



University of Nebraska at Omaha
DigitalCommons@UNO

Student Work

4-2014

Program Comprehension of Aspect-Oriented Programs

Jeffrey Steenbock

University of Nebraska at Omaha

Follow this and additional works at: <https://digitalcommons.unomaha.edu/studentwork>

 Part of the [Computer Sciences Commons](#)

Recommended Citation

Steenbock, Jeffrey, "Program Comprehension of Aspect-Oriented Programs" (2014). *Student Work*. 2896.
<https://digitalcommons.unomaha.edu/studentwork/2896>

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.



Program Comprehension of Aspect-Oriented Programs

**A Thesis
Presented to the
College of Information Science & Technology
and the
Faculty of the Graduate College
University of Nebraska
In Partial Fulfillment
of the Requirments for the Degree
Master of Science
University of Nebraska at Omaha**

**by
Jeffrey Steenbock, CSEP, GSSP-JAVA**

April 2014

**Supervisory Committee:
Harvey Siy, Ph. D.
Sanjukta Bhowmick, Ph. D.
Robin Gandhi, Ph. D.
Victor Winter, Ph. D.**

UMI Number: 1554610

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 1554610

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Abstract

PROGRAM COMPREHENSION OF ASPECT-ORIENTED PROGRAMS

Jeffrey Steenbock, MS

University of Nebraska, 2014

Advisor: Harvey Siy, Ph. D.

The aim of aspect-oriented development has been to address the issue of software reuse outside the domain of established object-oriented techniques within the challenging realm of similar cross-cutting concerns. By decoupling the concerns from the core functionality, aspect-oriented developed software results in a smaller code base and reduced code duplication. This decoupling though presents new challenges to the software development process. The process of separating concerns impacts the developers established engineering inclinations as well as existing, established notations, such as UML, that developers are familiar with utilizing for both designing and understanding the implemented software systems. This thesis will study the impact of aspect-oriented software development on programmers' ability to comprehend the core system in addition to their comprehension of the aspect implementation.

“Each problem that I solved became a rule, which served afterwards to solve other problems.”

Rene Descartes

Acknowledgments

I wish to express my heartfelt gratitude to everyone who supported me with this thesis.

Thank you to my advisor, Dr. Harvey Siy. Your mentorship and guidance not only shaped this thesis into something we can all be proud of but also rekindled the thrill of discovery that life-long learning provides.

Thank you to the rest of my thesis committee, Dr. Sanjukta Bhowmick, Dr. Robin Gandhi, and Dr. Victor Winter. Your shared insights, suggestions, and challenging questions served to both support and motivate me throughout this research.

To my sons, William and Aaron, thank you for your patience and understanding during a process you probably do not yet fully understand. Someday the two of you will face similar challenges in your life and I hope that this endeavor will provide you with the same inspiration you provided me.

Finally, and most significantly, I would like to thank my wife Nicole. Without your support I could not have accomplished a fraction of what we achieve together. While only my name is on this thesis, in my heart your name is right next to my name, not only for this thesis but also for all the challenges we overcome together.

Contents

1. Introduction	1
1.1. Motivation	1
1.2. Problem Statement	5
2. Background and Related Work	6
2.1. Background	6
2.1.1. History	6
2.1.2. Aspect-Oriented Programming	12
2.1.3. Program Comprehension	13
2.2. Related Work	19
2.2.1. Aspect-oriented Development Effort	19
3. Definition and Validity Of Scope	25
4. Method	30
4.1. Applications	30
4.2. Tasks	32
4.2.1. First Task: Logging.	33
4.2.2. Second Task: Profiling.	35
4.2.3. Third Task: Null parameter checks.	37
4.2.4. Fourth Task: Field validation.	38
4.3. Participants	40
4.4. Measurements	41
4.5. Tools and Instrumentation	42
4.6. Procedure	43
5. Results	45
5.1. Subject Profile	45
5.2. Experiment Results	48
6. Discussion	52
6.1. Answer to Hypotheses	52
6.1.1. Hypothesis 1	52
6.1.2. Hypothesis 2	53
6.1.3. Hypothesis 3	54
6.1.4. Hypothesis 4	54
6.2. Improving Aspect-Oriented Modeling	55

6.3. Studying How People Organize Crosscutting Concerns	60
6.4. Threats To Validity	64
6.4.1. Internal Threats	64
6.4.2. External Threats	66
7. Conclusions	67
7.1. Contribution to Research	67
7.2. Implications to Practice	68
8. Future Work	69
8.1. Aspect-Oriented Refactoring	69
8.2. Aspect-Oriented Requirements Engineering	70
8.3. Aspect-Oriented and Verification	71
8.4. Aspect-Oriented Languages	72
A. CITI Completion Report	75
B. Consent	76
C. Survey	77
D. Detailed Application Metrics	79
E. Tasks	82
E.1. Paint Application	82
E.2. JHotDraw Application	84
F. Results Database Description	86
G. Experiment Results	87
H. Links	90
Bibliography	92

List of Figures

1.1. The observer aspect modeled using the AOSD profile	2
2.1. The Building Blocks of Software Engineering	7
2.2. Software Change Process	15
2.3. Norman's Stages of Execution Process Flow[Bos14]	17
4.1. Example log-invocation in Java	33
4.2. Example log-invocation in AspectJ	34
4.3. Example method profiler in Java	35
4.4. Example method profiler in AspectJ	36
4.5. Example null parameter check in Java	37
4.6. Example null parameter check in AspectJ	38
4.7. Example field validation in Java	39
4.8. Example field validation in AspectJ	39
5.1. Application task efforts (in seconds) by methodology	51
5.2. Clean application task efforts (in seconds) by methodology	51
6.1. Task effort with Hanenberg, et al Included(seconds)	58
A.1. CITI Completion Report	75
B.1. Consent Form	76
D.1. Paint Application Inheritance	79
D.2. JHotDraw Application Inheritance	80
D.3. JHotDraw Application Package Dependencies	81
D.4. JHotDraw Application Package Type Member Structure	81
F.1. Results Database Schema Definition	86

List of Tables

2.1. Principles of Software Engineering	8
2.2. Norman's Stages of Execution	16
2.3. Results from Hanenberg, et al	20
4.1. Target application metrics	31
5.1. Group Skill Assessment of object-oriented technologies	46
5.2. Group Skill Assessment non object-oriented technologies	47
5.3. Average task effort (in seconds) by methodology	48
5.4. Average participant effort (in seconds) by methodology	48
5.5. Average clean task effort (in seconds) by methodology	49
5.6. Paint task effort (in seconds) by methodology	50
5.7. JHotDraw task effort (in seconds) by methodology	50
5.8. Clean Paint task effort (in seconds) by methodology	50
5.9. Clean JHotDraw task effort (in seconds) by methodology	51
D.1. Detailed Target Application Metrics	80
G.1. Survey Responses	87
G.2. Paint Application Object-Oriented Participant Results	88
G.3. Paint Application Aspect-Oriented Participant Results	88
G.4. JHotDraw Application Object-Oriented Participant Results	89
G.5. JHotDraw Application Aspect-Oriented Participant Results	89

1. Introduction

1.1. Motivation

The objective of this thesis is to advance the understanding of how aspect-oriented programs are developed. Dissatisfaction with the current software engineering methodologies for modeling aspects instigated the interest in this research topic. Specifically, this dissatisfaction rises from a lack of intuitiveness with the nomenclatures currently utilized for modeling aspects as compared to the standard nomenclatures utilized in the Unified Modeling Language (UML) for modeling traditional object-oriented systems.

Fig. 1.1 from a survey on Aspect-Oriented modeling conducted by Wimmer, et al[WSK⁺11] depicts a typical implementation of modeling aspects. The initial deficiency encountered when modeling aspects utilizing this approach is the reuse of existing nomenclatures from object-oriented modeling to model

aspects. Utilizing this approach, developers model aspects with the same nomenclature used for specifying objects and classes, with only a profile indicator to differentiate the aspect from traditional objects or classes.

