. In order
to achieve it, the weaving points and the action to take when reaching those points
need to be specified in separated module.

Weaving rules in AspectJ are defined programmatically using the following constructs:

- Join point that is a point in the execution of a program such as a method or an
assignment to a member of an object. Join point is a place where crosscutting actions
are woven in. The join point is fundamental concept of AspectJ identifying an
execution point in a system. The categories of join points available in AspectJ are
method call and execution, constructor call and execution, read/write access to a field,
exception handler execution, and object and class initialization execution. [21] The
example below, shows join points, in the Account class, such as the method credit()
and the access to the _balance instance member.

```java
public class Account {
    ...
    void credit (float amount) {
        _balance += amount;
    }
}
```
• Point cut that is a program construct selecting join points as well as collecting context at those join points. A point cut may select a call to a method and capture the method’s context. An example below shows a point cut that captures the execution of the credit() method in the Account class (shown when talking about join point)

```
execution (void Account.credit(float))
```

In other worlds, we may say that point cuts are specifying weaving rules and join point are situations satisfying those rules.

• Advice that is the code executed at a join point. Advises are selected by point cuts. It is possible to execute advices before, after, and around a join point. The advice code looks similar to a method. The around advice may modify, replace or bypass the execution of the code located at a join point. The before advice may log a message before the execution of the code located at a join point. [19] An example below shows an advice that prints a message before execution of the credit() method in the Account class.

```
before() : execution(void Account.credit(float)) { 
    System.out.println("About to perform credit operation"); 
}
```

Dynamic crosscutting rules are created by point cuts and an advice. Point cuts identify necessary join points and an advice is the code executed at the join points.

• Introduction that is a static crosscutting instruction that introduces changes to the classes, interfaces, and aspects of the application. [19] An introduction can be used to make static changes to applications’ modules such as adding methods or field to a class. An example below shows an introduction that declares the Account class to implement the BankingEntity interface.

```
declare parents: Account implements BankingEntity;
```
• Compile-time declaration that is a static instruction used to add compile-time warnings and errors when a specific condition is detected. An example below shows a declaration that makes the compiler to issue a warning when the save() method in the Persistence class is called. The point cut call() is used to capture calls to the save() method.

    declare warning : call(void Persistence.save(Object))
    : “Consider using Persistence.saveOptimized()”;

• Aspect that is the code defining weaving rules for dynamic and static crosscutting. The idea of an aspect in aspect-oriented program is similar to a class in object-oriented program. An aspect can be described as a collection of point cuts, advice, introductions, and declarations as well as data, methods, and nested class members.

An example below shows the ExampleAspect aspect code.

    public aspect ExampleAspect  { 
    before()  : execution(void Account.credit(float))  { }
    declare parents: Account implements BankingEntity;
    declare warning : call(void Persistence.save(Object))
    : “Consider using Persistence.saveOptimized()”;

The last example shows how to design a crosscutting behavior. It is done in the following order:

• Join points, at which the behavior is expanded or modified, are identified

• The new behavior is designed

• An aspect, implemented as a module to include the overall implementation, is created

• Point cut, inside the created aspect, is created to capture join points

• An advice, for each point cut, is created to define what code is executed when join point is reached.
AspectJ allows limited form of reflection to examine information at any point cut’s execution point. [21] The limited reflective access is very important in regards to logging and debugging aspects.

AspectJ compiler, called ajc, combines Java and aspect source files as well as Java Archive (JAR) files to create woven class files or Java Archive (JAR) files as output. Java Archive (JAR) file is a Java archive file that aggregate many files into one file and is used to distribute Java applications and libraries. The weaving process in AspectJ is done during the compilation. During the compilation the base source code and the code defining weaving rules are processed to create an output class file or a Java Archive (JAR) file. The future release of AspectJ may allow runtime weaving taking place when classes are loaded into the virtual machine. Figure 5 shows AspectJ compilation process.

Figure 5. AspectJ compilation process

Ajbrowser is another important AspectJ tool that displays how weaving rules affect different parts of a program. The browser helps developers to:

- Explore relationships between various elements in a program
- Understand the effect of crosscutting constructs on different modules
- Debug a program
- Verify constructs crosscutting the desired parts of a program.
Figure 6 shows a typical AspectJ’s browser session.

Figure 6. A typical AspectJ’s browser session

AspectJ is integrated with many development applications such as Eclipse, Forte, NetBeans, JBuilder, and EmacsJDEE. Those development tools offer similar functionalities as offered by AspectJ browser.

An example of AspectJ code is presented in Appendix I. It illustrates how AspectJ may help implement resource-pool management.

AspectJ expands Java language abilities by adding aspect-oriented features such as separation of crosscutting concerns, reusability of code, aspect-aware debugger, and documentation generator as well as easy maintenance of code. There is a possibility that the aspect code is hard to follow, especially when developing complex system or adding new functionality to the existing aspect. The new language retains all the benefits of Java language. AspectJ is easy to learn by any developer familiar with Java language. AspectJ is constantly improved by creating newer and better versions.
One of the most important processes introduced by Aspect-oriented Programming paradigm is weaving the aspect into the code of all classes that required it. There are three possible weaving scenarios such as:

- Static weaving that is performed at compile-time before a program starts
- Load-time weaving that is performed when a program is loaded
- Dynamic weaving that is performed after a class was loaded. [17]

There are several challenges for aspect weaving related to weaving scenarios, context, and variants. As per weaving scenarios, both static and dynamic weaving has weaknesses. Static weaving weakness is that the process must rely on a developer to identify all classes that belong to a program. Dynamic weaving weakness is that it cannot enforce an aspect application to dynamically loaded classes. Weaving context might be applied in fully controlled or in partially controlled contexts depending on weaving entities creation, storing, and formats. The problem with weaving variants is that the number of variants depends on its context of use. For example, an application may or may not need to integrate a logging aspect. To make the situation more complicated, it could be prohibitive to generate all variants statically and in advance. [17]

A method that can be used in response to all challenges related to aspect weaving might be JMangler framework. JMangler is a freely available framework for generic interception and transformation of Java programs at load-time. The framework was created by a group of German scientists: Michael Austermann, Pascal Constanza, Gunter Kniesel, and Helge Koch from University of Bonn. Since the year 2001, three versions of JMangler have been created.
The main problem for JMangler designer is to ensure the interception of all application class files without compromising portability. It is achieved by modifying the final method defineClass() in the class java.lang.ClassLoader, changing behavior of its subclasses, and making JMangler to be activated while specific class is loaded. The interception used in JMangler is independent on a custom Java Virtual Machine (JVM) and on a platform-specific DLL. It allows transformation of all classes with exception for system classes. Figure 7 shows four methods of intercepting classes at load-time.

Figure 7. Methods of intercepting classes at load-time

Figure 7 illustrates that other interception methods such as Binary Component Adaptation (BCA) or DLL-based load-time adaptation are dependent on either custom Java Virtual Machine or a platform-specific DLL.

The third version of JMangler offers advantages of an open architecture such as

- Implementation of the generic interception mechanism
- Extending Java’s linear class loading scheme by nested class loading
- Implementation of the weaver based on Byte Code Engineering Library (BCEL) offering the real weaving functionality without any constraints.

The Byte Code Engineering Library (BCEL) is intended to give users a convenient possibility to analyze, create, and manipulate (binary) Java class files (files with extension .class). Figure 8 shows JMangler’s open architecture schema.

Figure 8. JMangler’s open architecture schema

![JMangler's open architecture schema](image)

Figure 9 shows how JMangler extends Java’s linear mechanism for loading classes by providing nested class loading. Classes loaded and transformed during processing of another class are stored and only passed to the Java Virtual Machine after completion of the suspended class loading process. [17] The advantage of storing transformed classes is that they can be used for later non-local analyzes without repeating transformation process.
Figure 9. JMangler nested classes loading process

JMangler has one problem related to unanticipated composition of independently developed transformers. The latest version of JMangler does not have any way of determine in which order those transformers should be combined. The future version of JMangler may solve the problem. The partial solution provided by JMangler is to support only interface transformations that preserve binary compatibility and to iterate interface transformations until a fixed point is reached. Figure 10 shows how JMangler handles unanticipated composition of independently developed transformers using BCEL weaver.
Figure 10. JMangler implementation of BCEL weaver

Summing up, there are several reasons why JMangler is a good framework to be used in aspect weaving process such as:

- It requires no source code that means it enables transformation of Java programs without access to source code using the information included in the class files.
- It enables load-time adaption that means it can adapt classes in static and load-time weaving scenarios allowing processing all classes that are executed during a program run including classes created and loaded dynamically as well as classes from remote hosts.
- It is class loader independent that means it supports transformations for classes loaded by arbitrary class loaders with exception for bootstrap class loader. Therefore, it can transform any application classes with exception for system classes. It can be used in applications and environments deploying their own class loaders.
- It is powerful that means it allows a wide range of code and interface transformations.
• It enforces binary compatibility that means it is compatible with transformations compatible with the Java Binary Compatibility Specification

• It supports multiple transformers that can be applied simultaneously to all classes of a program as well as user defined transformations

• It supports composition of independently developed transformers that means independently developed and unaware of each other transformers can be combined

• It supports in place modifications that means a class can be modified in place without affecting performance of transformations

• It supports caching modifications that means transformations can be applied statically or the output of the load-time transformation process can be stored for later use

• It is easy to configure that means transformers can be combined by editing a text file

• It is pure Java that means it is written in Java language and can be used on any platform supporting JDK1.3, 1.4 or later

• It is freely available that means it is available under the terms of the GNU General Public License.

Figure 11 shows how a coverage tool can be implemented using JMangler. Code coverage is a measure for the percentage of a program exercised (“covered”) by the tests run on that program. The coverage tool reports the percentage of statements in a program that have been executing during testing as well as indicates the places in a program that have not been covered.
An example of coverage code is presented in Appendix J. It illustrates how code coverage can be implemented as a code transformer using JMangler. The example presented in Appendix J illustrates that JMangler is a powerful tool, but its proper use requires an expert knowledge.

DEVELOPMENT IN JAVA USING ASPECT WERKZ

Aspect Werkz is another Aspect-oriented Programming framework for Java. Unlike other implementations of Aspect-oriented Programming, Aspect Werkz uses Java language to define aspects. The main idea when creating Aspect Werkz framework was to design a definition model that is simple and intuitive, but at the same time expressive and powerful.

[1] The framework supports dynamic and static weaving at load time. It weaves classes at bytecode level using the bytecode modification library (BCEL).

Aspect Werkz is using similar constructs as AspectJ such as:

- Join point that is a well-defined point in the execution program flow. Aspect Werkz uses dynamic join points represented by the JointPoint class and used to retrieve Runtime Type Information (RTI) about the class, method, and field at the current join point.
point. As part of the framework, a construct JointPointController is added to allow users to control the execution flow of the advices. The control is enforced by custom defined constraints and rules and those rules can be modified at runtime. JoinPointController is important when handling inter-aspect compatibility and dependency as well as aspect redundancy. [1]

- **Point cut** that is a mechanism to aggregate sets of join points. It led to the implementation of a pattern language which consists of a combination of a class and a field. Aspect Werkz supports point cut composition that means creating a point cut using an expression of other point cuts such as logical operators. The example below shows how point cuts could be built up.

```java
(pc1 || pc2 ) && pc3
!pc1 && pc2
```

The logical operators supported are ‘&&’, ‘||’, and ‘!’.

- **Advice** that is a piece of code that is executed when a point cut is reached. Aspect Werkz supports three types of advices such as before, after, and around advice. Before advice is executed as the join point is reached, just before the join point is executed. “After” advice is executed after the join point is executed. “Around” advice is executed as the join point is reached with ability to control whether the join point is executed. An advice is implemented by a method taking as an argument a JoinPoint class instance and returning a java.lang.Object for around advice and void for before and after advices. Aspect Werkz allows passing parameters, accessible at runtime, to advices making advices reusable. For example, the same advice can be reused in different contexts with different configurations. The example below shows an around advice.
public Object execute(JoinPoint joinPoint) throws Throwable {
    // do some things before the invocation
    Object result = joinPoint.proceed();
    // do some things after the invocation
    return result;
}

- Introduction that is a construct used to extend a class with a new interface, superclass, methods or fields. Aspect Werkz implements introductions by using the concept of **Mixin** that is a smart way to make classes implement other interfaces and add implementation of these interfaces’ methods. Mixin may be also called a multiple inheritance. The example below shows a simple Mixin used by the client

    // Mixin interface
    public interface Hello {
        String sayHello();
    }

    // Mixin implementation
    public class HelloImpl implements Hello {
        String sayHello() {
            return "Hello!";
        }
    }

    ... // if the Mixin Hello has been applied to class Foo
    String greeting = ((Hello)instanceOfFoo).sayHello();