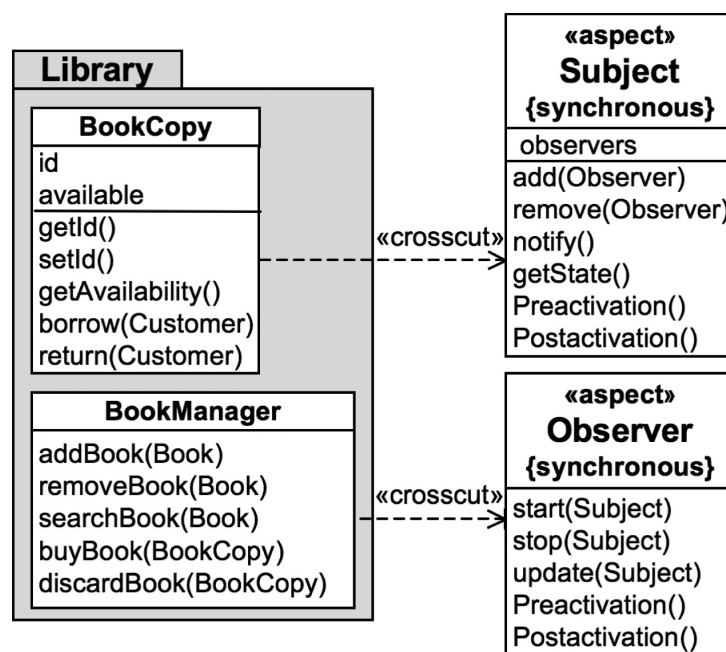


Figure 1.1.: The observer aspect modeled using the AOSD profile

More troubling with the current approaches for modeling aspects is the depiction of the application of the aspect to the core functionality. Referring again to Fig. 1.1 the application of the aspect to the core functionality is depicted reusing the relationship nomenclature of traditional object-oriented modeling with a profile indicator indicating the relationship is a cross cutting concern. What this approach fails to account for is 1) to scale applying an aspect to multiple, disparate core concerns, 2) having multiple disparate aspects applied to the same core concern, and 3) combinations of the previous applica-

tions. Wimmer, et al's survey noted that the surveyed methods typically only modeled aspects related to minimal number of core concerns and questioned the capability of the surveyed methods to model aspects relevant to multiple disparate targets. Additionally, this approach while perfectly suitable for depicting the hierarchical and owned relationships of object-oriented development fails to represent the less constrained relationships of decoupling an aspect from the core functionality accurately.

To arrive at a more suitable method for modeling aspect-oriented programs it is necessary to take a step back and approach the problem from the software developer's perspective. What is required is information on the developer's mental model of an aspect in relation to the objects and classes. This mental image of an aspect to object relationship may or may not align with the traditional object-oriented model nomenclatures. Thus, it is important to approach this problem from a fresh perspective without biasing possible solutions with preconceived notions of how software developers should model aspects based on the currently available and accepted methods.

At this point the issue is determining the process on how software developers fabricate the mental model of their software design. This fabrication of software design is constructed in a top-down approach. The developer forms a mental model of the solution based on their knowledge of the problem do-

main and their experience with similar problems and successful solutions they have applied in the past. This history of previously applied solutions is a developer's mental repository of design patterns. The method for building this mental repository of design patterns was accomplished through a bottom-up approach. At points in the developer's past they encountered problems in existing systems in which they had little to no domain knowledge or design patterns to draw upon. At this point it was necessary for the developer to traverse the relationships of the existing system to understand its structure and behavior. This newly gained knowledge is added to the developer's design pattern repository. Thus, before a developer can perform a top-down approach a developer must have performed a relevant bottom-up approach.

To advance the state of current modeling of aspect-oriented development it is necessary to understand how developers perform bottom-up comprehension of aspect-oriented programs. Only after understanding how developers traverse the relationships of aspect-oriented programs can it be determined how the mental models of aspect-oriented programs are internalized in the developer's mental design pattern repository. With this understanding of the developer's mental model of aspect-oriented programs, it should be possible to arrive at more suitable and intuitive nomenclatures for modeling aspect-oriented programs.

1.2. Problem Statement

The research's intent is to identify the impact aspect-oriented development methodologies have on the developer's ability to understand or comprehend software systems. The research approach to identifying the impact of aspect-oriented methodologies will be accomplished through measurement and analysis of effort that developers expend performing the evaluation stage as defined by Norman. This research will seek evidence indicating the challenges developers face in overcoming a "Gulf of Evaluation" of program comprehension of aspect-oriented systems. This evidence provides the support needed for future research to identify nomenclatures that are more effective at modeling aspect-oriented systems and more closely align with developer's mental models of separated concern capabilities.

2. Background and Related Work

2.1. Background

2.1.1. History

Referring to Fig. 2.1 Ross, et al [RGI75] identified the issues in effectively developing software programs through a software engineering process. The issues software engineering processes address were categorized into four fundamental goals: *modifiability*, *efficiency*, *reliability*, and *understandability*. Additionally, from Tab. 2.1 Ross, et al identified seven principles that affect the process of attaining the fundamental goals. These seven principles have driven the evolution of software languages from the early era of low-level machine language to the current state of high-level object-oriented languages.

The first step in the evolution of the programming languages was addressing the goal of understandability. The earliest forms of programs were de-

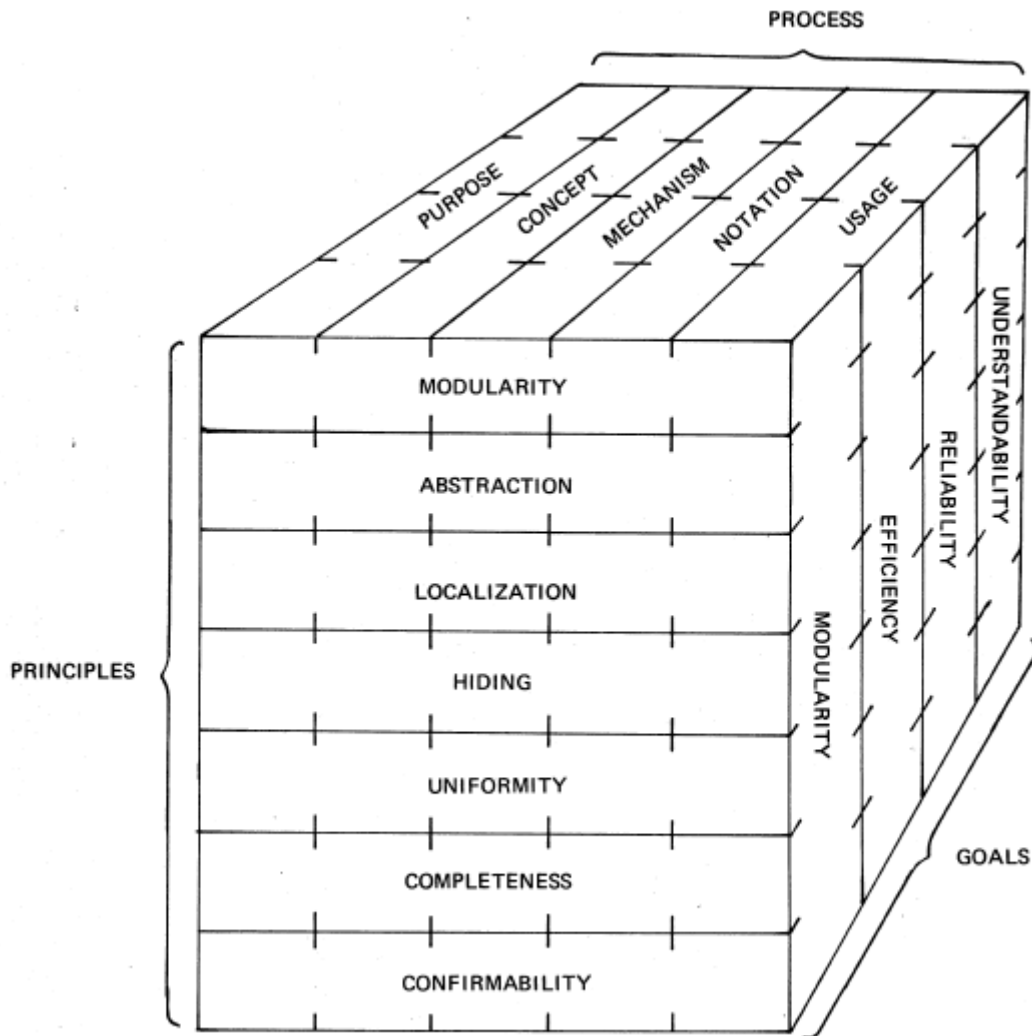


Figure 2.1.: The Building Blocks of Software Engineering

veloped in machine language, a set of operation codes coupled to the target systems instruction sets. While these operational codes accurately represent what the programs execution it is virtually impossible for a human reader to comprehend the intent of the code through examination. By the 1940's assembly language was introduced which introduced a mnemonic to represent the machine language's low level opcode. Assembly language applied the princi-

Principle	Definition
Modularity	Defines how to structure a software system appropriately
Abstraction	Identify essential properties common to superficially different entities
Hiding	Making inessential information <i>inaccessible</i>
Localization	Bringing related things together into physical proximity
Uniformity	Ensure consistency
Completeness	Ensure that nothing is left out
Confirmability	Ensure that information needed to verify correctness has been explicitly stated

Table 2.1.: Principles of Software Engineering

ple of abstraction by identifying the essence or intent of the opcode and presenting that information to the reader in comprehensible manner. Additionally, assembly language began the application of the principle of uniformity, through assembly language coupling the representation of the developed programs less tightly than machine language's coupling to the target system's operation codes.