- Aspect that is a unit of modularity for crosscutting concerns composed of point cuts, advices, and introductions. Aspect Werkz supports abstract aspects. Figure 12 shows an example of a simple aspect with a couple of advices.
Figure 12. A simple aspect with a couple of advices

```java
/**
 * @Aspect perInstance
 */
public class MyAspect extends Aspect {
    /* ==== Pointcuts ==== */
    /**
     * @Aspect perInstance
     */
    public class MyAspect extends Aspect {
    /* ==== Pointcuts ==== */
    /**
     * @Call * foo.bar.*{..}
     */
    Pointcut methodsToLog;
    /**
     * @Execution * foo.baz.*{..}
     */
    Pointcut methodsToCache;
    /* ==== Advices ==== */
    /**
     * @Around methodsToCache
     */
    public Object cacheMethod(JoinPoint joinPoint) throws Throwable {
        // check the cache for cached result, if found return it
        final Object result = joinPoint.proceed();
        // put result in cache
        return result;
    }
    /**
     * @Before methodsToLog
     */
    public void logMethod(JoinPoint joinPoint) throws Throwable {
        // log method entry
    }
    /**
     * @After returning methodsToLog
     */
    public void logMethod(JoinPoint joinPoint) throws Throwable {
        // log method entry after returning normally
    }
    /**
     * @After throws(java.lang.Exception) methodsToLog
     */
    public void logMethod(JoinPoint joinPoint) throws Throwable {
        // log method exit after throwing exception
    }
}

Aspect Werkz implements a dynamic aspect model by using loose coupling and
delation. The constructs such as aspects, advices, and introductions are registered in the
system and referenced by handles. They also live in different virtual memory spaces with
different life-cycle. In such design, it is easy to add, remove, and restructure aspects, advices, and introductions at runtime without reloading or re-weaving the target classes.

Let’s describe the scenario when an advice is added to a join point. A JoinPoint field is weaved into the class, instead of the advice itself. It defines the specific join point and a method call to the method proceed() in the JoinPoint class (at the join point). At the time when the target class is instantiated, the JoinPoint instance is registered in the system and receives the handles to the advices bound to the join point it defines. When the join point is reached and the method proceed() is invoked, the advices are also invoked. [1]

The presented example describes layer of indirection that has advantages related to structural changes to aspects, advices, and introductions at runtime, but also has a small performance penalty related to the overhead during the initialization phase. Aspect Werkz supports the following changes done at runtime:

- Swapping the implementation of a Mixin
- Removing advices from join points identified by a point cut
- Loading a completely new advice or aspect and adding it to the system
- Adding an advice to a certain point cut.

The dynamic aspect model in Aspect Werkz is not perfect having some weaknesses such as lack of support for redefining the point cuts at runtime and model’s limitation to do dynamic redefinitions at join points which have already been waved.

As per deployment, Aspect Werkz supports four different deployment models:

- perJVM - one model instance per Java Virtual Machine (JVM)
- perClass – one model instance per class
- perInstance – one model instance per class instance
- perThread – one model instance per thread.

The different deployment models have different life. For example, an introduction with ‘perThread’ deployment model lives and dies with the thread and the same introduction with ‘perJVM’ deployment model acts like a singleton.

Aspect Werkz supports both dynamic weaving and static weaving. As per dynamic weaving, there are five different types of load time weaving separated into two categories: HotSwap and bootclasspath.

The HotSwap is based on the Java Platform Debugger Architecture (JPDA) architecture. The main idea of the HotSwap is to run two Java Virtual Machines. The first Java Virtual Machine runs an application and launches the target application in the second Java Virtual Machine. The second Java Virtual Machine allows HotSwap. It means that the classes in the target application can be redefined from the application running in the first Java Virtual Machine. There are three ways of class redefinition using HotSwap such as:

- Regular HotSwap where the first Java Virtual Machine sets AspectWerkz in the second Java Virtual Machine before the main class and applications dependencies are loaded and connects to the stdout, stderr, and stdin stream of the second Java Virtual Machine to make them start as usual through the first Java Virtual Machine

- Native HotSwap where a native C Java Virtual Machine extension, running in the target application Java Virtual Machine, handles the replacement of the class loader by the enhanced one at Java Virtual Machine initialization

- Remote HotSwap where the application Java Virtual Machine is started in suspended mode and the replacement of the class loader is done by a separate manual process.
The bootclasspath is based on the idea that the bootclasspath can be modified using the Xbootclasspath/p option. There are two ways of class redefinition using bootclasspath such as:

- Transparent bootclasspath where an enhanced class loader, created transparently at the startup, is put in the target application Java Virtual Machine bootclasspath using another Java Virtual Machine

- Prepared bootclasspath where an enhanced class loader is build and zipped into a Java Archive (JAR) file. After the enhanced loader is created, the application’s Java Virtual Machine is started with options to use the previously created enhanced class loader.

Aspect Werkz implements static weaving using a post-processor and adding an extra compilation step to the build process. Static weaving is used by users that do not have full control of the startup process or accept an extra compilation process added when transforming the classes at runtime.

When Aspect Werkz was designed it should support the definition format that felt intuitive for users of the framework. The first version of Aspect Werkz supported XML definition model. The XML model has many advantages, but also some limitations such as separation of metadata (definition) and implementation and separation the implementation parts of the aspect (advices and introductions) into different classes. Because of the limitations of the XML definition model, the new definition model was created. The new definition model uses the concept of runtime attributes such as classes, methods, and fields with meta-data in bytecode. The runtime attribute definition model support the following constructs:
• Aspects that are regular Java classes which can be abstract and inheritance treated like regular Java class inheritance. This approach allows easy creation of reusable aspect components and libraries. The aspect class must extend the Aspect class.

• Advices that are implemented as regular member methods in the aspect class. The model supports five advices such as:
  o Around advices that are invoked “around” the join point
  o Before advices that are invoked before the join point
  o After advices that are invoked after the join point
  o After returning advices that are invoked after a method invocation that returns normally
  o After throws advices that are invoked after a method invocation that throws an exception of a specific type.

• Introductions are implemented as regular member methods in the aspect class. Mixins are defined by using an inner class of the aspect.

• Point cuts are defined by declaring a member variable in the aspect class with the type Pointcut. The are five types of supported point cut attributes such as:
  o Execution point cut that picks out join point defining method execution
  o Call point cut that picks out join point defining method call
  o Set point cut that picks out join point defining field modification
  o Get point cut that picks out join point defining field access
  o CFlow point cut that picks out join point defining a control flow.

Aspect Werkz offers power and simplicity as well as easy integration of Aspect-oriented Programming in both new and existing projects. The fact that the framework uses Java
language makes it easy to learn by Java developers, but it also means that Aspect Werkz inherited all limitations of Java language such as complex to implement runtime weaving. Aspect Werkz aspect model has a weakness; it is not able to define new point cuts at runtime. Aspect Werkz is free software licensed under GNU General Public license and this is an important fact for any developer.

DEVELOPMENT IN JAVA USING JIAZZI

Jiazzi is an enhancement of Java that adds support for encapsulated code modules known as program units. A unit is a container of compiled Java classes with support for “typed” connections. There are two types of units: atoms, which are built from Java classes, and compounds, which are built from atoms and other compounds. Units import and export Java classes. Jiazzi implements the main idea of Aspect-oriented Programming, separation of concerns, by modularizing the code of a concern into a unit, regardless if the code crosscuts Java classes that refers to different names or requires extra arguments to be propagated through method calls. Units are linked and classes included in those units are connected. The connected classes are also called packages. Jiazzi is using an expressive linking language, acting as aspect configuration language, to link units. Packages are described by packages signatures. Package signatures are constructs used to describe the visible structure of classes in a Java package. A package signature contains a class signature that describes the structure of a class included in a package. A class signature is similar to a class definition Java source files. A unit signature describes the structure of a unit’s imported and exported packages, each of which is described using a package signature. [25]

The advanced component programming in Java is supported by Jiazzi basic features such as:
- External linking where the user of a component resolves dependencies of its external class by making the component as flexible as possible for client programmers

- Hierarchical composition that allows combination of multiple components into a larger encapsulated component making possible the incremental construction of software

- Separate compilation where components and units can be compiled and type checked independently of other components and units enabling development of large programs and deployment of components in binary form

- Flexible hiding that refers to the ability of a component to accept imported class implementations which supply more methods than it expects as well as to export class implementations that supply more methods than its clients expect. Flexible hiding allows for flexible composition and encapsulation. Flexible composition means the exact matching is not required when connecting to a component’s imports and flexible encapsulation is a way of allowing a component to restrict access to exported classes.

- Inter-component sub-classing where a component can define a subclass of an imported class as well as import subclasses of its exported classes. Inter-component sub-classing is necessary for grouping classes and class extensions into components.

- Cyclic linking where component linking can resolve mutually recursive dependencies among components enabling natural component organizations and naturally spanning component boundaries.

Class imports, exports, and sub-classing between units can be combined into an open class pattern, which allows the addition of features to classes without changing the sub-typing
relationship between classes or modifying their source code. Units have abilities to express mixins, where an exported class subclasses an imported class. [25]

One of Jiazzi’s features called separate compilation is very important from aspect-oriented development perspective. It promotes the separate reasoning, independent development, and binary deployment of code that implements concerns. [24] Jiazzi accommodates concerns modularization by using:

- Open class patterns that allow units to modularize concerns whose implementations crosscut object and class boundaries
- Open signatures that are supported by units to refine details of methods and classes while the unit undergoes linking.

One of Jiazzi weaknesses is that it cannot separate concerns whose implementations are deeply tangled with other code requiring invasive weaving and meta-programming mechanisms such as provided by AspectJ. If we use AspectJ terminology, we may say that Jiazzi is limited to member and around advice.

Jiazzi’s main concern is on simplifying and advancing code modularization with a simple linking paradigm.

An example of Jiazzi implementation is presented in Appendix K. It shows an aspect-oriented program implemented using Jiazzi. The presented example illustrates that using a simple idea of linking is sufficient to implement aspect-oriented paradigm.

The advantage of Jiazzi is that it does not require extensions to the Java language. Jiazzi is integrated with Java using a stub generator and an external linker. Both stub generator and external linker operates on class files. The linker re-writes class files to form the binary forms of units that can be loaded into Java Virtual Machine. Jiazzi implements Aspect-oriented
Programming with units enabling open classes and open signatures. The separate compilation of units allows separation of concerns. The biggest weakness of Jiazzi is the fact that it cannot modularize concerns whose implementations are tangled into the statements and expressions of method definitions.

DEVELOPMENT IN .NET AND AOP#

In 1997, Microsoft created .NET framework as a set of types, classes, services, and tools to fulfill the following goals:

- Provide a development platform for both Internet and distributed applications
- Simplify applications development
- Improve inter-operability and integration between system and applications
- Support a multi-language environment. [16]

The .NET framework comprises of the elements such as languages and developer tools, base class library, and Common Language Runtime (CLR). The most important element of base class library is the Code Document Object Model CodeDOM namespace that provides interfaces, classes, and architecture to represent the structure of a source code document independent of language as an object graph or tree. [13]

Another important element of the .NET framework is Common Language Runtime (CLR) that maintains the code performing activities such as compilation of the code into native code and provides memory management. It is analogous to the Java Runtime Environment (JRE). There are three categories of Common Language Runtime: the type, the execution, and the Metadata systems. Types supported by the framework are either value such as built-in value and user defined types or reference types such as user defined objects, interfaces, and pointer
Code that runs outside of Common Language Runtime control is called **unmanaged code**. Figure 13 shows the execution model in .NET framework.

During the execution of an application by Common Language Runtime (CLR), the Microsoft Intermediate Language (MSIL) compiles the source code of executed application into machine code. Figure 14 shows .NET compilation process.

The fundamental parts of any .NET development are assemblies. An **assembly** is a primary building block of a .NET application with the following functionality:

- Includes code required for execution with Common Language Runtime
• Forms the security rules
• Types boundary as an associated assembly
• References boundary using the manifest. The manifest contains metadata used for resolving conflicts.
• Maintains version units within Common Language Runtime (CLR)
• Deploys units within Common Language Runtime (CLR).

Figure 15 show the structure of an assembly.

Figure 15. The structure of an assembly

An assembly contains elements such as:

• Manifest that describes an assembly
• Metadata that contains information about classes included in an assembly
• Modules that contain the Microsoft Intermediate Language (MSIL) code used during compilation
• Resources that can be anything from text to image files enabling an assembly to be self contained. [16]

Aspect-oriented C# language called AOP# was developed by Mario Schupany, Christa Schwaninger, and Egon Wuchner from Siemens Corporation. The new aspect-oriented language should be:
• Easy to learn that means the language should not extend existing language as well as it should use a standard compiler and constructs
• Offering clear separation of base and aspect code
• Offering easy mechanism to join base and aspect code
• Able to switch on and off aspects that is a concept called *aspectual polymorphism*
• Able to use any language supported by the .NET framework as either base or aspect code. [16]

AOP# language architecture comprises of three parts: application assembly, aspect assembly, and XML Connector. Assemblies are used to hold specific code. Application assembly holds the base code such as application and business code and aspect assembly holds the aspect code. The XML Connector is coordinating component used to associate aspects with base code. Figure 16 shows AOP# architecture.

Figure 16. AOP# architecture

Compilation of base and aspect classes is similar to compilation of .NET language and both classes are compiled into Microsoft Intermediate Language (MSIL). After compilation assemblies are created. The function of XML Connector is to provide AOP Environment
with information on which base classes should have aspects. The base code is intercepted by
AOP Environment at runtime by using the Profiling and Metadata API of .NET framework.
During the interception AOP Environment adds the aspect code into the base code. This
method of interception has one weakness because Profiling and Metadata API does not
handle unmanaged code.
Aspects in AOP# are regular .NET classes inherited from a base class Aspect and declared as
abstract. AOP# implementation of aspects is done in two steps:

- Declaration of an interface expected from a base class to which the aspect is woven.
  An interface includes two methods: IsRequired and IsExtended. IsRequired method is
  used to interact with weaved base classes (base classes to which the aspect was
  woven) and IsExtended method extends base class methods.

- Implementation of the aspect methods corresponding to the idea of the around advice
  in AspectJ.

Base and aspect classes are mapped together based on the description provided by XML
Connector. AOP# has ability of turning on and off aspect at runtime and this feature is called
Aspectual Polymorphism. Figure 17 shows an example of an aspect in AOP#.
AOP# supports both static and dynamic crosscutting. Static crosscutting means that the language allows a developer to add variables and methods to existing types and dynamic crosscutting means that the language allows a developer to define additional implementation to run at well defined point in the program. AOP# uses the Join Point model that is a sub set of the AspectJ Join Point model and uses similar constructs as AspectJ. [16] The constructs used by AOP# are:

- Aspect that is a modular unit for crosscutting concerns
- Advice that is the same as in AspectJ, but it has to be a valid C# code
- Join point that is the same as in AspectJ. The developer has reflective access to a join point by using a variable called ThisJoinPoint.

Figure 18 shows AOP# engine architecture.
A parser, shown in figure 18, is used to analyze the source code as well as to model namespace, class, function, and variable items. A parser models the source code in the form of Abstract Syntax Tree (AST). The next step is to map aspects to base methods using XML connector and Aspect Deployment Descriptor. After mapping is completed, the weaving is performed. Weaving assures that all advices specified in Aspect Deployment Descriptor are inserted into the target methods. The last step involves compiling the woven code into an assembly using a standard C# compiler.

AOP Engine comprises of parts such as parser, AST representation, weaver, compiler, and Aspect Deployment. Aspect Deployment is the most important element of AOP#.

Let’s describe each part of AOP Engine.

The purpose of a parser is to analyze the source file and find out whether it is a base or aspect source file. During the parsing process, regular expressions classes providing a mechanism for capturing large amount of text are used. For example, a namespace directive can be inspected to find the possible combinations using the Code Identifier.