While assembly language facilitated the ability of the programmer to understand what the program was performing at a single instruction it still failed in informing the programmer what the actual intent of the program as a whole was. With the introduction of FORTRAN and COBOL in the 1950s programmers were now able to comprehend the intent of the program. The introduction of these modern programming languages added an additional layer of abstraction which now isolated the programmer from the machine's

low level processing by representing a set of operations as a single statement representing the programmer's intent. Now, instead of storing memory in a machine's memory address the programmer defined a variable; and instead of instructing the machine to perform a processing operation on the memory address, the programmer defines an action to perform on the variable or set of variables.

Additionally, these languages applied additional principles to enhance the programs understandability. First was a continuation of the application of the principle of uniformity introduced with Assembly language. Representations of the programs developed in these modern languages had stricter adherence to a standard syntax which essentially severed any remaining coupling to the underlying target systems operational processing codes. This standardization of the syntax also applied the principle of modularity by providing a consistent structure for the program to conform.

The next stage of the program language evolution was the introduction of the C programming language in the early 1970s. The development of the C language was a continuation of the imperative language development in the ALGOL tradition, meant to address some of the perceived problems with the FORTRAN language. One of C's primary enhancements to program understandability was the application of the hiding principle through use of lexical

scoping of variables. C's lexical scoping of variables limited the scope of a variable to an individual block or function and made the existence of the variable invisible to code outside the block. This information hiding allowed the program to focus on only the relevant data for a specific function and reduced the cognitive load for understanding the behavior of the system.

C also addressed the goal of program efficiency by introducing the concept of pointers. Pointers provided the capability of dynamic memory allocation and facilitated the processing of large memory structures through manipulation of the pointer versus actual access to the underlying memory address. This increased program efficiency though it came at a cost to the programs understandability and reliability. The concept of accessing a variable through a pointer reference is a difficult concept for inexperienced programmers to grasp. Additionally, the careless use of pointers introduced program defects through the inadvertent access of an unintended memory access and memory leaks through improperly managing dynamically allocated memory.

By the 1980's and continuing through to present day program size and complexity became a primary factor in inhibiting understandability of the software systems. One method for addressing these issues was the introduction of object-oriented programming. Object-oriented programs applied another layer of the abstraction principle. Program code was no longer represented as

functions and variables that the computer operated on, but were instead represented as objects that matched a programmer's mental model of the problem domain. Additionally, the principle of locality was applied by grouping and encapsulating the related functions and state variables into the relevant object specification. This encapsulation also extended to the hiding principle by exposing the functions relevant to the external objects and hiding implementation details. To utilize the object, programmers need only understand what the object's intent is and do not need to know how the object accomplishes the intent.

While object-oriented programming was available with the C++ language in the early 1980's, the benefits were not fully realized until the introduction of Java in 1995. While C++ provided the benefits of object-oriented programming it suffered from the same deficiencies that hindered its predecessor C. C's deficiencies in dynamic memory allocation through pointers were exacerbated with the capability of dynamic object allocation. Java would come to be more fully embraced by addressing this issue. The problem of addressing dynamic object allocation can be seen as two-fold. First, the programmer fails to properly manage the cleanup of dynamically allocated objects because that code is not relevant to the core functionality that is being performed. Secondly, while a program may manage the code is one area of the program,

the memory management must be addressed in all areas where dynamic object allocation is performed. While object-oriented programs are capable of reusing modules in related objects, the act of dynamic memory management was difficult to apply because the capability cross-cut among all disparate, unrelated objects. Java was able to address both problems through the application of an automatic garbage collection capability. This garbage collection happened without programmer implementation, thus decluttering the core functionality and the program applying the garbage collection to all dynamic object allocations, regardless of the object relationships or intents.

2.1.2. Aspect-Oriented Programming

The evolution of software languages has not only improved the understandability of programs but has addressed the remaining three goals as well. One of object-oriented programming's largest contributions was the facilitation of modularity. Object-oriented programming applies the principle of modularity by introducing the capability of inheritance. Inheritance allows the programmer to write reusable functions or fields in one parent implementation and expose the capability to child implementations that extend the parent implementation.

While object-oriented programming greatly facilitated the ability to min-

imize the occurrences of implementing the same logic in multiple modules through inheritance, there remained a subset of capabilities that were difficult to implement in a modularized manner. Programming logic that defied the reuse through the object-oriented inheritance hierarchy are considered cross-cutting concerns due to the logic cutting across multiple abstractions. Tarr, et al [TOHS99], attribute this inability of object-oriented programming to modularize separation of concerns due to the object-oriented mechanism of only being able to support a single, dominant dimension. An example of the type of logic that spans across disparate objects of the dominant dimension is the management of dynamically allocated objects which the Java programming language was able to address through the garbage collection capability. Aspect-oriented programming addresses the implementation of these cross-cutting concerns by applying the principle of modularity to isolate and structure these concerns to a single implementation, termed an aspect[KLM⁺97].

2.1.3. Program Comprehension

From Biggerstaff, et al [BMW93], program comprehension or understanding is exhibited by the developer's ability to explain the program structure and behavior in terms of its relationship to the application domain. Additionally, this explanation must be expressed in terms that are qualitatively different

from the tokens and nomenclatures utilized in the source implementation. Essentially, program comprehension is the developer's action of reverse engineering a software capability represented for machine understandability to an internal mental representation within a human oriented context. This internal mental representation is referred to as a mental model [Nor02], an individual's interpretation and understanding of the structures that exist in the world.

To place program comprehension within the larger software development context it is necessary to see how it relates to the overall software change process. Fig.2.2 from Rajlich [Raj11] depicts the process software developers perform to modify a software system. Within this software change process, the Evaluation phase comprised of Concept Location and Impact Analysis entails program comprehension activities. During Concept Location, developers map the source implementation machine representation to their own mental model, mapping the program language tokens and nomenclature to their own interpretation of the application's structure and intent. Subsequently, the developer utilizes this mental model to form a plan of actions to accomplish the goals of the software change during Impact Analysis.

To assist in understanding how software developers conduct aspect-oriented development during these phases it is beneficial to map or relate these activ-

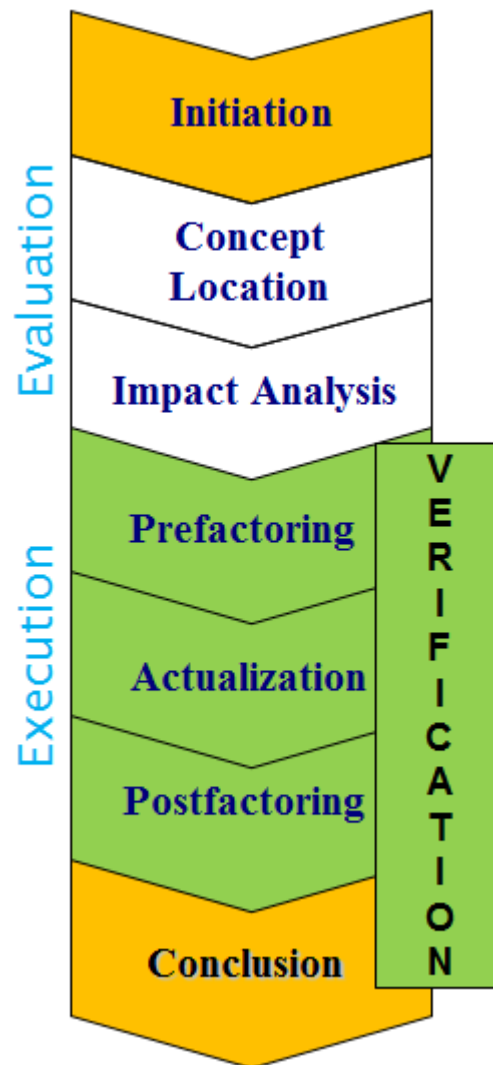


Figure 2.2.: Software Change Process

ities to stages in Norman's seven stages of action[Nor02], depicted in Tab.2.2 and Fig.2.3, and the difficulties Norman defines in performing the activities. Aspect-oriented development during the analysis and design phases are concerned with the developer's ability to recognize and choose the correct notation for developing the end system. In essence, this research should address the effectiveness a software developer has in forming the intention to

create software. The difficulty developers encounter in understanding aspect-oriented notation in the engineering process is defined as a “Gulf of Execution” according to Norman. This “Gulf of Evaluation” arises when the developer’s mental model of the system refined during the interpretation of perception does not align with the actual system structure that exists in the world. During maintenance activities the concern is to identify the effectiveness the notation exhibits in allowing the developer to assess the impact and emergent behaviors of the applied aspects. This research path should be concerned with the effectiveness in which developers are able to evaluate the developed systems end state. Maintenance activities are primarily concerned with aspect traceability.

Stage		Definition
Goals		The state to be achieved
Action	Intention to act	The action to achieve the goal
	Sequence of actions	The specific internal command steps
	Execution of the action sequence	Physical performance upon the world
Evaluation	Perceiving the state of the world	Sensing the current state of the world
	Interpreting the perception	Understanding the current state of the world
	Evaluation of interpretations	Comparing the interpretation to the goal

Table 2.2.: Norman’s Stages of Execution

To properly assess the validity of the reviewed research on aspect-oriented

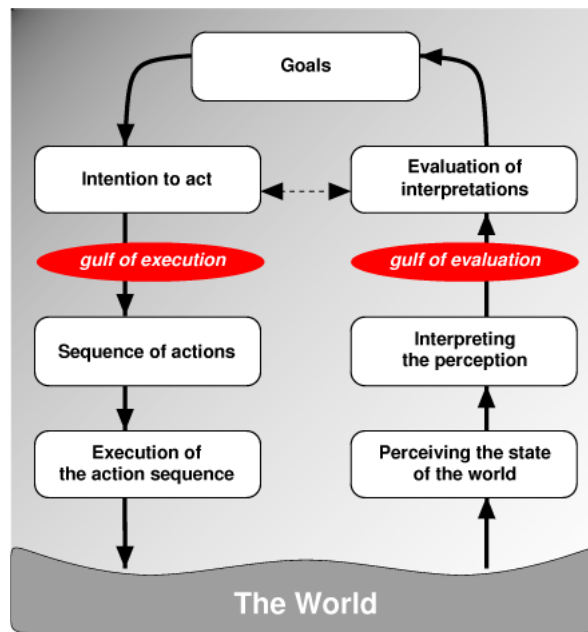


Figure 2.3.: Norman's Stages of Execution Process Flow[Bos14]

program comprehension, this thesis will place the research within the context of already established studies on program comprehension. Primarily, the research will be reviewed in relation to how aspect-oriented program comprehension aligns with the findings researched by von Mayrhauser and Vans[vMV97]. von Mayrhauser and Vans' findings indicate that software developers approach understanding of large scale software utilizing either a top-down approach, a bottom-up approach, or a combination of the two.

Top-down Program Comprehension

In the top-down approach, the developer creates a mental image of the pattern of the program structure as a hypothesis for understanding the program. This

mental image pattern is based on the developer's experience solving similar problems and discovered by applying knowledge of patterns held in long term memory in conjunction with domain knowledge of the system. The developer then tests the hypothetical mental image of the pattern during maintenance activities which either confirms and verifies the correctness of the hypothesized pattern or refutes the hypothesis which leads the developer to seek an alternate solution. Because the top-down approach relies on a solid understanding of multiple patterns and domain knowledge of the target software system, experienced or "expert" software developers are the usual practitioners of the top-down approach.

Bottom-up Program Comprehension

In the bottom-up approach, developers identify the potential function or entry point to where a maintenance defect exists. The developer then traces the flow of execution through the related methods and objects until eventually forming the entire chain of execution necessary to understanding the program structure relevant to solving the maintenance activity. The bottom-up approach is usually practiced by inexperienced developers or developers without adequate domain knowledge of the target software system. As new developers repeat application of the bottom-up approach to a software system, the developer

begins to store in long-term memory a collection of recognizable patterns, in addition to a stronger grasp of the target software system's domain. Eventually, as the developer builds their collection of patterns and domain knowledge, they move beyond the inexperienced developer stage to the experienced developer stage and begin applying the top-down approach to program comprehension.

2.2. Related Work

2.2.1. Aspect-oriented Development Effort

Hanenberg, et al [HKJW09] conducted a study to assess the impact of utilizing aspect-oriented programming versus object-oriented programming in development of crosscutting code. The aim of their research was to identify when, or even if, the utilization of aspect-oriented programming provided a positive impact on the time to develop a specific module of software. Tab.2.3 provides the measurement results from Hanenberg, et als' empirical experiment. While the large standard deviation prohibited Hanenberg, et al, from reaching a solid conclusion a number of interesting results were evident that would require future experimentation and research. Significantly, one finding was that tasks with less than thirty-six code targets demonstrated a signifi-

cant negative impact from utilizing aspect-oriented techniques. This finding on the negative impact of utilizing aspect-oriented techniques as well as Hanenberg, et als' experiment design will serve as a starting point, as well as an inspiration and template, for the design of this Thesis's empirical experiment.

Function	Task																	
	Logging		Parameter Null		Synchronized		Player Check		Notify Observers		Observers Null		Refresh Constraint		Label Value Check		Level Check	
	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO
Sum	77300	97271	17232	18707	15408	11558	9547	1371	25414	4467	11425	3360	8335	2937	37020	4071	18007	4636
max	9956	12269	5193	2065	1667	1443	1651	137	5442	731	2196	376	1074	367	4359	371	2622	986
min	595	2643	148	448	223	274	165	22	177	88	119	10	162	62	333	80	238	34
arith. mean	3865	4864	862	935	770	578	477	69	1271	223	571	168	417	147	1851	204	900	232
med	3793	4600	445	787	689	479	327	68	838	181	370	172	333	134	1513	212	775	184
std. dev.	2627	2356	1184	432	464	284	426	34	1251	151	494	97	226	73	1344	83	647	203
Sum _{ao} - Sum _{oo}	-19971		-1475		3850		8176		20947		8065		5398		32949		13371	
mean diff.	-999		-74		193		409		1047		403		270		1647		669	
med diff	-977		-328		122		270		596		246		243		1265		622	
Sum _{ao} / Sum _{oo}	79,47%		92,12%		133,31%		696,35%		568,93%		340,03%		283,79%		909,36%		388,42%	
# (diff < 0)	15		17		8		0		1		4		1		0		2	

Table 2.3.: Results from Hanenberg, et al

Hanenberg, et als' experiment utilized twenty subjects selected using a convenience sampling. The subjects were students drawn from the researchers' university and had completed five semesters or more of study. The participants entered the study with minimal to no experience with aspect-oriented programming development but received a 1.5 hour introduction to the AspectJ constructs necessary to perform the experiment prior to the experiment. The participants received no additional explicit training of object-oriented development or Java programming since all had completed and successfully passed Java programming courses. For the study the groups were divided into two

groups. Based on the results of a questionnaire, the development experience of both groups were similar. For the questionnaire, participants provided their own personal estimate of their development capabilities.

For Hanenberg, et als' experiment, the two study groups were tasked to write software modules to address crosscutting concerns in an existing application. The target application was a small game comprised of nine classes with 110 methods written in pure Java (version 1.6). The game architecture was based on a model-view-controller architecture with a small graphical user interface. The two groups were asked to complete nine tasks with one group performing the tasks first utilizing an object-oriented technique and then subsequently utilizing an aspect-oriented technique and the other group performing vice versa, first utilizing aspect-oriented techniques and then later using the object-oriented techniques. As developers worked to perform the tasks, the developers' IDE automatically logged the actions that modified the code base to a database. The research extracted snapshots at thirty-second intervals and then measured the time required to accomplish a specific task.

Hanenberg, et als' intent was to make a broad assessment on the development effort of aspect-oriented programming versus object-oriented programming. The measurement results from Hanenberg, et al, depict the total effort. In contrast, this thesis's intent is to provide a finer fidelity of the de-

velopment effort by measuring the evaluation and execution phases independently. Where Hanenberg, et als' findings indicate a negative impact of aspect-oriented programming on code targets less than thirty-six, this thesis aims to identify the root cause of that negative through isolating and measuring the impacts contributed by the evaluation and execution phases. As such, while the thesis utilizes Hanenberg, et als' experiment design as a template for its experiment design, there are significant differences introduced to facilitate the independent measurements of evaluation and execution phases.

Hanenberg, et als' experiment had the subjects implementing new crosscutting capabilities. In relation to program comprehension the implementation of a new capability is more closely aligned to the activities performed during top-down comprehension than bottom-up. Since top-down comprehension, as well as new capability implementation, requires developers to have acquired adequate domain knowledge on the existing application some form of bottom-up program comprehension must be performed beforehand to acquire that domain knowledge. In this sense, Hanenberg, et al, required their study subjects to perform some rudimentary bottom-up program comprehension before implementation of the new capability with the resulting measurements depicting the aggregate sum of both activities. In contrast, this thesis experiment's intent is to only measure program comprehension activities related

to crosscutting capabilities. To achieve this intent, the thesis experiment design will have the subjects performing a maintenance change to existing crosscutting capabilities. Utilizing this approach, the thesis experiment results will be able to measure the developers' establishment of the program mental-model more accurately and discriminate between the evaluation and execution phases the developer performs.