The Abstract Syntax Tree (AST) representation relies on CodeDOM API and code modeled to a tree that is language independent. Several classes, involved in the process of representation, create the tree and generate information about the source file.

Aspect weaver combines aspect and non-aspect code. Weaving is done of two type of crosscutting implementations: dynamic and static. Several classes are involved in the weaving process performing the following functions:

- Weaving code at method join points
- Checking each method to find out if it has an associated advice. If a method has an associated advice, a MethodJoinPoint object that contains the advice is created.
- Referencing the aspect code to be woven at the join point
- Obtaining the advice code.

Weaving of dynamic crosscutting implementation involves:

- Identifying the target methods using Aspect Deployment Descriptor
- Inserting the advice into the Abstract Syntax Tree object graph
- Weaving of aspect and base code for before and after advice checking if a method has an advice using hasAdvice method
- Creating a MethodJoinPoint object for each method having an advice
- Weaving code at the object graph level.

Weaving of static crosscutting implementation involves creating a custom attribute called introduction and associating the attribute representing data with an element within a program. Compiler is used to compile woven code into an assembly logging and reviewing any errors or warnings generated during the compilation process. AOP# uses standard C# compiler and
a developer may specify input and output files. The compiler can be used to create executables and libraries.

**Aspect Deployment Descriptor** is like the main configuration file for the application. It is a XML file that contains the mappings of aspects to targets and the base and aspect code directories.

An example of a simple AOP# application is presented in Appendix L. It illustrates how a simple application can be designed and developed using object-oriented and aspect-oriented paradigms.

Unfortunately, AOP# is only a research prototype that has not been implemented yet. AOP# has advantages such as separation of crosscutting concerns and decreasing the size of code. The language is easy to use, because it does not extend existing C# language. However, it also has disadvantages such as:

- Lack of support for unmanaged code that means the code not being in control of Common Language Runtime (CLR)
- Parser limitations related to not handling inner classes, structs, delegates and events, and expressions and statements within methods
- CodeDOM Api limitation related to not supporting variable lists, parameter keyword such as params, expressions and statements, and attribute targets.

The full evaluation of AOP# language can be done after its first implementation.

**IBM IMPLEMENTATION OF ASPECT-ORIENTED PROGRAMMING**

IBM Rational Unified Process (RUP) is a popular software development process framework used to support advanced risk management in the requirements management process. [33] Recently IBM created a new tool called IBM Rational Method Composer
(RMC) to build custom delivery processes and used by organizations to create and manage processes suited for their business models and principles. Rational Method Composer (RMC) is also used by Rational Unified Process (RUP) and other processes frameworks such as IT Infrastructure Library (ITIL), Project Management Body of Knowledge (PMBOK), IBM Rational SUMMIT®Ascendant, and Eclipse Process Framework (EPF) as a source of best practice guidance.

IBM is seriously considering the use of Aspect-oriented Approach (AOP) for describing IBM Rational Unified Process (RUP) extensions with IBM Rational Method Composer (RMC) as a supporting platform. [33] The reasons for using aspect-oriented paradigm in process engineering are:

- It supports a modular way of capturing process extensions
- Tools, implemented by common process engineering, may help in customizing and accepting processes
- It allows to localize process changes and extensions as well as to apply them without necessity of changing existing delivery process.

Let’s explore how Aspect-oriented Approach can be applied to Rational Unified Process (RUP) customization.

Table 1 below shows how to apply aspect-oriented concept to a process extension definition template. [33]

Table 1. The Extension Definition template implementation using AOP

<table>
<thead>
<tr>
<th>Concept</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concern</td>
<td>Define a concern that is a single process extension use case such as “A RUP PM Role has inadequate responsibilities”, “A program director wants tighter control over projects”, or “Insufficient vendor communication”. Provide a description of the concern and the proposed solution. Provide description of cases when process elements crosscut the process.</td>
</tr>
</tbody>
</table>
Join Point(s) | Provide a list of join points
---|---
Point cut | Define the point cut that incorporates the identified join points. It summarizes an approach that uses one of the RUP/RMC extensions techniques to grouping aspects in RMC.
Aspect | Provide references to RMC method packaging elements that contain implementation of the aspects. There are three possible options:
- Write the complete implementation of the aspect here including valid links to the documents in a native format if richer implementation is required
- Provide a summary of the aspect implementation describing new and extended content elements. Detailed implementation of the aspect is deferred to RMC.
- Provide URL address link to the implementation of the aspect in RMC.
Advice

The template presented in table 1 is used for describing process extension in real-life process customization and tailoring projects. The usage of Aspect-oriented Approach is more beneficial when a change happens at many points in the process.

Table 2 shows AOP examples and RUP equivalents and examples.

Table 2. Examples of AOP examples and RUP equivalents and examples

<table>
<thead>
<tr>
<th>Crosscutting concern</th>
<th>AOP example</th>
<th>RUP equivalents and examples</th>
</tr>
</thead>
</table>
This example shows classes with multiple methods. Some methods need to store details about the user who accessed the method and the money amount that was transferred.

| Join point | “When a method that affects employer financials is called” | “At the onset of iteration”
| --- | --- | ---
| Point cut | pointcut employerFinancialUpdates (Employer e, User u): call (public void updateEmployerFinancial* ()); In this example, the employerFinancialUpdates point cut refers to join points where method of the name beginning with “updateEmployerFinancial” is called. | pointcut RUP_Element.BasicMethodElement.Task _set outputArtifactIsIterationPlan (): output_is (RUP_Element.BasisMethodElement.WorkProduct.rup_iteration_plan) In this example, the point cut outputArtifactIterationPlan describes tasks (join points) that modify the IterationPlan artifact. |
| Aspect | public aspect logTransaction { pointcut employeeFinanceUpdates(Employer e, User u) : call (public void updateEmployerFinancials* ()) && args (e, u); after (Employer e) returning : employerFinanceUpdates(e, u) { << Advice >> } } | Extension Work Product (Artifact, Deliverables) Extension Activity(Task, Activity, Iteration, Phase) Extension Role Extension Step Extension Capability Pattern Extension Delivery Process |
| Advice | System.out.println("\tUser : " + u.getFullUserName() + " has made a financial transaction through "); System.out.println("\tmethod " + th) In this example, the name of the user accessing the method and the method name are printed. | The purpose of this task is: • To identify the classes that perform a use case’s flow of events • To distribute the use case behavior to those classes using analysis use case realizations • To identify the responsibilities, attributes, and associations of the classes • To note the usage of architectural mechanisms |
The weaving process on Aspect-oriented Programming can be implemented at load-time or runtime. Unfortunately, Rational Method Composer (RMC) does not support a process runtime environment and, as a consequence of this limitation, it can only support source level weaving.

Rational Method Composer (RMC) supports several process extension techniques such as:

- Process plug-ins that uses modular architecture, supporting expanding and extending existing method content with new methodology and technology guidance, provides a solution for modularizing existing process content, and allows incremental creation of process extensions
- Method configuration that simplifies the assemble process from logical building blocks and allows publishing a delivery process as a navigable HTML structure accessible by the implementation team
- Content variability that supports automatic propagating of updates and replacements to the content library and updating references to the modified content in the delivery process. This technique allows easy design time implementation of crosscutting concerns.
- Capability pattern that defines reusable activities sets
- Process contribution that is used to make changes to a delivery process by either replicating content from other processes or dynamically extending the process
- Integrated search that is a search for an entry within defined scope including both process and method content.
Rational Unified Process (RUP) extensions process requires a good knowledge of an element’s dependencies. Rational Method Composer (RMC) navigability tool can be used to identify and trace content dependencies. Figure 19 shows an example of element dependencies in RMC.

Figure 19. An example of element dependencies in RMC

When using Aspect-oriented Approach (AOP) to describe process concerns (extensions) as well as when creating process point cuts, a good knowledge of Rational Unified Process (RUP) structure (meta-model) is required. The tool’s structure is based on IBM Rational’s Unified Method Architecture (UMA). Table 3 shows a sample of hierarchical representation of the RUP meta-model. [33]

Table 3. A sample of hierarchical representation of the RUP meta-model

<table>
<thead>
<tr>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
<th>Level 4</th>
<th>Level 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUP_Element</td>
<td>BasicMethodElement</td>
<td>Role</td>
<td>Work Product</td>
<td>Artifact</td>
</tr>
</tbody>
</table>
The most important part of the Process Tailoring discipline is process extension (concern) capturing. This process is accomplished through the three initial activities:
- Process Analysis that consists of concern identification, analysis, and capturing
- Process Modeling that consists of join point identification and point cut declaration
- Process Description that consists of aspect definition and advice creation.

In cases when Rational Method Composer (RMC) process configurations is used as a technique for merging content from number of different plug-ins, the process of extension aspects may go on through the Process Configuration activity. Figure 20 shows the Process Tailoring discipline.

Figure 20. The Process Tailoring discipline

The Extension Definition template presented in table 1 is used to capture Rational Unified Process (RUP) customization requirements such as:
• Produce a work product at the wrap-up of a repeatable activity that produces a new work product at the onset of an iteration such as when a stakeholder wants to receive an “iteration report” or “integrated data model”

• Perform an action when a work product changes. For example, when a stakeholder wants to be immediately informed about planned changes to be made to a single or multiple work products.

• Replace a work product throughout the process lifecycle that may occur when some organizations want to produce an “Architecture Blueprint” instead of the Software Architecture document.

It looks like Aspect-oriented Approach is a good choice for building program extensions. However, it requires good understanding of Rational Unified Process (RUP) meta-model and availability of consistent extension mechanism. Also, organizations must build a model of the existing implementation process in either Rational Method Composer (RMC) or Eclipse Process Framework (EPF) because both tools contain comprehensive method content library and flexible extensibility patterns based on the Software Process Engineering Metamodel (SPEM) standard. [33]

The next chapter will summarize the entire master’s essay providing conclusions and recommendations drawn.
CHAPTER V

CONCLUSIONS AND RECOMMENDATIONS

This paper analyzed Aspect-oriented paradigm, its concept, features, and its implementations. Aspect-oriented paradigm was compared to the Object-oriented Programming that is the most popular programming paradigm. During the comparison several issues related to application development such as security, real-time concerns, and error and failure handling were considered.

The theory behind Aspect-oriented programming sounds promising. The idea of separation of crosscutting concerns seems to be the right answer to problems happening during object-oriented development. It is interesting that it took almost 25 years to create the Aspect-oriented paradigm since the first theory has been formulated by D.L. Parnas, a Canadian software engineer. The idea of separating crosscutting concerns in the source code and weaving them later at either compile-, load-, or run-time sounds like a brilliant solution to a difficult problem.

The comparison between Aspect-oriented Programming and Object-oriented Programming brought the following results:

- Security concerns are handled much better by aspect-oriented application offering advantages such as separation of concerns, ability to easy add new security concerns, and faster development of the system by developers

- Real-time concerns include several concerns such as thread scheduling and dispatching, synchronization and resource sharing, asynchronous thread termination, memory management, physical memory access, asynchronous event handling, and asynchronous transfer of control. The general result is that Aspect-oriented
programming has benefits related to separation of concerns without affecting performance of an application. When analyzing performance for specific concerns, Aspect-oriented Programming improved performance for thread scheduling and dispatching, physical memory access, and asynchronous event handling concerns. Aspect-oriented Programming provided no performance improvement for synchronous and resource sharing, asynchronous thread termination, memory management, and asynchronous transfer of control. The memory management concern may be handled better by Aspect-oriented Programming when certain improvements are made.

- Error and failure handling concerns are handled better by aspect-oriented application offering advantages such as clear separation of concerns from the rest of code, reduction of duplicated code, and easy maintenance of exception code. However, Aspect-oriented Programming has one weakness related to the difficulty of understanding the interaction between exception code and base code.

Several implementations of Aspect-oriented Programming such as development in Java using AspectJ, JMangler, Aspect Werkz, and Jiazzi and in .NET using AOP# were analyzed. The results of analysis were following:

- AspectJ is probably the most popular implementation of Aspect-oriented Programming. It expands Java language abilities by adding aspect-oriented features such as separating of crosscutting concerns, reusability of code, aspect-aware debugger, documentation generator, and easy maintenance of code retaining all benefits of Java. AspectJ code may sometimes be hard to follow, especially when developing complex systems or adding new functionality to the existing aspect.
- JMangler is a framework for generic interception and transformation of Java programs at load time. It has a lot of benefits such as transformation of Java programs without access to source code as well as support for: load time weaving, arbitrary class loaders, combination of independently developed transformers, classes’ modification without affecting transformation performance, and caching modification. JMangle has weaknesses related to not handling un-anticipating composition of independently developed transformers and not transforming system classes. The tool is powerful, but it is not easy to use.

- Aspect Werkz is a framework that uses Java language to implement aspects. It offers power and simplicity as well as easy integration of Aspect-oriented Programming in both new and existing projects. Unfortunately, Aspect Werkz retains all limitation of Java language such as complex to implement runtime weaving.

- Jiazzi is an enhancement of Java language that implements Aspect-oriented Programming with units enabling open classes and open signatures. It supports separate compilation of units allowing separation of concerns. A big weakness of Jiazzi is the fact that it cannot modularize concerns whose implementations are tangled into the statements and expressions of method definitions.

- AOP# is a research prototype that implements Aspect-oriented Programming with .Net. It has some potential benefits such as separation of crosscutting concerns and decreasing the size of code, but it also has weaknesses such as lack of support for unmanaged code, and parser and CodeDOM Api limitations. Parser limitations are related to not handling inner classes, structs, delegates and events, and expressions
and statements within methods. CodeDOM Api limitations are related to not supporting variable lists, parameter keyword, and attribute targets.

The paper also presented a proposal to implement Aspect-oriented Approach for describing IBM Rational Unified Process (RUUP) extensions with IBM Rational Composer (RMC) as supporting platform. The last implementation proposed by IBM shows that Aspect-oriented Programming is seriously considered as a good alternative to Object-oriented Programming.

Currently, Aspect-oriented Programming paradigm is considered as an immature and an academic project with little industrial implementations. As a consequence of it, this paradigm has not been extensively tested to catch all possible problems related to its implementation.

Summing up, Aspect-oriented Programming paradigm is a promising approach that one day may become as popular as Object-oriented Programming.
REFERENCES


http://ti.arc.nasa.gov/m/pub/archive/2001-0240.pdf


http://foosel.org/_media/uni/mitschriften/sedesign/aopmodularity.pdf


http://technology.asu.edu/files/documents/tradeshow/May05/Krishnan,%20Srinivas.pdf


http://www.engr.mun.ca/~dpeters/papers/necec05_ShakerPeters.pdf


http://www.cse.cuhk.edu.hk/~elisa/papers/AO4RTJ.pdf


APPENDIX A

Example of a prototype of Aspect–Oriented Programming including aspectual decomposition and short description of weaving process

The example shows one of the prototypes of Aspect-oriented Programming developed based the domain presented in Table 4. The example presents only small portion of the entire system.

Table 1 shows examples of aspects in Aspect-oriented Programming which enable developers to think and program [15]

Table 4. Examples of aspects in Aspect-oriented Programming which enable developers to think and program

<table>
<thead>
<tr>
<th>Domain</th>
<th>Distributed computing</th>
<th>Image processing</th>
<th>Numerical simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aspects</td>
<td>What the objects do</td>
<td>Image filters</td>
<td>Algorithm</td>
</tr>
<tr>
<td></td>
<td>Their location</td>
<td>Control structure</td>
<td>Numerical stability</td>
</tr>
<tr>
<td></td>
<td>Communication</td>
<td>Memory usages</td>
<td>Data structures</td>
</tr>
<tr>
<td></td>
<td>Synchronization</td>
<td>Sharing</td>
<td>Memory locality</td>
</tr>
</tbody>
</table>

Based on the above domain, the aspectual decomposition breaks the system down into several key aspects such as the basic functionality, the communication aspect describing the communication strategy of sending messages, and the coordination aspect synchronizing of the threads activity.

Figure 21 show how the identified aspects can be developed using a simplified C++/Java language and the imperative Object-oriented Language. The modules included in the Figure 21 define a bounded stack in the system where:

- The communication aspect deals with the system parameters deciding what parameters are to be copied and to what extent.
• The coordination aspect deal with system methods defining their attributes and pre=conditions.

Figure 21. An actual decomposition example

```java
class BStack {
    Integer head;
    Element els[MAX];
    // other variables here
    void! insert (Element e) {
        if (head == MAX) !;
        else els[head++] = e;
    }
    Element! remove () {
        if (head == 0) !;
        else els[--head];
    }
    Element! top () {
        if (head == 0) !;
        else els[head];
    }
    void newClient (Client c){
        //some admin code
    }
};
```

```java
interface BStack {
    void! insert (greg Element);
    greg Element! remove ()
    greg Element! top ()
    void newClient (copy Client:id);
};
```

```java
relax BStack {
    autoex insert, remove, newClient;
    mutex (insert, remove);
    mutex (remove, top);
};
```

The entire system, which sample is shown in the Figure 21, requires about 43 line of code, when the system written in Java can be done in 120 lines of codes. [15]

It illustrates how the Aspect-oriented Programming is more efficient in implementation of communication and synchronization strategies.

What does make the Aspect-oriented Programming more efficient?

The difference is in dealing with crosscutting concerns. Object-oriented Program such as Java must put small amount of code all over the program. Those small pieces of code are important when the program is compiling and executed.

Aspect-oriented Programming approach is different. The new process introduced in Aspect-oriented Programming is called weaving. Basically it takes local aspect-description programs and weaves them throughout the executable code. [15]
Aspect-oriented approach does not change the source because weaving is done either at compilation time or at runtime.

Figure 22 shows the basic components of an Aspect-oriented Programming system.

Figure 22 Basic components of an Aspect-oriented Programming system
APPENDIX B

Example comparing Aspect-oriented and Container Managed Security

The container, usually Enterprise Java Bean (EJB) container, implements the security infrastructure that is external to the component system. [32] Therefore the component system relies only on the security services provided by the Enterprise Java Bean (EJB) container. The container supports user’s authorization based on the role (access control) and authentication. It means that a user must pass through the security provided by the container before the access to a component system is granted. The security setup is located in the deployment descriptor external to the component system. It allows setting the security independently of the component system, for example during the system deployment.

The example presented below shows how the Enterprise Java Bean (EJB) container may restrict access to BankEJB only to users granted the Client role.

The Figure 23 shows a system descriptor providing declarative security

Figure 23. An example of a system descriptor providing declarative security

```
<session>
   <ejb-name>Bank</ejb-name>
   <ejb-class>BankEJB</ejb-class>
   ...
   <security-role-ref>
      <role-name>Client</role-name>
   </security-role-ref>
   ...
</session>
```

It is possible to add a programmatic security to the declarative security. In such case, the code for the programmatic security is located inside the component system and it can obtain some security information from a container. Unfortunately, the information received from a container is limited to the user and its security role. Implementing security inside the
component system is time consuming and requires an expert knowledge of security mechanisms and policies. The disadvantage of declarative and programmatic security is the fact that both cooperate only with Java 2 Enterprise Edition (J2EE) application servers.

Aspect-oriented Programming idea is to separate crosscutting concerns into independent modules called aspects. Aspects define object, methods, and events related to security. In this comparison Aspect-oriented Programming is implemented using AspectJ. AspectJ is an extension of Java that introduces new features related to the Aspect-oriented paradigm in Java language such as:

- **Point cuts** that are collections of join points
- **Advices** that are special constructs attached to point cuts
- **Join points** are well-defined points in the dynamic execution of an application. Join points are the elements such as method calls or method executions that specify how classes and aspects are related.

The comparison between Aspect-oriented and Container Managed security is based on the aspect-oriented implementation with AspectJ (aspect-oriented extension of Java) and Container Managed security implementation with pure Java. In the comparison, we compare how the AspectJ and Java applications handle security related concerns such as:

- Identification and authentication
- Access control
- Accountability and audit.

**Comparison of handling identification and authentication based on Java Authentication and Authorization Service (JAAS).**
Figure 24 shows authentication example of an application created with Java language and
Figure 25 shows an example authentication example of an application created with AspectJ.
The authentication in both Java and AspectJ implementations requires the initialization of the
security manager that includes configuration of a login module, creation an instance of
LoginContext, and log in.

Figure 24. Authentication example of an application created with Java language

class BankClient {
    LoginContext lc = null;

    public static void main(String[] args) {
        // Callback to get username and password. Required by LoginContext
        AppCallbackHandler handler = new AppCallbackHandler( "scott",
                "echoman" );
        try {
            lc = new LoginContext( "Bank", handler );
            lc.login();
            catch( LoginException e ) {
                // ...
                }
            // ...
            BankHome homeBank = (BankHome) ctx.lookup( "ejb/Bank" );
            Bank bank = homeBank.create();
            System.out.println( bank.getAccountInfo( "bill" ) );
            // ...
            try {
                lc.logout();
            } catch( LoginException e ) {
                // ...
                }
        }
    }
}

The lines of code in gray identify the code that is used to authenticate of bank component.
class BankClient {
    public static void main(String[] args) {
        // ...
        BankHome homeBank = (BankHome) ctx.lookup("ejb/Bank");
        Bank bank = homeBank.create();
        System.out.println(bank.getAccountInfo("bill"));
        // ...
    }
}

aspect BankAspect {
    LoginContext lc = null;

    pointcut mainExecution():
        execution( public static void main( .. ) );

    // Login before execution of main()
    before(): mainExecution() {
        AppCallbackHandler handler = new AppCallbackHandler( "scott", "ehowman" );
        try {
            lc = new LoginContext( "Bank", handler );
            lc.login();
        } catch( LoginException e ) {
            // ...
        }
    }

    // Logout after execution of main()
    after() returning: mainExecution() {
        try {
            lc.logout();
        } catch( LoginException e ) {
            // ...
        }
    }
}

The comparison of both examples of codes demonstrates that the AspectJ (aspect-oriented extension of Java) provides separation of security concern (authentication) from the rest of application’s code. The authentication code is included in the aspect called BankAspect. The Java application example does not provide separation of security concern (authentication) from the rest of application’s code.

Comparison of handling access control based on authorization server compatible with Resource Access Decision Facility (RAD).
The access control is based on an external (to an application) access control server. The container managed security is either declarative or programmatic. The declarative security is based on an application descriptor shown in Figure 23. The container is responsible for identifying clients and their security roles as well as roles for accessing an application. The programmatic security is coded inside an application. The communication with the container is limited to security information related to a client such as client id and its security role.

The access control in both Java and AspectJ is using the authorization server compatible with Resource Access Decision Facility (RAD).

Figure 26 shows access control example of an application created with Java language and Figure 27 shows access control example of an application created with AspectJ.

Figure 26. Access control example of an application created with Java language

class BankEJB implements SessionBean {
    public String getAccountInfo( String customer ) {
        // access control
        // get instance of RADConfig to prepare parameters for access_allowed()
        RADConfig radConfig = RADConfig.getInstance();
        // prepare an instance of ResourceName to identify the customer’s account
        ResourceName resourceName = radConfig.getResourceName( RADConfig.CUSTOMER_ACCOUNT, customer );
        // prepare the operation that is performed on the resource
        String operation = radConfig.getOperation( "getAccountInfo" );
        // prepare user’s security credentials
        SecAttribute[] attributeList = radConfig.getSecAttributes( SecAttribute.USER, context.getCallerPrincipal().getName() );
        // get reference to the RAD service
        RAD rad = RADLocator.getInstanceOfRAD();
        // make an access decision
        if ( !rad.access_allowed( resourceName, operation, attributeList ) ) {
            // do something if access has been denied
            // ...
        }
        // main functionality
    }
}
Figure 27. Access control example of an application created with AspectJ

class BankEJB implements SessionBean
{
    public String getAccountInfo(String customer)
    {
        //main functionality
    }
}

privileged aspect BankAspect
{
    pointcut bankMethodsInvocation(String customer):
        execution(String getAccountInfo(String)) && args(customer);

    Object around(String customer, SessionBean obj):
    bankMethodsInvocation(customer) &&
    this(obj) { //access control
        RADConfig radConfig = RADConfig.getInstance();
        ResourceName resourceName = radConfig.getResourceName(RADConfig.CUSTOMER_ACCOUNT, customer);
        String operation = radConfig.getOperation(thisJoinPointStaticPart.getSignature().getName());
        SecAttribute[] attributeList = radConfig.getSecAttributes(SecAttribute.USER,
            ((BankEJB)obj).context.getCallerPrincipal().getName());
        RAD rad = RADLocator.getInstanceOfRAD();
        if(!rad.access_allowed(resourceName, operation, attributeList))
            //do something if access has been denied
        else
            return proceed(customer, obj);
    }
}

Again, the comparison of both examples of codes demonstrates the AspectJ (aspect-oriented extension of Java) provides separation of security concern (access control) from the rest of application’s code. The access control code is included in the aspect called BankAspect. The Java application does not provide separation of security concern (access control) from the rest of application’s code. Any changes to the access control security in AspectJ code require modification to the aspect code only. Similar changes to the access control mechanism in pure Java code require modifications in several modules through the entire application.

Comparison of handling accountability and audit
Accountability and audit are about collecting and analyzing the activities of the application and detecting any possible security violations as well as finding out what caused security violations. The container managed security does not provide standard functionality for accountability and audit. The only way to handle this security is to use programmatic security and include the security in an application. Accountability and audit can be implemented much easier with AspectJ.

The Figure 28 shows accountability example of an application created with AspectJ

Figure 28. Accountability example of an application created with AspectJ

```java
aspect BankAspect {
    pointcut bankMethods():
        execution( public * bank.*( .. ) ) && this( SessionBean );

    // Log information after throwing BankSecurityException from SessionBeans which belong to the bank package
    after() throwing (BankSecurityException e): bankMethods() {
        Log log = Log.getInstance();
        log.write( e );
    }
}
```

AspectJ can also handle audit, but the information is directed to audit module such as an intrusion detection module. Again, the AspectJ code shows that security code included in the BankAspect is separated from the rest of an application.

Table 5 shows brief summary of comparison of Aspect-oriented and Container Managed Security.

Table 5. Brief summary of comparison of Aspect-oriented and Container Managed Security

<table>
<thead>
<tr>
<th>Security concern</th>
<th>Container Managed Security</th>
<th>Aspect-oriented Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identification and authentication</td>
<td>Limited to the container provided security</td>
<td>Limited to the container provided security</td>
</tr>
<tr>
<td>Access control</td>
<td>Limited to the container provided security</td>
<td>Modification to an application code required</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>Accountability</td>
<td>No Standard security functions available</td>
<td>Modification to an application code required</td>
</tr>
<tr>
<td>Audit</td>
<td>Not supported</td>
<td>Modification to an application code required</td>
</tr>
</tbody>
</table>
APPENDIX C

Example analyzes a Personal Information Management System implemented with AspectJ (aspect-oriented extension of Java) and object-oriented design

A Personal Management System is designed to keep track of personal information such as user’s agenda, contact information including personal and business contacts and to work as reminder about daily tasks. The system is deployed to a very popular device called a Palm Pilot. The security concerns handled by the system are: access control, authentication, and authorization. [7]

The Figure 29 shows the class diagram of Personal Management System modified to implement security concerns using object-oriented design

Figure 29. The class diagram of Personal Management System

As I mentioned before the system manages appointments, contacts, and security concerns. Security concerns are enforced by the access control model and application’s functionality.
Before security is implemented, the security rules need to be specified. In this case security rules may be simple such as:

- The owner or administrator of the system can perform every possible operation
- Access to Contacts is limited to the Owner only
- All other users of the system have read only access.

If we wanted to implement this system using object-oriented programming, we have to associate the PIMUnit module with the owner by inserting an owner attribute into this module and initialize the attribute when the module is created. For the user identification purpose, the authentication functionality needs to be added to the PIMSystem module. The authorization is required to make sure of what type of operations a user can perform. To implement authorization, the signature of the operations must be modified to pass user’s identity from PIMSystem to the authorization checks. As for the quite simple system, its implementation is quite complex. The changes related to security concerns are shown in Figure 29.

The implementation of the same security concerns with AspectJ (aspect-oriented extension of Java) is shown in Figure 30.

Figure 30. Aspect-oriented implementation of the system
The system implementation consists of three aspects: OwnerManagement, Authentication, and Authorization. The OwnerManagement aspect is handling storage and initialization of the PIM unit owners. Each instance of an aspect is associated with every PIMUnit object and every unit is decorated with an owner attribute. When the unit is created, an owner attribute is initialized by the after advice. The Authentication aspect authenticates users and handling functionalities similar to the login() method in object-oriented design. The Authentication aspect’s attributes include the current user as well as the operations requested by the current user. The Authorization aspect handles the access control. The around advice, in this aspect, verifies the current user’s authority to perform requested operation and either accepts or denies the request for an operation. The restricted Access point cut enforces the user verification.
The object-oriented implementation seems to be complex. On the other hand, aspect-oriented implementation is offering separation of concerns that has several advantages such as:

- Easier maintenance of the system because security related code is in one place
- Easier expansion of the system
- Easier development that means a developer may concentrate on the core of the problem instead of taking care about side issues such as code consistency
- Better understanding of which security measures are implemented and how they function.

In addition, an aspect-oriented approach supports a capability based model in which an owner may delegate access privileges to other users.
APPENDIX D

Example compares real-time concerns implementation using Real-time Java (RTJava) and AspectJ (aspect-oriented extension of Java) applications.

In this comparison several real-time concerns are evaluated such as:

- Thread scheduling and dispatching
- Synchronization and resource sharing
- Asynchronous thread termination
- Memory management
- Physical memory access
- Asynchronous event handling
- Asynchronous transfer of control.

In this example, two applications handling a real-time sentient traffic simulation are compared. One of them is object-oriented application called OOSim created with Real-time Java (RTJava) and the other is aspect-oriented application called AOSim created with AspectJ (aspect-oriented extension of Java). These applications used in this comparison were developed based on the Sentient Traffic Simulator created by the Distributed Systems Group in Trinity College. The simulator imitates a traffic management system. It lets vehicle to self-drive on four-lane highway. Each vehicle has a sensor installed to establish speed and position of other vehicles on the highway. There are two rules that apply to each vehicle on the highway:

- A vehicle on a highway must obey traffic rules and let emergency vehicle to pass
- A vehicle must reduce speed when approaching a slower moving vehicle.
The OOSim application has the same functionality as the original simulator and is implemented with RTJava constructs. OOSim has several asynchronous events handler classes dealing with asynchronous events. SimulatorUtil supports utility functions. SimulationRTThread and SimulationPeriodThread subclasses support real-time threading. Figure 31 shows Sentient Traffic Simulator Class Diagram created with RTJava.

![Figure 31. Sentient Traffic Simulator Class Diagram created with RTJava](image)

The AOSim was created by refactoring the OOSim application. During this process the crosscutting concerns were identified and separated into aspects. Join points were located in aspects and point cuts were created for the purpose of weaving. Figure 32 shows Sentient Traffic Simulator Aspect-oriented design diagram.
Figure 32. Sentient Traffic Simulator Aspect-oriented design diagram

Figure 32 illustrates that one aspect is created for each real-time concern. Arrows, in Figure 32, point into the core classes created after weaving process.

Let’s begin the comparison on how OOSim and AOSim handle real-time concerns. Each real-time concern, implemented in both OOSim and AOSim applications, is compared. The results of comparison are summarized and the final conclusion is drawn.

**Thread Scheduling and Dispatching concern implementation**

OOSim application has two method of handling this concern such as:

- Subclassing RealtimeThread that is used for threads with the same behavior and the same instantiation arguments. There are two subclasses in OOSim application:
  
  - SimulationRTThread
  - SimulationPeriodicThread
• RealtimeThread objects constructor that is used for direct creation of threads. The object’s constructor arguments must be included in the application by developer when a thread is created. This method is used for threads that required different parameters.

Figure 33 shows the OOSim application constructor for RealtimeThread.

Figure 33. The OOSim application constructor for RealtimeThread