Finally, Hanenberg, et al, repeated the experiment with the same subject pool utilizing different methodologies but using the same application in both phases. Utilizing this approach, Hanenberg, et al, were able to eliminate the impact of different application complexity or size would have on their experiment result. The tradeoff utilizing this approach is that Hanenberg et als' approach is unable to account for the impact that acquisition of application domain knowledge has on the experiment's measurements. While Hanenberg, et al, were able to accept the variability introduced through application domain knowledge acquisition for their research, this thesis's intent to measure the effort of evaluation in program comprehension precludes utilizing this same approach. As a result, the design of this thesis experiment will require the subject pool to perform the experiment in two phases as well, utilizing different methodologies in both phases, but using a different application in the second phase. Utilizing this approach, this thesis experiment minimizes the

impact of acquisition of application domain knowledge to the measurement of bottom-up program comprehension but unfortunately introduces the unknown variability of how the different applications size and complexity may influence the compared measurement results.

Endrikat and Hanenberg [EH11] continued this line of research with an empirical experiment to measure the impact of development effort with aspect-oriented programming on repeated maintenance tasks. The design and execution of this experiment was similar to the previous experiment with the main difference being that subjects were tasked with performing multiple iterations of a maintenance task against a same separated concern capability. Conclusions from the Endrikat and Hanenberg indicate that the negative impacts encountered in their previous experiment due to aspect-oriented programming may be out-weighed by the positive impact that aspect-oriented programming contributes to repeated visits to an implemented aspect during maintenance activities.

3. Definition and Validity Of Scope

To quantify the objectives, the thesis defines the following properties:

$$e = v + x$$

where e is the total effort of a task with v being the effort to evaluate and x being the effort to execute. This thesis defines the properties within the specific applied development methodologies of object-oriented development and aspect-oriented development as:

$$o = ov + ox$$

$$a = av + ax$$

where o is the total effort of an object-oriented task with ov being the effort to evaluate the object-oriented task and ox being the effort to execute the object-oriented task and a is the total effort of an aspect-oriented task with av being the effort to evaluate the aspect-oriented task and ax being the effort to execute the aspect-oriented task. Within each methodology, the task is further categorized as being associated with either core functionality or cross-cutting concern functionality. For object-oriented methodology, the efforts for core and cross-cutting tasks are defined as:

$$o_c = ov_c + ox_c$$

$$o_a = ov_a + ox_a$$

with o_c being the effort for an object-oriented core functionality task and o_a being the effort for an object-oriented cross-cutting concern task. Utilizing an aspect-oriented methodology, the efforts for core and cross-cutting tasks are defined as:

$$a_c = av_c + ax_c$$

$$a_a = av_a + ax_c$$

The intent of this thesis is to address the following four hypotheses:

Hypothesis 1. *The effort of understanding core functionality in an aspect-oriented development software system is less than the effort of understanding the same core functionality utilizing an object-oriented methodology.*

$$av_c < ov_c$$

Aspect-oriented programming facilitates the bottom-up comprehension of the core functionality. Software developers will extend less effort in the evaluation and formation of a mental-model of the core functionality of an aspect-oriented development software system versus core functionality of a software system developed with traditional object-oriented methodology.

Hypothesis 2. *The effort of implementing core functionality in an aspect-oriented program is equal to or less than the effort of implementing the same core functionality utilizing an object-oriented methodology.*

$$ax_c \leq ox_c$$

Software developers will extend similar amount of effort in implementing the core functionality of an aspect-oriented development software system compared to implementing the core functionality of a software system developed with traditional object-oriented methodology.

Hypothesis 3. *The effort of understanding concern related functionality in an aspect-oriented development software system is greater than the effort of understanding the same concern functionality utilizing an object-oriented methodology.*

$$av_a > ov_a$$

Aspect-oriented programming inhibits the bottom-up comprehension of identifying concern functionality separated from the core functionality. Software developers will extend more effort in the evaluation and formation of a mental-model of the separated concern functionality of an aspect-oriented development versus the entangled concern functionality developed solely using object-oriented methodology.

Hypothesis 4. *Aspect-oriented programming reduces the effort to implement software concerns compared to object-oriented development methodologies.*

$$ax_a \leq ox_a$$

Software developers will extend less effort in implementing separated concern functionality of an aspect-oriented development versus duplicating entangled concern functionality using object-oriented methodology.

4. Method

To test the hypothesis an empirical experiment was conducted. The intent of the experiment was to determine a developers mental effort in forming the execution plan for tasks related to software aspects, where the software aspect is specified in a traditional object-oriented methodology or utilizing an aspect-oriented methodology. Therefore the experiment design requires participants to perform the same programming tasks using an object-oriented and an aspect-oriented language. The experiment utilizes Java as a representative of object-oriented language and the Java extension AspectJ as a representative of an aspect-oriented language.

4.1. Applications

For the experiment subjects were tasked to perform crosscutting maintenance against two target applications. The first application was a paint application

previously utilized by Ko, et al [KMCA06] for their study on reachability. The second application was the open-source project JHotDraw. The paint application is a relatively small application consisting of ten classes within three packages with sixty-one methods for a total of 434 lines of code. In contrast, the JHotDraw application is much larger and more complex consisting of 350 classes within eighteen packages with 3,253 methods for a total of 21,119 lines of code. Tab.4.1 provides a summary of the target applications metrics¹.

Metric	Paint	JHotDraw
Lines of Code	434	21,119
Packages	3	18
Classes	10	350
Methods	61	3,253
Function Complexity	1.2	1.6
Class Complexity	6.9	14.3
Total Complexity	69	4,997

Table 4.1.: Target application metrics

The experiment design implementation process placed each project under configuration management control utilizing a Git repository and established a baseline. After the projects were baselined, the design implementation branched each project into two different implementations, one implementation as a pure object-oriented Java (version 1.7) implementation and a second aspect-oriented implementation utilizing AspectJ (version 1.7). The experiment design then modified both implementation to require crosscutting main-

¹Metrics compiled using automated tool SonarQube

tenance tasks to be performed. The AspectJ version of both applications utilized the preferred technique of annotations to define aspects and join points versus the utilization of the AspectJ specific language nomenclatures.

4.2. Tasks

Using the two target applications, subjects need to perform four cross-cutting maintenance tasks utilizing either pure Java or AspectJ. This thesis designed the maintenance tasks to meet the following criteria:

- The maintenance tasks should be in the domain of crosscutting concerns for which AspectJ facilitates modularization.
- To measure the effort more accurately expended on the evaluation of crosscutting concerns, the task should minimize the effort required by execution by limiting the number of lines of code required for implementation to five or less.
- To ensure subjects comprehend the structure and relationship of the relevant targets of the task, the experiment design may need to obfuscate class and method names to prohibit the subject from completing the maintenance task through simple IDE provided searching capabilities.

Appendix E provides the full text of the tasks provided to the subjects.

4.2.1. First Task: Logging.

The first task requires the subject to modify an existing logging capability by changing the current logging level to a different level based on the logged method's scope. Fig. 4.1 provides an example of the original logging statement implemented in pure Java and Fig. 4.2 provides an example of the original logging aspect implemented in AspectJ.

```
class C {  
    private final static Logger LOGGER =  
        Logger.getLogger(C.class);  
    ...  
    public void m(int i) {  
        LOGGER.trace("Enter m()");  
        ...  
    }  
    private void n(int x) {  
        LOGGER.trace("Enter n()");  
        ...  
    }  
}
```

Figure 4.1.: Example log-invocation in Java

For the object-oriented methodology, the task requires the subjects to change the logging invocation on the first line of all non-public methods, in the example this is method `n()`, from `trace` to `debug`. For the aspect-oriented methodology, the task requires the subjects to change the logging invocation in the `logNonPublic()` method of the aspect implementation from `trace` to `debug`. For the paint application experiment, subjects can accomplish both the object-

```

@Aspect("pertypewithin(*)")
public class LoggingAspect {

    private Logger logger;
    @Before("staticinitialization(*)")
    public void init(JoinPoint.StaticPart jps) {
        logger = Logger.getLogger(

            jps.getSignature().getDeclaringType());
    }
    @Before("execution(public * *.*(..))")
    public void logPublic(JoinPoint jp) {
        logger.trace("Enter " +
            jp.getSignature().getName() +
            "()");
    }
    @Before("execution(!public * *.*(..))")
    public void logNonPublic(JoinPoint jp) {
        logger.trace("Enter " +
            jp.getSignature().getName() +
            "()");
    }
}

```

Figure 4.2.: Example log-invocation in AspectJ

oriented and aspect-oriented maintenance tasks with a one line code modification. For the JHotDraw application, subjects can accomplish the aspect-oriented maintenance task with a four-line code modification whereas the object-oriented subjects can accomplish the maintenance task with a one line code modification.