```java
public RealtimeThread (SchedulingParameters scheduling,
    ReleaseParameters release, MemoryParameters memory, MemoryArea
    area, ProcessingGroupParameters group, java.lang.Runnable
    logic)
```

AOSim application is using aspects to create and start real-time threads. To create the RealtimeThread object, several separately created parameters are required. The parameters are created when the createRTThread aspect captures all calls to the SimulationRTThread constructor and substitutes them with parameter that creates and initializes code. Before each aspect is started, the feasibility check is performed. The construction calls for all threads are captured by the ExecuteRTThread. After those construction calls, the feasibility check is executed. Figure 34 shows AOSIm code handling threads.

Figure 34 AOSim application code handling threads

```java
pointcut createRTThread():execution(* SimulationRTThread.<init>();
RealtimeThread around():createRTThread() {
    // thread creation code
}

pointcut executeRTThread():call(* * RealtimeThread.<init>(..));
after() : executeRTThread() {
    // code for checking feasibility and
    // starting of the real-time thread
}
```

Memory management concern implementation

OOSim application was created with RTJava. RTJava uses three types of memory such as physical, immortal (persistent until Java Virtual Machine termination), and scoped memory
The immortal memory is used for objects such as Car, EmergencyVehicle, ViewableCar ListOfCars, and Road when garbage collection is not needed. OOSim application uses immortal object memory created with and without arguments constructors. Creation an object with constructor without arguments involves a call to the newInstance method of the ImmortalMemory class. Creation an object with constructors with arguments is more complicated and involves Class and Object arrays as well as the Constructor object.

AOSim application uses aspects to simplify the code so core classes do not need to reference the ImmortalMemory object. Figure 35 shows AOSim application code handling memory management.

![AOSim application code handling memory management.](image)

```java
aspect MemoryManagement {
  pointcut createViewableCarMemory(String thread, boolean isVeh) : call(simulation.ViewableCar.new(String, boolean)) & args (thread, isVeh);
  ViewableCar around(String thread, boolean isVeh): createViewableCarMemory(thread, isVeh) {
    //building and initialising of ViewableCar
    //parameters and allocation to immortal memory
  }
}
```

The point cut called `createViewableCarMemory` creates immortal memory for the ViewableCar object as well as it captures all ViewableCar constructor calls and their parameters. The constructors implementation in the ViewableCar object is substituted by the implementation stated in the around advice. Each immortal object in the memory management has corresponding point cut.

**Synchronization and resource sharing implementation**

In OOSim application, the ListOfCars object needs locking mechanism because all vehicles access the list to get information about nearest vehicles. The synchronization of the
ListofCars object is done by re-factoring the code from the original application into real-time.

AOSim application includes the SynchronizationAspect to handle synchronization and resource sharing. Figure 36 shows AOSim application code handling synchronization.

Figure 36. AOSim application code handling synchronization and resource sharing

```java
pointcut updateAction(): execution(* ViewableCar.*Poll(..));
before(): updateAction(){
    // wait and lock object
}
after(): updateAction(){
    // notify object waiting for locks
}
```

The point cut updateAction states that all methods ending with ‘Poll’ are to be synchronized. The waiting and locking of shared resources are implemented before the specified method such as clearPoll is executed by the before advice. The after advice serves the purpose of notification after the operations in the capture method have been executed.

**Asynchronous event handling implementation**

In OOSim application, events are generated based on information providing by sensors. Examples of such information are: an emergency vehicle behind, a slower car is ahead, or the lanes on either side are busy. Events based on sensors information are handled asynchronously. It allows the application to handle internal and external events to Java Virtual Machine. Figure 31 shows classes implementing asynchronous events. An example of such a class is FaultTriggeredEventHandler class. When an exception is thrown, this class is handling the thrown exception.

In AOSim application, aspects bind events to handlers and perform inheritance declarations in asynchronous event handler classes. [34] Figure 37 shows AOSim code handling asynchronous events.
Figure 37. AOSim code handling asynchronous events

```java
pointcut bindEvent(Class eventClass, String bindName) :call(*
    .bindAsyncEvent (Class, String)) & args (eventClass,
    bindName);
void around(Class eventClass, String bindName) {
    // perform binding actions
}
declare parents: eventHandler.*EventHandler extends
    AsyncEventHandler;
```

All calls to the bindAsyncEvent method having arguments of Class and String type are captured by the point cut. The around advice contains the implementation that is executed instead of the bindAsyncEvent method. The separation of concerns is achieved because the aspect includes all constructs associated with the binding of asynchronous events. The ‘declare parents’ declaration lists all classes ending in EventHandler. It also means that the eventHandler package extends the AsyncEventHandler class. In AOSim application, the inheritance declaration is done only once when in OOSim all asynchronous event handler classes must extend the AsyncEventHandler class. This is an advantage of AOSim application over OOSim.

**Asynchronous transfer of control implementation**

In the OOSim application, if a vehicle processing is based on incorrect sensor information, the process must be interrupted. The interruption is done in two ways such as:

- By throwing an exception, controlled by the asynchronous transfer of control, into the threads carrying out processing

- By using the fire approach when blocks of code can be interrupted by calling an object’s method called interruptible*() such as interruptibleUpdateViewableCar() or Java’s exception handling mechanism.

The interruption methods are implemented in the ViewableCar object.
AOSim application is modularizing the fire approach. Figure 38 shows AOSim code handling asynchronous transfer of control.

Figure 38. AOSim code handling asynchronous transfer of control

```
declare soft: AsynchronouslyInterruptedException : execution(* *.run
  (AsynchronouslyInterruptedException))
  & within(simulation);

pointcut catchAIE() : call (* *.interruptible*{.}) &
  within(simulation);
after() : catchAIE() {
  // catch AsynchronouslyInterruptedException
}
```

The ‘declare soft’, in Figure 38, is related to checked exceptions and it means that all methods having the exception as an argument must throw an AsynchronouslyInterruptedException. The ‘within’ construct means that the aspect applies to all classes in the application.

The point cut catchAIE is created to catch all calls to thread’s interrupted method.

**Asynchronous thread termination implementation**

In OOSim application, thread termination is done using Asynchronous Transfer of Control (ATC) technique together with the interrupt method. The interrupt method is defined by RealtimeThread in the RTJava API allowing restoration of inconsistent data.

AOSim application implements asynchronous thread termination in similar way as the transfer of control. The only difference between those two implementations is that in asynchronous thread termination implementation a call to the interrupt() method is made after the AsynchronouslyInterruptedException is thrown. Figure 39 shows AOSim code related to asynchronous thread termination.
pointcut catchAIE() : call (* *.interrupted*{}()) &&
        within{simulation};
after() : catchAIE() {
    // catch AsynchronousInterruptedException
    // call to interrupt()
}

Physical memory access implementation

Physical memory access is not implemented because of hardware constraints. Physical memory access implementation analysis for both OOSim and AOSim applications are done based on theoretical study of both systems.

In OOSim application, objects that require physical memory access are Vehicle, Car, and EmergencyVehicle. Those objects have frequently accessed and modified attributes such as velocity and position coordinates and might benefit from being allocated to fast physical memory. Fast memory allocation may involve setting of specific position and sizes of physical memory locations as well as throwing and handling of a number of exceptions. [36]

AOSim application implementation might involve modularizing exception-catching code at each point of physical memory access. To accomplish this, the point cut should capture all methods ending with PhysicalMemoryAccess and the after advice should provide implementation for catching different exceptions thrown during access of physical memory addresses. Figure 40 shows AOSim code related to physical memory access.

Figure 40. AOSim code related to physical memory access

pointcut catchPhysicalMemoryException() : call (* *.PhysicalMemoryAccess{}()) && within{simulation};
after() : catchPhysicalMemoryException() {
    // catch all thrown exceptions
}

C&K metrics is used to compare implementation of real-time concerns in OOSim and AOSim applications. C&K metrics is a set of metrics proposed by Chidamber and Kemerer.
in 1991 that analyzes object-oriented software. The following measures were used in comparison:

- **Weighted Methods per Class (WMC)** that is a measure of the number of methods implemented within a class
- **Depth of Inheritance Tree (DIT)** that is the maximum distance from a class node to the root of the tree
- **Number of Children (NOC)** that is the number of immediate subclasses of a class
- **Coupling between Objects (CBO)** that is a count of the number of other classes from which elements are used such as attribute accesses between classes
- **Response For a Class (RFC)** that is the number of methods that can potentially be executed in response to a message received by an object of a class
- **Lack of Cohesion of Methods (LCOM)** that is the degree to which methods within a class are related to one another in terms of shared variables.

Because C&K metrics was designed to analyze object-oriented application, the clarifications on how to use several measures, when analyzing AOSim application, are required. Those clarifications include:

- Aspects are counted as classes and advice blocks as methods to use Weighted Methods per Class measure
- Classes like in Depth of Inheritance Tree measure are not used in AOSim application
- AOSim application does not need sub-classing and Number of Children measure counts subclasses
- Aspects are coupled to classes only if the aspects explicitly name the classes to use in Coupling between Objects measure
• Calls to methods affected by aspects are counted when calculating Response For a Class

• Point cuts and advice blocks are treated as methods to use Lack of Cohesion of Methods.

It looks like that C&K metrics is not a perfect tool to analyze Aspect-oriented application.

Table 6 shows change of C&K metrics measures due to Use of Aspects. The values of each of the metrics are given as a reduction or increase the change that is incurred by the particular metrics as a direct consequence of using aspects. [34] It means that a negative value indicates reduction due to use of aspects (improvement) and a positive value indicates an increase (reduction in performance).

Table 6. Change of C&K metrics measures after implementing AOSim application

<table>
<thead>
<tr>
<th>Metric</th>
<th>Thread Sched &amp; Disp</th>
<th>Mem Mgmt</th>
<th>Synchro &amp; Sharing</th>
<th>Asynch Events</th>
<th>Asynch Control Transfer</th>
<th>Asynch Thread Term</th>
<th>Phys Mem Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>D/I/T</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NDC</td>
<td>-2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CBO</td>
<td>-2</td>
<td>-2</td>
<td>0</td>
<td>-10</td>
<td>-3</td>
<td>-3</td>
<td>-15</td>
</tr>
<tr>
<td>RFC</td>
<td>2</td>
<td>14</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>LCOM</td>
<td>0</td>
<td>0</td>
<td>-2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7 shows combination of Individual Results for System Properties after implementing AOSim application. The negative values mean improvement and positive values mean a reduction in performance.
Table 7. Combination of Individual Results for System Properties after implementing AOSim application

<table>
<thead>
<tr>
<th>Understandability</th>
<th>Weighted Methods per Class</th>
<th>Depth of Inheritance Tree</th>
<th>Number of Children</th>
<th>Coupling Between Objects</th>
<th>Response For Class</th>
<th>Lack of Cohesion of Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg:</td>
<td>WMC</td>
<td>DIT</td>
<td>NOC</td>
<td>CBO</td>
<td>RFC</td>
</tr>
<tr>
<td>+1.1</td>
<td>-3</td>
<td>-0.14</td>
<td>-0.28</td>
<td>5</td>
<td>-6.5</td>
<td>-0.28</td>
</tr>
<tr>
<td>+1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0.27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8 shows average change over all system properties for real-time concerns after implementing AOSim application. The negative values mean improvement and positive values mean a reduction in performance. Gray colored cells indicate which individual results are used when accessing system properties.

The average results show overall improvements for most measures with exception for ‘Weighted Methods for Class’ and ‘Response For a Class’. As per areas, the average results show improvements only for reusability when understandability, maintainability, and testability show no improvements.
Table 8. Average change over all system properties for real-time concerns after implementing AOSim application

<table>
<thead>
<tr>
<th></th>
<th>Thread-Sched</th>
<th>Mem-Mgmt</th>
<th>Synchro</th>
<th>Events</th>
<th>Control</th>
<th>Thread-Term</th>
<th>Mem-Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understandability</td>
<td>-0.25</td>
<td>3.50</td>
<td>1.50</td>
<td>-0.50</td>
<td>2.25</td>
<td>2.25</td>
<td>-1.25</td>
</tr>
<tr>
<td>Maintainability</td>
<td>0.00</td>
<td>4.67</td>
<td>2.00</td>
<td>-0.67</td>
<td>3.00</td>
<td>3.00</td>
<td>-1.67</td>
</tr>
<tr>
<td>Reusability</td>
<td>-1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-1.80</td>
<td>0.60</td>
<td>0.60</td>
<td>-2.20</td>
</tr>
<tr>
<td>Testability</td>
<td>-0.75</td>
<td>3.00</td>
<td>1.00</td>
<td>-0.75</td>
<td>0.75</td>
<td>0.75</td>
<td>-2.25</td>
</tr>
</tbody>
</table>

Based on both individual and average results, aspect-oriented implementation provides some improvements to the application when handling concerns such as thread scheduling and dispatching, asynchronous events handling, and physical memory access. Those improvements are related to reusability and testability. As per concerns such as memory management, asynchronous transfer of control, synchronization and resource sharing, and thread termination aspect-oriented implementation does provide any improvement. The fact that there is no improvement in some concerns may be related to the results of ‘Response For a Class’ and ‘Weighted Methods per Class’ measures.
APPENDIX E

Example compares object-oriented and aspect-oriented implementation of RG application developed at Xerox PARC for image processing

Reverse Graphics (RG) application is based on a mode of image processing. The application operates on entire images, rather than working a pixel at a time. RG application is built on a collection of primitive filters that process one or more input images to create an output image. [26]

In object-oriented implementation, objects represent images and messages represent filters. The language of this implementation is CLOS (Common LISP Object System). An object is stored in as an array of pixel values representing the image data. The original application used primitive operations to allocate a new image object to hold the result and iterate over inputs to fill in the pixel values of their output. The new image object is returned after their iteration is completed. Object-oriented implementation is using higher level operations defined by the method definition mechanism. New operations include calls to primitives and other operations. Automatic garbage collector takes care of any unused objects. Implementing object-oriented application requires fixing three serious performance issues related to the original implementation such as:

- Redundancy in computation that means the original application has tendency to make redundant calls. It is happening when a higher level filter requests a lower level filter that has already been computed, at the request of a different higher level filter.
- Excess memory turnover that means frequent and expensive garbage collections. It happens during operation of fresh images when each image, especially a large one, is
allocated at a very high rate forcing automatic memory management to perform frequent and expensive garbage collections

- Inefficient data cache usage means that the access to the intermediate results causes many cache misses and slow access times. This problem is related to processing of large images when the on-chip data cache is filled quickly without ability of being reused. [26]

The solution to eliminate redundant calls is memoization, a technique used to implement purely functional systems. As per this solution, each primitive filter keeps a record of the input and output images. When a new image is processed, it is compared to previously stored input images and if such image was already processed, the recorded output is returned instead of processing the input again.

Excess memory turnover and inefficient data cache usage problems are usually the side effect of producing many intermediate images. The problem can be solved by using fusing loops and applying several primitive filters in a single iteration. For example, in the situation when one filter produces an image that is immediately used by another filter and both filters can iterate over the image in the same order, the computations of those filters can be fused meaning each pixel can be used by the second filters as soon as it is released by the first filter. To make sure that excess memory turnover problem is fixed in all possible scenarios, allocated memory should be reused instead of being de-allocated and re-allocated.

The implementation of the solutions to problems related with the original application is neither easy nor feasible and it creates the code scattered throughout the application that is hard to read, debug, and maintain. The implemented solutions may create another problem
related to the automatic memory management. Figure 41 shows an example of the object-oriented RG application implementation.

Figure 41. Example of the object-oriented RG application implementation

(defmethod color-by-number-generator ((I image))
  (let ((left-neighbor (right-neighbor! I))
        (right-neighbor (left-neighbor! I))
        (up-neighbor (up-neighbor! I))
        (down-neighbor (down-neighbor! I)))
    (if (already-computed 'color-by-number-generator I)
        (precomputed-result 'color-by-number-generator I)
        (remember-precomputed-result
         'color-by-number-generator)
        (let ((result (find-available-storage)))
            (loop-pointwise index
               ((ipix I))
              (left-pixel left-neighbor)
              (right-pixel right-neighbor)
              (up-pixel up-neighbor)
              (down-pixel down-neighbor))
            (if (or (not (equal ipix left-pixel) ipix))
                (not (equal ipix right-pixel) ipix)
                (not (equal ipix up-pixel) ipix)
                (not (equal ipix down-pixel) ipix))
                (setf (aref result index) ipix)
                (setf (aref result index) 0))))))

Aspect-oriented implementation is using similar techniques such as memoization, loop fusion, and memory management to address to address problems with the original application.

The implementation of memoization means that for every message an invoking primitive filter is sent. Results of invocations (outputs) need to be recorded. If the same input is invoked again, the recorded output may be retrieved instead of repeating the same invocation of a primitive filter again.

The implementation of the loop fusion technique means that for every message an invoking primitive filter is sent to do the comparison between the loop structure required to calculate the filter and the loop structure required to calculate the argument. The comparison is done before computing filter’s arguments. If both loops have the same structure, a single loop structure is generated to compute both the argument and the filter values.
The memory management technique implementation means maintaining a pool of free arrays. When a sent message invokes a primitive filter, the array is allocated to hold the output value from the free pool. For each returned message, the identity of each argument array is checked and if that array is not used in any message, it is stored in the free pool. [26]

Using the aspect-oriented terminology, sent messages are the join points between the component code and the aspect code. Aspect-oriented implementation consists of component part coded with CLOS (Common LISP Object System) language and aspects part coded with Lisp aspect language. Both components and aspect are combined during the weaving process creating an executable. Aspects run at the compile time to see join points. Join points are represented by a data flow of all primitive message invocations in the basic program. Each aspect is handling one performance technique and is called on each primitive method invocation in the graph. For the purpose of aspect-oriented implementation the following aspects are created:

- Memoization aspect that has only access to the node it communicates with
- Fusion aspect that has access to all nodes with ability to modify nodes
- Memory management aspect that has access to all nodes with ability to modify nodes.

The memoization aspect is invoked when a call-frame is to be created. The function called try-memoization is defined in the aspect and accepts two arguments: the caller and the nodes represented input and a generator for a node. The function is creating a record for new call-expressions. The created record is made of the node corresponding to the call-expression. If the call-expression was previously used, the earlier node is return. Figure 42 shows example of memoization aspect code.
Figure 42. Example of the memoization aspect code

```
(define-aspect memoization
  :when :on-method-call
  :exec try-memoization)

(defun try-memoization (call-expr call-maker)
  (or (get-memoized-call call-expr)
      (set-memoized-call call-expr (funcall call-maker))))

(defun get-memoized-call (call-expr)
  ;;; if call-expr was seen before, return earlier call
  ...)

(defun set-memoized-call (call-expr call)
  ;;; record that call-expr was seen before, and returns call
  ...)
```

The fusion aspect is called after the join point representation is available. When the aspect is called, the function called try-fusion is executed. The purpose of this function is to fuse loops of each node with the loops of its input. It is possible to add more rules to the aspect to handle other fusions. Figure 43 shows example of the fusion aspect code.

Figure 43. Example of fusion aspect code

```
(define-aspect fusion
  :when :first
  :exec try-fusion)

(defun try-fusion (nodes)
  (mapc #'try-fuse-with-inputs nodes)

(defun try-fuse-with-inputs ((loop loop-node))
  (let ((new-loop))
    (dolist (input (loop-inputs loop))
      (let ((fused (try-fuse input loop)))
        (if fused (setq new-loop fused))))
    new-loop))

(defun try-fuse ((input loop-node) loop)
  (let ((loop-shape (loop-shape loop))
        (input-shape (loop-shape input)))
    (cond
      ((and (eq (car loop-shape) 'pointwise)
             (eq (car input-shape) 'pointwise))
       (fuse loop input 'pointwise
             :inputs (splice ...) :loop-vars (splice ...) :body (subst ...))
       (t nil)))))
```
The memory management aspect is called after the fusion aspect. When the memory management aspect is called, the function called allocate-memory is executed. The purpose of this function is to set the list of nodes in the order of execution. There are two kinds of nodes: future nodes that appear after the node in this list and available nodes that appear before the node in this list. Available nodes are not children of future nodes. Figure 44 shows example of the memory management aspect code.

Figure 44. Example of the memory management aspect code

```
(define-aspect ctm
  :when :second
  :exec allocate-memory)

(defun allocate-memory (nodes)
  (let ((available-nodes nil)
        (future-nodes nodes))
    (loop while future-nodes do
      (let* ((node (car future-nodes))
             (previous-allocation
              (there-exists b available-nodes
               (for-all n future-nodes
                (not (immediate-subnode? b n)))))))
       (make-allocation node previous-allocation)
       (setq available-nodes
               (cons node (remove previous-allocation available-nodes)))
       (setq future-nodes
               (cdr future-nodes))))
```

The performance of the original RG application, object-oriented implementation, and aspect-oriented implementation is compared using a table recognizer program that analyzes parts of a scanned document identifying table and plain columnar text. To show performance evaluation 128 x 18 pixels images are used. The results of comparison are shown in Table 9.

Table 9. Results of measurement of different RPG implementations

<table>
<thead>
<tr>
<th>Original RPG application</th>
<th>Object-oriented implementation</th>
<th>Aspect-oriented implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>manually optimized</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timing</td>
<td>&gt; 101 seconds</td>
<td>195 milliseconds</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Number of images allocated</td>
<td>158,250</td>
<td>233</td>
</tr>
<tr>
<td>Source Complexity (line of code count)</td>
<td>1,500 lines</td>
<td>35,213 lines</td>
</tr>
</tbody>
</table>

Based on Table 9 results, the performance of aspect-oriented application is similar to object-oriented application. The main advantage of aspect-oriented application is simplification of the source code without affecting the performance.
APPENDIX F

Example on how aspect-oriented implementation is handling error and failure handling.

In this example, the ability of Aspect-oriented Programming to improve exception handling in Java application created with Eclipse is analyzed. The example of code presented in Figure 45 shows duplicated code in Eclipse plug-in. The class EclipseSynchronizer contains two methods: endBatching() and members() implementing the same exception handling strategies. [6]

Figure 45. Duplicated code in Eclipse plug-in

```java
public class EclipseSynchronizer implements IFlushOperation {
    public void endBatching(...) throws CVException {
        try (...) catch (TeamException e) {
            throw CVException.wrapException(e);
        }
    }
}