4.2.2. Second Task: Profiling.

The second task requires the subjects to modify an existing profiling capability that logged the time to execute a method by adding the profiling capability to other methods based on either the methods scope or implementing class. Fig. 4.3 provides an example of the original profiling capability implemented in pure Java and Fig. 4.4 provides an example of the original profiling aspect implemented in AspectJ.

```
class C {  
    private static final Profiler PROFILER =  
        Profiler.getProfiler(C.class);  
    public void m(int i) {  
  
        Calendar time = Calendar.getInstance();  
        ...  
        PROFILER.profileEnd("m", time);  
    }  
}
```

Figure 4.3.: Example method profiler in Java

For the object-oriented methodology the task requires the subjects to duplicate the profiling capability by adding the assignment of the method start time to a Calendar object at the beginning of the target method and calling the profileEnd() method before exiting the target method. For the aspect-oriented methodology the task requires the subjects to modify the join point defined in the @Around annotation in the profiling aspect to match the de-

```
@Aspect("pertypewithin(*.C)")
public class ProfileAspect {

    private Profiler profiler;
    @Before("staticinitialization(*)")
    public void init(JoinPoint.StaticPart jps) {
        profiler = Profiler.getProfiler(

            jps.getSignature().getDeclaringType());
    }
    @Around("* *.*(..)")
    public Object profileMethod(ProceedingJoinPoint jp)
        throws Throwable {
        Calendar time = Calendar.getInstance();
        Object retVal = jp.proceed();
        profiler.profileEnd(jp.getSignature().getName(),
            time);
        return retVal;
    }
}
```

Figure 4.4.: Example method profiler in AspectJ

sired target methods' signatures. For both the paint application experiment and JHotDraw application experiments, subjects can accomplish the aspect-oriented maintenance task with a single line code modification whereas the object-oriented subjects can accomplish the maintenance task with a three-line code modification.

4.2.3. Third Task: Null parameter checks.

The third task requires the subjects to modify an existing null parameter check capability that throws an exception when parameters assigned a null value are passed to a method by adding the null parameter check capability to other methods based on either the methods' name or passed in parameter types. Fig.4.5 provides an example of the original null parameter check capability implemented in pure Java and Fig.4.6 provides an example of the original null parameter check aspect implemented in AspectJ.

```
class C {
    public void m(Object v) {
        if (v == null) {

            throw new NullPointerException();
        }
        ...
    }
}
```

Figure 4.5.: Example null parameter check in Java

For the object-oriented methodology the task requires the subjects to duplicate the null parameter check by adding the check for null and exception throwing block at the beginning of the target method. For the aspect-oriented methodology the task requires the subjects to modify the join point defined in the @Before annotation in the null parameter check aspect to match the desired target methods' signatures. For the paint application experiment

```
@Aspect
public class NullCheckAspect {
    @Before("execution(* *.*(*) && args(v)")
    public void checkParm(Object v) {
        if (v == null) {

            throw new NullPointerException();
        }
    }
}
```

Figure 4.6.: Example null parameter check in AspectJ

subjects can accomplish the aspect-oriented maintenance task with a single line code modification whereas the object-oriented subjects can accomplish the maintenance task with a two line code modification. For the JHotDraw application experiment subjects can accomplish the aspect-oriented maintenance task with a four line code modification and the object-oriented maintenance task with a two line code modification.

4.2.4. Fourth Task: Field validation.

The fourth task requires the subjects to modify a field validation capability that checks that the value of a field assignment matches the fields legal values and reassigns the value to a default value if the assignment is out of bounds. Fig.4.7 provides an example of the original field validation implemented in pure Java and Fig.4.8 provides an example of the original field validation

aspect implemented in AspectJ.

```

class C {
    private int f;
    public void m(int v) {
        ...
        if (v < 0) {
            v = 0;
        }
        f = v;
        ...
    }
}

```

Figure 4.7.: Example field validation in Java

```

@Aspect
public class FieldValidateAspect {
    @Around("set(* int *) && args(v)")
    public void checkSet(int v, ProceedingJoinPoint jp)
        throws Throwable {
        if (v < 0) {
            v = 0;
        }
        jp.proceed(new Object[] {v});
    }
}

```

Figure 4.8.: Example field validation in AspectJ

For the object-oriented methodology the task requires the subjects to duplicate the field validation by adding the legal range check prior to field assignment and overriding the assignment to the default value if the assignment is out of range. For the aspect-oriented methodology the task requires the sub-

jects to modify the join point defined in the `@Around` annotation in the field validation aspect to match the desired target field signature. For the paint application experiment subjects can accomplish the aspect-oriented maintenance task with a five line code modification whereas the object-oriented subjects can accomplish the maintenance task with a two line code modification. For the JHotDraw application experiment subjects can accomplish the aspect-oriented maintenance task with a two line code modification and the object-oriented maintenance task with a four line code modification.

4.3. Participants

Nine subjects participated in the experiment. Subjects were selected from a pool of graduate students taking a course on advanced software-engineering topics and professed to having a requisite basic capability in programming with Java. The experiment was performed in two sessions due to scheduling conflicts with finding a common date with one of the subjects. The experiment divided the subjects into two groups with one group performing the experiment tasks utilizing an object-oriented methodology and later utilizing an aspect-oriented methodology and the other group vice versa.

To assure the protection of the human subjects participating in this research, the experiment followed the protocols and procedures established by

University of Nebraska Medical Center Institutional Review Board. Appendix A provides the report showing the completion of the primary investigator's required training from the Human Research Curriculum provided by the Collaborative Institutional Training Initiative. A copy of the approved protocol (reference ID # 642-13-EX) is available from the University of Nebraska Medical Center Institutional Review Board.

4.4. Measurements

The intention of the experiment is to identify the impact aspect-oriented development has on the effort developer's expend on evaluation in program comprehension. The challenge of this experiment is identifying the point at which the developer understands the program and moves on to the execution phase of implementing the change. While the experiment design cannot accurately identify the exact point at which developers make the transition from evaluation to execution, the experiment design does minimize the effort developers expend to perform the execution phase. Thus the experiment measures the total effort time expended on each task and assumes that the execution effort had minimal contribution to that total effort time. In order to perform the total measurement, the subjects were responsible for recording the time they started and completed each task from a provided digital clock.

The entire set of measurements taken for this thesis experiment are as follows. The incidental measurements are the demographic data collected from the subject survey used to identify the similarity and differences in the grouped subjects skill profile. The independent variables are the two projects with associated sizing and complexity, the utilized methodology, and the maintenance task type. The dependent variable is the subject-recorded time to complete the task.

To facilitate the compiling and analysis of the measurements, the measurements were loaded into a MySQL database. Appendix F provides the database design description used for relating the measurements for compiling and analyzing the results.

4.5. Tools and Instrumentation

The subjects performed the experiment on University of Nebraska-Omaha provided personnel computers. The hardware is a basic desktop personnel computer with a standard keyboard, mouse, and single monitor. The computers performance and memory were sufficient for execution of this experiment. Each machine was preloaded with the requisite software to perform the experiment. The preloaded software required for program execution was the Windows operating system, the Eclipse Integrated Development Envi-

ronment (Indigo release), Java Software Development Kit (version 1.7), the AspectJ library (version 1.7).

4.6. Procedure

Subjects performed the experiment in computer labs provided by the University of Nebraska-Omaha. Prior to starting the experiment, participants read and signed the consent form. Subjects then completed a short survey used to assess their skill level in the object-oriented and aspect-oriented development technologies relevant to the experiment. After the subjects completed the survey, the subjects received a basic thirty minute training tutorial on aspect-oriented programming. This training tutorial provides an overview of the concepts of software modularity that aspect-oriented programming seeks to address and the mechanisms, such as join points, that aspect-oriented programming provides to facilitate software modularization. The tutorial concludes with a short lab exercise in which participants are able to apply aspect-oriented programming concepts to a small Java program utilizing the tools and APIs utilized in the experiment.

For the experiment execution, the experiment divided the subjects into groups. Assignment to a group was done by randomly selecting subjects based on seating choice in the experiment lab. For the first phase of the experi-

ment both groups performed the previously defined maintenance tasks on the paint application but utilizing either the object-oriented or aspect-oriented methodology. For the second phase of the experiment, both groups performed the previously defined maintenance tasks on the JHotDraw application and switching methodology from the previous phase's utilized methodology.