d public IResource[] members(...) throws CVException {
    try {...} catch (CoreException e) {
        throw CVException.wrapException(e);
    }
    ...
}
```

Let’s see how AspectJ (aspect-oriented extension of Java) solves the duplication code and separation of concerns problems. We use the Error Handling Aspect design pattern features to prevent code duplication and to separate exception handling from the rest of application code. In order to achieve it, an aspect EclipseSyncHandler is created. EclipseSyncHandler aspect defines a point cut eh() to associate advice handler1 to both methods endBatching() and members(). The purpose of handler1 advice is to implement exception handlers as well as to connect them to different parts of a program using the composition mechanisms of AspectJ. Figure 46 shows UML diagram illustrating implantation of patterns to solve exception handling related problems.
Detailed structure of patterns comprise of three aspects: EHASpect1, EHASpect2, and GenericEHASpect. Each aspect has corresponding advice such as handler1() and handler2() and methods throwing exceptions such as exc1 and exc2. In aspect-oriented terminology those methods are join points. An advice implements an exception handler, connects it to normal classes (normal code), and executes it when exceptions are raised within the context methods. The context methods such as contextMethod1() and contextMethod2() are defined in normal classes. Normal code is represented by normal classes such as NormalClass1 and NormalClass2. Connecting an exception handler to multiple normal classes helps to prevent duplication of code. Another way of preventing duplication of code is by using abstract error handling aspects. This method is useful when a handler advice is common to multiple error handling aspects. In such case, it is done by binding the advice to an abstract point cut and making it concrete in the sub-aspects. [6] The example of abstract aspect in structure of pattern is GenericEHASpect. In order to simplify any possible modification to exception handling code, it is important that join points are easy to be captured by aspect-oriented language and the contextual information is easily accessed by the error handling aspects. Figure 47 shows the structure of the patterns.
To complete analysis on how aspect-oriented language handles exceptions, we need to analyze the runtime interactions of the Error Handling Aspect pattern. The analysis of runtime interactions are based on two cases.

**Case 1:** An error handling aspect successfully handling an exception raised by a method invoked by a client.

These are the following steps during this case: [6]

- Method contextMethod(), included in NormalClass, is invoked by a client object
- The exception E is raised while executing the method
- The EHAspect catches the raised exception E and tries to handle it
- The aspect’s handler executes normally without raising an exception
- The application resumes execution without interruption.

There are two possible results of such case. In the first one, the method contextMethod() continue its execution. In the second, the method sends a normal message that does not require raising an exception to the client object. This kind of exception handling is used when
an application does need to be interrupted or stopped. Figure 48 illustrates steps in case 1 when handling an exception.

Figure 48. Steps in case 1 when handling an exception

Case 2: This case is based on nested try-catch blocks. When the inner handler advice cannot handle the raised exception, it raises an exception caught by the outer handler advice.

These are the following steps during this case: [6]

- Method contextMethod(), included in NormalClass, is invoked by a client object
- The exception E is raised while executing the method
- The innerHandler() catches the raised exception E and tries to handle it
- Failing to handle the exception the innerHandler() raises the exception Einner
- The outerHandler() catches the raised exception Einner and raise the exception Eouter
- The Eouter exception is sent to the NormalClass which redirects it to the client object.

This case illustrates the situation when the first level exception handling fails to handle an exception, does not cause any problem because the second level can catch the exception and successfully handles it. Figure 49 illustrates steps in case 2 when handling an exception.
It looks that aspect-oriented language handled both cases quite well, but it had some weaknesses.

The benefits of aspect-oriented implementation in respect to error handling are separation of error handling code, reduction of duplicated code, wide range of exception successfully handled, and easy maintenance of exception code.

However, there are also some weaknesses such as difficulty to understand the interaction between exception code and base code, necessity for refactoring of the base code (in some specific cases), difficulties to handle checked exception, and increase in exception code size.
APPENDIX G

Example on how aspect-oriented C# and Spring.Net deal with error handling

In this example, we analyze how C# and Spring.Net deal with error handling by refactoring a class called OrderDAO. The class is used to access database to either update an order on database or retrieve order information from database. It includes two methods: UpdateOrder and GetOrder. If any of those methods raises a database exception DbException, debugging information about an order are logged using Log4Net ILog object. Figure 50 shows a class OrderDAO.
When examining the code, we detect two problems related to:

- Code duplication in catch section of blocks that means there are similar catch section for functions GetOrder(int OrderID) and UpdateOrder(Order order)
- The ToString function used to log arguments may not return all information about argument objects. [23]

Aspect-oriented C# and Spring.Net are used to modify the class OrderDAO to fix problems related to code duplication, and to separate and improve the exception handling code.

ExceptionHandlingAspect is used to fix the problem with code duplication.
The ExceptionHandAspect defines the advice and point cut. Figure 51 shows the class diagram for a new code design.

Figure 51. The class diagram for new code design

![Class diagram](image)

When simplifying the error handling in C# and Spring.Net code, the following steps are used to handle an exception:

- A method throws an exception
- The GetProxy method of the ExceptionHandlingAspect intercepts the thrown exception
- The method AfterThrowing from IThrosAdvice is called giving access to the method that has thrown an exception as well as to method arguments
- The thrown exception is sent to upper call stack.

Figure 52 shows LogArgumentsThrowsAdvice code. The purpose of this advice is to gather arguments and to log them using ILog.Debug(object).
Figure 52. LogArgumentsThrowsAdvice code

class LogArgumentsThrowsAdvice : IThrowsAdvice
{
    private ILogger logger;
    public LogArgumentsThrowsAdvice()
    {
        log4net.Config.XmlConfigurator.Configure();
        Logger = LogManager.getLogger(this.GetType().Name);
    }
    public void AfterThrowing(MethodBase method, object[] args, object target, Exception exception)
    {
        if (! (exception is DbException))
            return;
        if (args != null && args.Length > 0)
        {
            foreach (object arg in args)
            {
                Logger.Debug($"{arg}" + arg);
            }
        }
    }
}

Figure 53 shows GetProxy method code. The purpose of this method is to intercept exceptions being thrown. Based on the argument methodRE, the SdkRegularExpressionMethodPointcut is created. The created point cut is used to get method, which has thrown an exception. Later this method is intercepted. The advisor of type DefaultPointcutAdvisor, a module in aspect class, groups SdkRegularExpressionMethodPointcut point cut and LogArgumentsThrowsAdvice advice

Figure 53 GetProxy method code

private ILog logger;
    public LogArgumentsThrowsAdvice()
    {
        log4net.Config.XmlConfigurator.Configure();
        Logger = LogManager.getLogger(this.GetType().Name);
    }
    public void AfterThrowing(MethodBase method, object[] args, object target, Exception exception)
    {
        if (! (exception is DbException))
            return;
        if (args != null && args.Length > 0)
        {
            foreach (object arg in args)
            {
                Logger.Debug($"{arg}" + arg);
            }
        }
    }
}
The functionality of Spring.Net can be used to create proxy. It is done by the Spring.Net IOC container when the logic of GetProxy method is abstracted to the configuration file.

Figure 54 shows Exception Handling Aspect interaction.

Figure 54. Exception Handling Aspect interaction

Exception Handling Aspect interaction includes interceptions of thrown exceptions (1.1) and redirection of calls by the advisor to its advices (1.2).

The C# and Spring.Net code fixed problems related to code duplication and also successfully separated the exception handling code from the class OrderDAO.

The next step is to analyze C# and Spring.Net abilities to further improve error handling code from debugging perspective. Figure 55 shows how SerializeArgumentsAdvice can improve the error handling code.
Figure 55. SerializeArgumentsAdvice code

```csharp
/// <summary>
/// SerializeArgumentsThrowsAdvice is an around advice which
/// persists debug information.
/// </summary>

class SerializeArgumentsThrowsAdvice : IThrowsAdvice
{
    private BinaryFormatter binaryFormatter = new BinaryFormatter();
    /// <summary>
    /// Persists the target and arguments, if any, to file system.
    /// </summary>
    public void AfterThrowing(MethodInfo method, Object[] args, Object target, Exception exception)
    {
        //this.Serialize(target);
        if (exception is DbException)
            return;
        if (args != null && args.Length > 0)
        {
            foreach (Object arg in args)
            {
                Serialize(arg);
            }
        }
    }

    protected virtual void Serialize(object obj)
    {
        if (obj == null)
            return;
        FileStream fileStream = null;
        try
        {
            String fileName = "" + obj.GetType() + "" + obj.GetHashCode() + ".dat";
            fileStream = File.Create(fileName);
            binaryFormatter.Serialize(fileStream, obj);
        }
        catch (Exception e)
        {
            System.Console.WriteLine(e.StackTrace);
        }
        finally
        {
            if (fileStream != null)
            {
                fileStream.Close();
            }
        }
    }
}
```

The advice SerializeArgumentsThrowsAdvice replaced the advice LogArgumentsThrowsAdvice (see Figure 52). Both advices are similar, but the new advice is using the Serialize and the BinaryFormatter. In the name space
System.Runtime.Serialization.Formatters.Binary is used to persist the object onto file system. [23] The new advice can be easily used with ExceptionHandlingAspect by replacing the name of the old advice (LogArgumentsThrowsAdvice) with the name of new advice (SerializeArgumentsThrowsAdvice) in the GetProxy() method. When creating a new advice, an assumption that arguments are marked as Serializable is made. The new, improved error handling code creates serialized Order objects. We may debug the problem by de-serializing those objects, bringing them back to live .Net CLR, and investigating the Orders arguments.

The new advice illustrates abilities of C# and Spring.Net to create better code by providing detailed debugging information, better than debugging information in the original class OrderDAO.
APPENDIX H

Analysis of any possible problems related to AspectJ (aspect-oriented extension of Java) implementation in aspect-oriented programming.

The functionality of Health Watcher application in respect to error handling concern was analyzed using different scenarios when aspects act as exception signalers or exception handlers. Health Watcher is Web application created using AspectJ (aspect-oriented extension of Java) for citizens to register complains related to issues in health care institutions. Figure 56 show design of Health Watcher application.

Figure 56. Design of Health Watcher application

Based on analysis, related to aspect-oriented implementation, several new problems were found. Fortunately, solutions to the identified problems were identified. [4]

The following problems, categorized by type of scenarios, were found:

1. Scenario when aspects act as exception signalers
a. **Unstable exceptional interface** that is related to several exception thrown by a method when an exception may be not handled causing application to crash or handled by mistake by existing exception handler causing unwanted action. This problem occurs when the behavior of a method depends on how it is called. There are two possible solutions to fix this problem. The first solution is to create an exception handler for each condition when an exception is thrown and include such an exception handler in every method calling the intercepted method. Second solution is to replace advice which throws an exception with a declare error statement dynamically creating an error at the compile time.

b. **Handlerless signaler aspect** that is related to an aspect advice signaling an exception, but not handled by any exception handler causing application to crash or handled by mistake by existing exception handler causing unwanted action. This problem occurs when an aspect advice signals an unchecked exception and no exception handler is defined to catch it. There are two possible solutions to fix this problem. The first solution is to create an Error Handling Aspect intercepting specific points in the code in which the thrown exception needs to be caught and handled. The second solution is to create an Error Handling Aspect intercepting every aspect which may signal an exception or to include a catch clause inside every advice which signals the exception.

2. Scenario when aspects act as exception handler

   a. **Late binding error handling aspect** that is related to an aspect, created to handle an exception, intercepts the correct point in the base code to catch the exception, but the exception never reach the exception handler. This problem occurs when an
exception is caught by a catch clause on the base code before it reaches the exception handler. There are three possible solutions to fix this problem. The first solution is to avoid ‘catch all clause’ when developing exception handling code. The second solution is to replace ‘catch all clause’ with specific catch clauses. The third solution is to create multiple exception hierarchies: one used for exceptions signaled by the main program and others used for exceptions signaled by aspects.

b. **Unmatched error handling aspect** that is related to an aspect intercepting the wrong point in the program execution causing application to crash or handled by mistake by existing exception handler causing unwanted action. This problem occurs when an error handling aspect, created to handle an exception, does not handle it because of a mistake in the point cut expression. The solution to fix this is to correct the mistake in the point cut expression.

c. **Residual or obsolete handler** that is related to an exception handler created to handle a specific exception and no longer required, because a new exception handler was created to handle the same exception or the operation throwing this exception was removed. The obsolete exception handler may catch exceptions by mistake and handle them incorrectly. The solution to fix this problem is to monitor any changes to the program and remove any obsolete exception handler.
APPENDIX I

Example on how AspectJ may help implement resource-pool management.

Resource usage can be optimized by recycling previously used and no longer required resources such as threads and database connections. The AspectJ implementation is done in several steps. [22]

Step 1. TCPP/IP service is used for converting requested strings to upper case

A request for a new connection causes the server to create a new thread. After a connection is terminated, the thread serving this connection is no longer required and terminates naturally.

The code implementing step 1, written in Java, is shown in Figure 57.
Figure 57. Step 1 implementation code

```java
// UppercaseServer.java
import java.io.*;
import java.net.*;
public class UppercaseServer {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.out.println("Usage: java UppercaseServer <portNum>";
            System.exit(1);
        }
        int portNum = Integer.parseInt(args[0]);
        ServerSocket serverSocket = new ServerSocket(portNum);

        while (true) {
            Socket requestSocket = serverSocket.accept();
            Thread serverThread
                = new Thread(new UppercaseWorker(requestSocket));
            serverThread.start();
        }
    }
}

class UppercaseWorker implements Runnable {
    private Socket _requestSocket;
    public UppercaseWorker(Socket requestSocket) throws IOException {
        System.out.println("Creating new worker");
        _requestSocket = requestSocket;
    }
```
Step 2. Adding a thread-pooling crosscutting concern using AspectJ

To accomplish this task, we need to create a class ThreadPool acting as a stack for threads. The class contains two methods. The get() method acquires a thread from the stack. The put() method pushes a thread into the stack. The Delegating Thread class, included in the method put(), delegates the run() method to the "delegatee" worker object.

The code implementing step 2 is shown in Figure 58.
Step 3. Creating an aspect to add thread pooling to the server

After the class ThreadPool was created, we need to create an aspect ThreadPooling used to add thread pooling to the server. Figure 59 shows the ThreadPooling aspect code
Figure 59. The ThreadPooling aspect

```java
// ThreadPooling.java
public aspect ThreadPooling {
    ThreadPool pool = new ThreadPool();
    // Thread creation
    //
    pointcut threadCreation(Runnable runnable)
        : call(Thread::new(Runnable)) && args(runnable);
    Thread around(Runnable runnable) : threadCreation(runnable) {
        ThreadPool.DelegatingThread availableThread = pool.get();
        if (availableThread == null) {
            availableThread = new ThreadPool.DelegatingThread();
        }
        availableThread.setDelegatee(runnable);
        return availableThread;
    }
    // Session
    //
    pointcut session(ThreadPool.DelegatingThread thread)
        : execution(void ThreadPool.DelegatingThread.run()) && this(thread);
    void around(ThreadPool.DelegatingThread thread) : session(thread) {
        while (true) {
            proceed(thread);
            pool.put(thread);
            synchronized(thread) {
                try {
                    thread.wait();
                } catch (InterruptedException ex) {
                }
            }
        }
    }
    // Thread start
    //
    pointcut threadStart(ThreadPool.DelegatingThread thread)
        : call(void Thread::start()) && target(thread);
    void around(Thread thread) : threadStart(thread) {
        if (thread.isAlive()) {
            // wake it up
            synchronized(thread) {
                thread.notifyAll();
            }
        } else {
            proceed(thread);
        }
    }
}
```

The Thread Pooling aspect includes the following constructs:
• Point cut threadCreation() that captures join point and creates a new thread object. The point cut takes one argument: a Runnable object. The new thread is either acquired from the stack or created by the point cut.

• Point cut session() that captures execution of the run() method of any ThreadPool.DelegatingThread objects. The point cut is included in the loop and it never stops working. The point cut is always servicing and it guarantees that created thread never dies. After the request is completed, the thread is no longer needed and is put to the stack.

• Point cut threadStart() that captures a call to the Thread.start() method. The point cut checks if a thread is active and it wakes up any thread in a waiting state. The method isAlive() is used to check the state of a thread.

The example illustrates that AspectJ successfully modularized the resource-pool management concern.
Example illustrates how code coverage can be implemented as a code transformer using J Mangler.

The purpose of code coverage is to collect line coverage information.

The first step is to create a class CodeCoverTransformer that implements the interface CodeTransformerComponent. The class includes following method providing functions necessary for coverage tool:

- Method: recordCoveredLine that is called at run time to print a message to the standard output stream
- Method: transformCode that is called by J Mangler framework at load time.
  J Mangler passes an UnextendableClassSet, containing all classes loaded by a class loader, to the transformCode method
- Method: getIteratorForTransformableClasses that is used to iterate over all classes which need to be transformed. Each iteration returns an instance of ClassGen with information such its name, its constant pool, an instruction factory (utility class for creation of complex bytecode instructions), and its methods.
- Method: addRecordCaller that performs modifications at each step.

When doing transformation of classes we have to exclude the class CodeCoverTransformer from being transformed. During transformation of a class, we iterate over all non-native and non-abstract methods of a transformed class creating an instance of MethodGen for each method. Each method returns an InstructionList instance representing method’s code.

We use the line number table to determine the positions in the instruction list for each line of the original Java source code. [17]. A sample of line number table is presented below.
<table>
<thead>
<tr>
<th>Source File</th>
<th>Byte code</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class Sample { void sayHello() { System.out.println(&quot;Hello&quot;); } }</td>
<td>Method void sayHello() 0 getstatic #8 3 ldc #1 5 invokevirtual #9 8 return</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Program counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

The sample shows that the statements implementing line 3 start at program counter equal to 0 (zero). Each MethodGen instance is converted to the non-modifiable form and all modified methods are written back to the ClassGen instance.

All remaining functions are performed by JMangler.

The addRecordCaller method is responsible for creating the call sequence based on generated instruction list. The instruction list consists of pairs values such as the name of the current class and the line number. Because the current line possibly is the branch target of other instructions (for example of an if/then/else statement), we finally need to redirect all the branches from the old position to start of the new call sequence in order to have the line number recorded in all cases. [17]

The configuration file for JMangler is used to activate the CodeCoverTransformer. The example of such a file is shown below:

```java
transformer CodeCoverTransformer {
    name = org.cs3.jmangler.samples.codecoverage.CodeCoverTransformer;
}
```
The activation of the CodeCoverTransformer starts the Java Virtual machine, loads JMangler and the transformers (specified in the configuration file), and begin execution of the program passed as a command line parameter.

For example, we use the class Hello World to be analyzed by the CodeCoverTransformer.

The class HelloWorld is shown below:

```java
1: public class HelloWorld {
2:     public static void main(String[] args) {
3:         if (args.length > 0)
4:             {System.out.println(args[0]); }
5:         else
6:             {System.out.println("Hello World!"); }
7:     }
8: }
```

The CodeCoverTransformer analyzes the HelloWorld class and creates the output shown below:

```
>> jmangler - cf codecover.config HelloWorld

covered line 3 in class HelloWorld
covered line 6 in class HelloWorld
Hello World!
Covered line 7 in class HelloWorld
```

The output illustrates that the line 4 in the HelloWorld class is not covered.

Figure 60 shows the source code for the Code Coverage Tool
Figure 60. The source code for the Code Coverage Tool

```java
1: import de.fub.bytecode.*;
2: import de.fub.bytecode.generic.*;
3: import de.fub.bytecode.classfile.*;
4: import org.cs3.jwangler.*;
5: import org.cs3.jwangler.tau.*;
6: import org.cs3.jwangler.supplier.*;
7: import java.util.*;
8:
9: public class CodeCoverTransformer implements CodeTransformerComponent {
10: 
11: public static void recordCoveredLine(String className, int line) {
12: System.out.println("covered line "+line+" in class "+className);
13: 
14: private static String recorderMethodName = "recordCoveredLine";
15: private static Type recorderReturnType = Type.VOID;
16: private static Type[] recorderArgTypes = new Type[2];
17: {recorderArgTypes[0] = Type.STRING;
18: recorderArgTypes[1] = Type.INT;
19: }
20:
21: public void addRecordCaller
22: (ClassGen classGen, ConstantPoolGen poolGen,
23: InstructionFactory factory, int lineNo,
24: InstructionList intolist, InstructionHandle beforeHandle)
25: {
26: InstructionList recordCaller = new InstructionList();
27: 
28: InstructionHandle callHandle = recordCaller.append(
29: new LDC_W(poolGen.addString(classGen.getClassName())));
30: 
31: recordCaller.append(new LDC_W(poolGen.addInteger(lineNo)));
32: 
33: recordCaller.append(
34: factory.createInvoke("CodeCoverTransformer", recorderMethodName,
35: recorderReturnType, recorderArgTypes,
36: Constants.INVOKESTATIC));
37: 
38: intolist.insert(beforeHandle, recordCaller);
39: intolist.redirectBranches(beforeHandle, callHandle);
40: } // addRecordCaller
41:
42:
134
public void transformCode(UnmodifiableClassSet classSet) {
    Iterator classIterator = classSet.getIteratorForTransformableClasses();
    while (classIterator.hasNext()) {
        ClassGen classGen = (ClassGen)classIterator.next();
        String className = classGen.getClassName();
        if (className.equals(this.getClass().getName())) continue;
    }

    ConstantPoolGen poolGen = classGen.getConstantPool();
    InstructionFactory factory = new InstructionFactory(classGen);
    Method[] methods = classGen.getMethods();
    for (int methInd = 0; methInd < methods.length; methInd++) {
        Method meth = methods[methInd];
        if (meth.isNative() || meth.isAbstract()) continue;

        MethodGen methGen = new MethodGen(meth, className, poolGen);
        InstructionList instList = methGen/InstructionFactory();
        LineNumberGen[] lines = methGen/LineNumbers();
        for (int lineInd = 0; lineInd < lines.length; lineInd++) {
            LineNumberGen line = lines[lineInd];
            addRecordCaller(classGen, poolGen, factory, line.getSourceLine(),
                            instList, line/InstructionInstruction());
        }
        methGen.setMaxStack();
        methods[methInd] = methGen.getMethod();
    }
} // transformCode

public void sessionStart() {}
public void sessionEnd() {}
public String verboseMessage() { return toString(); }
} // CodeCoverTransformer
APPENDIX K

Example presents an aspect-oriented application called a maze game created in Java and Jiazzi.

A basic maze game involves a player exploring a maze of rooms that are connected by doors and populated with items. The example shows how Jiazzi implements the basic game as well as its abilities to add extra feature to enhance the game.

A basic game is implemented as a Jiazzi package comprised of the following classes: Maze, Entity, Room, Door, Player, and Item. Figure 61 shows the structure of classes described by the package signature “mzbase”.

Figure 61. The structure of classes described by the package signature “mzbase”

```java
signature mzbase = {
    class Maze extends Object { Maze(); : : : }
    abstract class Entity extends Object {
        Entity(); abstract void display(); : : : }
    class Room extends Entity {
        Room(); Item item(int n); : : : }
    class Door extends Entity {
        Door(); boolean enter(Player p); : : : }
    class Player extends Entity {
        Player(String name); void exec(); : : : }
    class Item extends Entity { Item(); : : : }
}
```

Main modules of Jiazzi are atom or compound units. The atom unit called “driver” imports and exports packages. The package “maze”, imported from other unit, creates a dependency of the basic maze game classes. The package “main”, exported from “driver”, supports an application entry point class to other units. Driver’s implementation is hidden from its clients and the entry point is provided by the main method in the Main class. [24] Figure 62 shows the source code of the unit “driver” and the package signature “program”.

136
Figure 62. The source code for the unit “driver” and the package signature “program”

```java
atom driver
{
    import maze : mzbase;
    export main : program;
}

signature program = {
    class Main extends Object {
        static void main(String args[]); }
}
// file: driver/main/Main.java
package main;
class MyMaze extends maze.Maze {
    public class Main extends Object {
    public static void main(String args[])
    {
        maze.Maze maze = new MyMaze();
        maze.Player p = new maze.Player(args[0]);
        maze.Room rooms[] = { : : :};
        maze.Door doors[] = { : : :};
}
```

The implementation of the game is provided by linking “driver” and “base” units and linking is happening in the compound unit called “game”. An atom unit called “base” exports the “mzbase” package containing maze game classes and imported by the unit “driver”. Figure 63 shows the source code of “base” and “game” units.

Figure 63. The source code of “base” and “game” units

```java
atom base
{
    export maze : mzbase;
}

compound game
{
    export main : program;
    export maze : mzbase;
    link unit base, driver;
}
```

As a result of linking “driver” and “base” units, any use of the basic maze game classes included in the unit “driver” becomes the use of the basic maze game classes included in the unit “base”. Figure 64 illustrates all connections between packages included in the unit “game”. Packages “maze” and “main” are also connected to packages exported from the unit “game”. The unit “game” does not import any packages and its classes can be loaded directly
into a Java Virtual Machine. Also, the unit “game” may act as a self-contained Java application, because it provides the executable class Main. [24]

Figure 64. Connections between packages included in the unit “game”

Now, we want to enhance the maze game with additional features called magic and locked features. The magic feature is about opening doors. A player who wants to open a door needs to find and cast spells. Additional classes and methods, described by the package signature “mzmagic” are created to support magic features. Figure 65 shows the package signature “mzmagic” that is built out of package signature “mzbase”.

Figure 65. The package signature “mzmagic”

```java
signature zmagic = mzbase + {
    class Spell extends Item if Spell(); }
    class Door +=
        { Spell neededSpell();
            void setSpell(Spell spell); : : : }  
    class Player +=
        { void addSpell(Spell spell);
            boolean hasSpell(Spell spell);
            void castSpell(Spell spell); : : : }
}
```

The package signature “mzmagic” includes additional class Spell and uses the accumulate operator (+=) to add new methods descriptions to classes described by the package signature “mzbase”. The magic feature is considered as an optional and crosscutting concern across classes Door and Player. Jiazzi solves the crosscutting concern problem with open classes. Open classes can be enhanced with new implementation without the need to modify their
original source code. [24] The atom unit “opmagic” is used to add new implementation to the open package “maze”. The keyword “open” makes the package “maze” a package of open maze games classes. The open package “maze” is described by two package signatures: “mzbase” that describes the imported structure of the maze game classes and “mzmagic” that describes the exported structure of the maze game classes. Because of that the open package “maze” is enhanced with the magic feature described by the package signature “mzmagic”.

Figure 66 shows the source code for the atom unit “opmagic”.

Figure 66. The source code for the atom unit “opmagic”

```
atom opmagic
{ open maze : mzbase -> mzmagic; }

// file: opmagic/maze/Door.java
package maze;
public class Door extends super Door {
    private Spell spell = null;
    public Door() { super(); }
    public void setSpell(Spell s) { spell = s; }
    public Spell neededSpell() { return spell; }
    public boolean enter(Player p)
    { if (neededSpell() != null)
      if (p.casting != this.neededSpell())
        return false;
      return super.enter(p); }
}
// file: opmagic/maze/Player.java
package maze;
public class Player extends super Player
{ Spell casting; }
```

The open class Door, included in the unit “opmagic”, extends the class “_super_Door” that is a special class used by Jiazzi to expose open class functionality to conventional Java source code. Jiazzi generates special classes automatically. Figure 66 shows two ways of open class enhancement. Using the first method, a new method call “setSpell” and a field called casting are added. Using second method, the existing method “enter” is enhanced with new code that addresses the magic feature and also refers to the newly added method “neededSpell” and field casting. [24]
The second extension of the maze game is called locked feature. This feature is implemented by “mzlocked” package signature and an atom unit “locked”. The implementation of the locked feature is done in a similar way to implementation of the magic feature. Both features are combined into the package signature “mzmagloc” describing a package of maze game classes with structure supporting magic and locked features. Open packages “opmagic” and “oplocked” are linked together and such a connection merges them (conceptually) into a single package implementing both features. Figure 67 shows the source for the package signature “mzlocked” and the atom unit “locked”.

Figure 67. The source code for package signature “mzlocked” and the atom unit “locked”

```java
signature mzlocked = mzbase + }
    class Key extends Item ) Key(); }
    class Door +=
        ) Key neededKey();
    void setKey(Key key); : : }
    class Player +=
        ) void addKey(Key key); boolean hasKey(Key key);
    void useKey(Key key, Door door); : : }

atom oplocked }
    open locked : mzbase -> mzlocked;
}
signature mzmagloc = mzmagic + mzlocked;

compound opmagloc }
    open maze : mzbase -> mzmagloc;
    link unit opmagic, oplocked;
}
```

It is important to establish the overriding order of methods implementing magic and locked features. Figure 67 illustrates that the method “enter” implementing the magic feature is overridden in the atom unit “oplocked” implementing the locked feature.

The complete maze game classes with support for magic and locked classes require creating link between the compound unit “opmagloc” and the atom unit “opbase”. Figure 68 shows the source code for the atom unit “opbase” and the compound unit “game2”.

140
Figure 68. The source code for the atom unit “opbase” and the compound unit “game2”

```plaintext
atom opbase 
  open maze : empty -> mzbase;
}
compound game2 }
  export main : program;
  export maze : mzmagloc;
  link unit driver;
  link unit opbase, opmagloc;
}
```

The link between “opmagloc” and “opbase” units is implemented by the compound unit “game2” and it finalizes the development of the maze game application. The unit “game2” can be loaded into a Java Virtual Machine because it does not contain any imported or open packages. Magic and locked features are controlled by Jiazzi’s linking language. The modifications to the configuration of the maze game do not affect maze application’s source code. Figure 69 shows connections established in linking of “game2” and “opmagloc” units.

Figure 69. Connections established in linking of “game2” and “opmagloc” units
APPENDIX L

Example shows how a simple application can be designed and developed using object-oriented and aspect-oriented paradigms.

An application called a Bank Account is created to support clients’ transactions such as deposits and withdraws from their accounts. Transactions are considered a crosscutting concern.

Object-oriented design and application

The design model comprises of four classes such as:

- Account that is a super class for an account holding all details about each client
- CurrentAccount that maintains all transactions related to a current account of a client
- SavingAccount that maintains all transactions related to a saving account of a client
- BusinessAccount that maintains all transactions related to a business account of a client.

Figure 70 shows object-oriented class diagram.
Figure 70. Object-oriented class diagram

Figure 71 shows an example of code taken out of CurrentAccount class. The basic function of the code is to perform deposits into a client’s account. Functionality of the code is quite simple and it can be described in a few steps such as:

- Connection to a database is open
- The transaction is started
- An SQL statement to update Account table is created
- An SQL statement, created in the previous step, is executed
- If the update is successful, the transaction is committed. Otherwise the transaction is rolled back and the error is created.
- Connection to a database is closed. [16]
Similar code, supporting different transactions, is included in all classes.

**Aspect-oriented design and application with AOP#**

The aspect-oriented design model is similar with one exception. Transactions have been removed from classes and included into an Aspect class called AspectTransaction.

AspectTransaction aspect class comprises of two methods beforeTransaction that maintains database connection and transaction setup and afterTransaction that maintains commit or rollback of transactions as well as closes database connection. Figure 72 shows aspect-oriented class diagram.
Figure 72. Aspect-oriented class diagram

Figure 73 shows methods included in the AspectTransaction aspect class.
Figure 73. The AspectTransaction aspect class methods

```java
public void BeforeTransaction()
{
    connection = new SqlConnection(Connection_Source);
    connection.Open();
    tx = connection.BeginTransaction();
    command.Transaction = tx;
}
public void AfterTransaction()
{
    if(message.Equals("Error"))
    {
        tx.Rollback();
    } else
    {
        tx.Commit();
    }
    connection.Close();
}
```

Figure 74 shows how the code presented in figure 62 was changed by introducing an aspect.

Figure 74. Aspect-oriented code: deposit into account

```java
try
{
    string CommandText = "Update Account " +
    " Set Balance = Balance + " + Amount +
    " Where CustomerID = " + base.getID() + "' " +
    " And Type = " + ACCOUNT_TYPE + "'");
    command.CommandText = CommandText;
    command.Connection = connection;
    command.ExecuteNonQuery();
    message = "Balance updated successfully";
}
catch(Exception e)
{
    Console.WriteLine(e.StackTrace);
    message = "Error";
}
return message;
```

The comparison of two applications shows several benefits related to aspect-oriented approach such as:

- Separation of crosscutting concern
- Smaller size of code
• The code is easier to understand and maintain.