5. Results

5.1. Subject Profile

Tab. G.1 in Appendix G provides the raw numbers from the subject self-assessment survey. Tab. 5.1 and Tab. 5.2 aggregate the results and provides a profile of the group the subject belonged to. This group profile is more beneficial in analyzing the impacts that subject similarity or differences may influence on the outcome of the experiment.

For future reference, Group 1 performed aspect-oriented methodology first on the Paint application and then object-oriented methodologies on the JHotDraw application. Conversely, Group 2 performed object-oriented methodologies on the Paint application and then aspect-oriented methodologies on the JHotDraw application.

From Tab. 5.1 both groups report a similar background and experience with

Function	1. Experience with OO development		2. Experience with UML		3. Experience with Java		4. Experience with other OO besides Java	
	1	2	1	2	1	2	1	2
max	4	5	4	3	5	4	3	5
min	3	3	3	2	4	3	3	3
arith. mean	3.80	3.75	3.20	2.75	4.20	3.25	3.00	4.00
med	4.00	3.50	3.00	3.00	4.00	3.00	3.00	4.00
std. dev.	0.40	0.83	0.40	0.43	0.40	0.43	0.00	0.72
mean diff	0.05		0.45		0.95		-1.00	
med diff	0.50		0.00		1.00		-1.00	

Table 5.1.: Group Skill Assessment of object-oriented technologies

object-oriented development and object-oriented modeling. Based on the subjects self-assessments, experience or lack of experience with object-oriented methodologies should not influence the experiment outcome as the reported median is above average and no subject reported having had no prior experience with object-oriented development. While Group 2 reports a lower skill level than Group 1 with Java programming they report a higher skill level with other object-oriented languages than Group 1 which should be an equalizing factor in the groups skill level. As with the object-oriented skill assessment, no subject reported having had no prior experience with the Java programming language.

From Tab.5.2 both groups report similar assessments on their background

Function	5. Experience with AOD		6. Experience with AspectJ		7. Experience with other AO besides AspectJ		8. Experience with Eclipse IDE		9. Experience with other IDEs	
	1	2	1	2	1	2	1	2	1	2
max	3	3	3	2	4	2	4	4	3	4
min	2	2	1	1	2	1	3	3	2	3
arith. mean	2.40	2.25	2.20	1.75	2.80	1.50	3.80	3.25	2.40	3.50
med	2.00	2.00	2.00	2.00	3.00	1.50	4.00	3.00	2.00	3.50
std. dev.	0.49	0.43	0.75	0.43	0.75	0.50	0.40	0.43	0.49	0.5
mean diff	-0.15		0.45		1.30		0.55		-1.10	
med diff	0.00		0.00		1.50		1.00		-1.50	

Table 5.2.: Group Skill Assessment non object-oriented technologies

with Aspect Oriented technologies including AspectJ with the reported median indicating little familiarity with the aspect-oriented development and related technologies. Group 1 reports a higher level of familiarity with the Eclipse IDE where it appears Group 2 is more familiar with other IDEs with no subject reporting having had no prior experience with working with the Eclipse IDE¹.

¹Note: Since both groups reported low familiarity with AspectJ it is probable that neither group has familiarity with the Eclipse AspectJ plugin.

5.2. Experiment Results

Tab.G.2 through Tab.G.5 in Appendix G provide the raw measurements on the subjects task completion effort times. Tab.5.3 through Tab.5.5 provide the initial aggregation of the raw measurements to begin detailed analysis. All three compiled measurements indicate similar findings. First, that subjects expended more effort on aspect-oriented maintenance tasks than object-oriented tasks in the first experiment phase for the paint application. Conversely, subjects expended more effort on the object-oriented maintenance tasks than the aspect-oriented tasks in the second experiment phase for the JHotDraw application. Finally, the subjects expended less effort in the second experiment phase for the JHotDraw application than the expended in the first phase for the paint application, regardless of the utilized methodology.

Experiment/Application	Aspect-Oriented	Object-Oriented
1. Paint application	684	606
2. JHotDraw application	411	519

Table 5.3.: Average task effort (in seconds) by methodology

Experiment/Application	Aspect-Oriented	Object-Oriented
1. Paint application	696	631
2. JHotDraw application	483	525

Table 5.4.: Average participant effort (in seconds) by methodology

Tab.5.6 and Tab.5.7 continue the analysis of the measurements by providing descriptive statistics of the measurements categorized by application

Experiment/Application	Aspect-Oriented	Object-Oriented
1. Paint application	633	505
2. JHotDraw application	379	490

Table 5.5.: Average clean task effort (in seconds) by methodology

Results from participants 3, 8, and 9 are excluded due to discrepancies in reported data

phase and maintenance task performed. An initial analysis from these measurements indicate largely varying values among all tasks types and methodologies as depicted by the wide range between minimum and maximum values and large standard deviations. Even with the large variance in results though, certain trends do emerge. First, for the paint application, the mean and median differences for each task fall within a narrow range, which is most evident after the initial logging task. Second, for the paint task the mean and median trend in the same down-ward direction for both methodologies. Third, the paint application's aspect and oriented tasks and the JHotDraw's aspect-oriented task appear to converge down-ward for each subsequent task to a common range, while the JHotDraw's object-oriented task trends upward for each subsequent task. Finally, for the initial logging task of each experiment application phase, the aspect-oriented means and medians are greater than the object-oriented means and medians. Fig. 5.1 provides a visual illustration depicting these trends.

Tab. 5.8 and Tab. 5.9, with the associated Fig. 5.2, illustrates the same mea-

Function	Logging		Profiling		Null Check		Field Validation	
	AO	OO	AO	OO	AO	OO	AO	OO
max	2940	1765	900	840	552	630	1105	457
min	405	325	140	254	420	120	108	180
arith. mean	1233	1035	587	617	498	397	393	295
med	793	1026	720	687	510	420	180	248
std. dev.	1019	510	324	222	53	193	414	118
mean diff	197		-30		101		98	
med diff	-233		33		91		-68	

Table 5.6.: Paint task effort (in seconds) by methodology

Function	Logging		Profiling		Null Check		Field Validation	
	AO	OO	AO	OO	AO	OO	AO	OO
max	865	381	710	1140	540	660	512	1020
min	172	175	120	180	120	392	420	546
arith. mean	507	252	328	502	308	577	471	803
med	495	255	153	389	265	634	480	824
std. dev.	324	72	271	331	174	103	38	177
mean diff	255		-174		-269		-333	
med diff	240		-263		-369		-344	

Table 5.7.: JHotDraw task effort (in seconds) by methodology

measurements as the previous tables and figures but with results from subjects who were unable to complete all tasks excluded. These results with excluded subject measurements indicate the same trends as the results including all subjects and support the same assertions made previously.

Task	Aspect-Oriented	Object-Oriented
1	1283	792
2	587	613
3	504	320
4	156	295

Table 5.8.: Clean Paint task effort (in seconds) by methodology

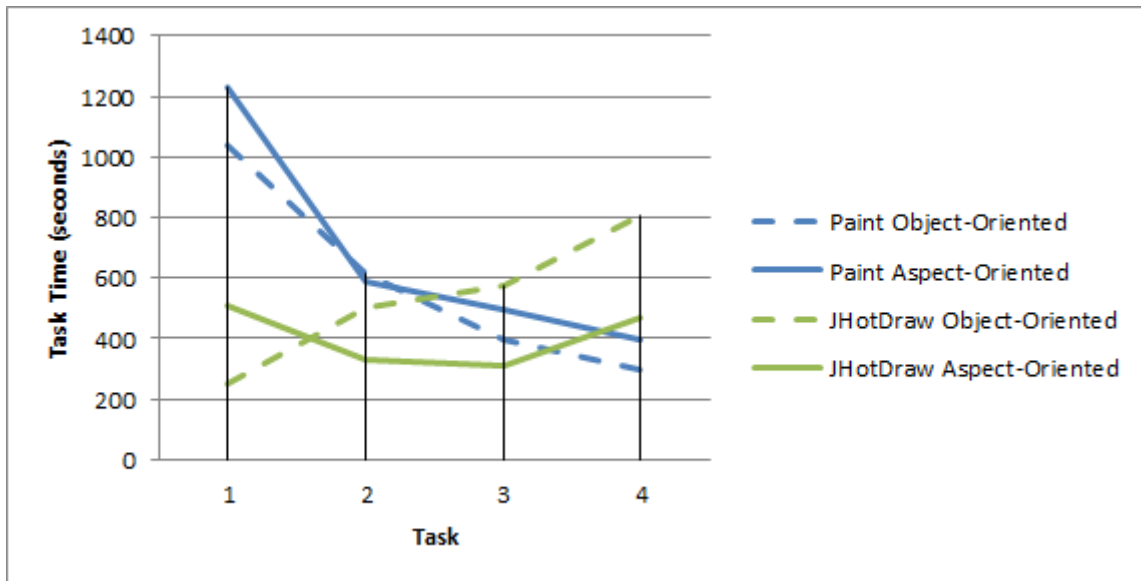


Figure 5.1.: Application task efforts (in seconds) by methodology

Task	Aspect-Oriented	Object-Oriented
1	411	270
2	328	396
3	308	562
4	471	731

Table 5.9.: Clean JHotDraw task effort (in seconds) by methodology

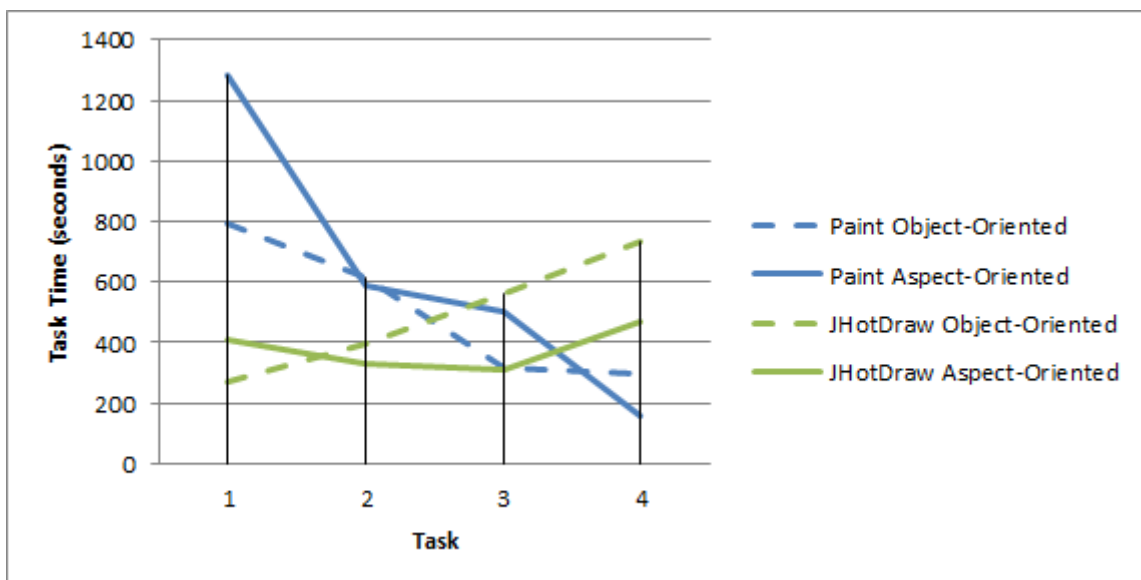


Figure 5.2.: Clean application task efforts (in seconds) by methodology

6. Discussion

6.1. Answer to Hypotheses

6.1.1. Hypothesis 1

The results of the experiment, specifically the measurements from the object-oriented tasks against the JHotDraw application, implicitly verify Hypothesis 1, that the utilization of an aspect-oriented methodology facilitates the understanding of core functionality. The results from the object-oriented tasks from the JHotDraw application indicate that as the complexity of object-oriented programs and the tasks increase, the effort to comprehend the program and tasks increases proportionality. Since an application of an aspect-oriented methodology to an object-oriented solution removes the reference to an entangled concern, the entangled concern's complexity would be eliminated from the core functionality. As a result, the remaining core functionality must be

less complex than the same functionality with the entangled concern. This reduction in complexity leads to a reduction in effort in performing program comprehension.

6.1.2. Hypothesis 2

The results of the experiment provided no evidence that either supported or contradicted the assertion of Hypothesis 2, that utilization of an aspect-oriented methodology facilitates the implementation of core functionality. While the experiment provided no explicit evidence relating to Hypothesis 2, it is a reasonable assumption that implementation of core functionality would not be negatively impacted utilizing an aspect-oriented methodology. Comparing the situation of implementing a core concern versus implementing a core concern with an entangled concern, the implementation effort of the pure core concern will be less than the implementation of the core concern with the entangled core concern. Intuitively, removing the implementation of the entangled core concern would result in the second case being equal in effort to the first case. Thus, utilization of an aspect-oriented methodology does not negatively impact the implementation of a core concern. Conversely, utilization of an aspect-oriented methodology will have no impact or a positive impact on the implementation of a core concern.

6.1.3. Hypothesis 3

The results of the experiment are inconclusive for Hypothesis 3, that the utilization of an aspect-oriented methodology negatively impacts the understanding of concern related functionality. The results from the first phase of the experiment utilizing the paint application initially verify the hypothesis but the results from second phase of the experiment utilizing the JHotDraw application contradict the assertion of the hypothesis. If analysis of the measurements is restricted to only the first task from both application phases the measurements would then verify the Hypothesis's assertion that aspect-oriented methodology impedes the developer's ability to comprehend concern related functionality. The fact that data from the subsequent tasks are inconclusive or contradict the hypothesis indicate an unanticipated phenomenon occurred. sec.6.2 provides further discussion and analysis of the measurements as they relate to Hypothesis 3 and potential solutions that address the rationale for the contradictory results.

6.1.4. Hypothesis 4

The results of the experiment, specifically the measurements from the JHotDraw application, support the assertion of Hypothesis 4, that utilization of an aspect-oriented methodology facilitates the implementation of concern related

functionality. The results from the JHotDraw phase of the experiment indicated less effort was expended utilizing the aspect-oriented methodology than utilizing a pure object-oriented methodology. Additionally, comparing the effort of implementing a single aspect versus the effort of implementing one or more entangled core concerns provides additional support to Hypothesis 4. The effort extended for implementation against a single aspect is constant in regard to the number of code targets, whereas the effort extended for implementation of multiple entangled concerns is proportional to the number of code targets. Thus, as the number of code targets increase for an entangled concern, eventually the implementation effort for entangled concerns will exceed the constant effort of implementation of a single aspect.

6.2. Improving Aspect-Oriented Modeling

Results from the experiment support Hypothesis 3 with the evidence indicating that developers will extend more effort in program comprehension of aspect functionality in an Aspect-Oriented program than in comprehending the same aspect functionality entangled in an Object-Oriented program. This is most evident when the efforts are compared for only the initial task that developers executed. For the Paint experiment, the Aspect-Oriented developers took an average 197 seconds longer to perform the first task than there

Object-Oriented counterparts and for the JHotDraw task the Aspect-Oriented developers took an average of 255 seconds longer than the Object-Oriented developers did.

Comparing the results of the first task from this experiment with results of the first task from Hanenberg, et al[HKJW09] provides evidence supporting Hypothesis 4. For both experiments, the first task was the implementation of a logging capability. In Hanenberg, et al the Object-Oriented developers took an average of 4864 seconds to complete the task, compared to 3865 seconds for the Aspect-Oriented developers. In Hanenberg, et al the Object-Oriented developers took an average of 999 seconds longer to perform the task than the Object-Oriented developers. This difference in results can be attributed to in Hanenberg, et al the Object-Oriented developers were required to implement the functionality in 110 code targets whereas this Thesis's experiment the Object-Oriented developers only need to implement the functionality in one code target. While not entirely precise or accurate, there is some benefit in taking Hanenberg, et als' results and averaging the numbers by number of code targets. With this approach, the Object-Oriented developers took an average of 486 seconds to implement each code target compared to the Aspect-Oriented developers' average of 3865 seconds for implementing a single logging aspect.

Results from tasks two through four did not support Hypothesis 3 but do indicate an unexpected phenomenon occurred. Of the six tasks executed after the initial logging task only two indicated additional effort in understanding concerns in an Aspect-Oriented program compared to an Object-Oriented program. While the results from tasks two through four do not support Hypothesis 3 they do indicate that a previously unanticipated phenomenon by the experiment hypotheses has occurred. Fig.5.1 provides the best indication of the phenomenon. Fig. 5.1 and the supporting data from Tab.5.6 and Tab.5.7 indicate potential trends related to three factors: 1) the previously acquired domain knowledge gained by the developer, 2) the complexity of the program, and 3) the methodology used. The trends indicated by this data point to a previously unidentified hypotheses concerning the developers effort in transitioning from bottom-up program comprehension to top-down program comprehension.

The first trend indicated by the data is the sharp decline in effort from the first task performed to all subsequent tasks. For the first task the Aspect-Oriented developers took an average of 1232 seconds and the Object-Oriented developers took an average of 1035 seconds. For all subsequent tasks the range of means for both Aspect-Oriented and Object-Oriented fell between 252 seconds and 803 seconds. Fig.6.1 includes data from Hanenberg, et al

which further indicates that a significant event occurs either during or after developers complete the initial experiment task. Data from both experiments suggest that developers have acquired adequate domain knowledge of the application utilized in the experiment to transition from performing bottom-up program comprehension to top-down program comprehension.

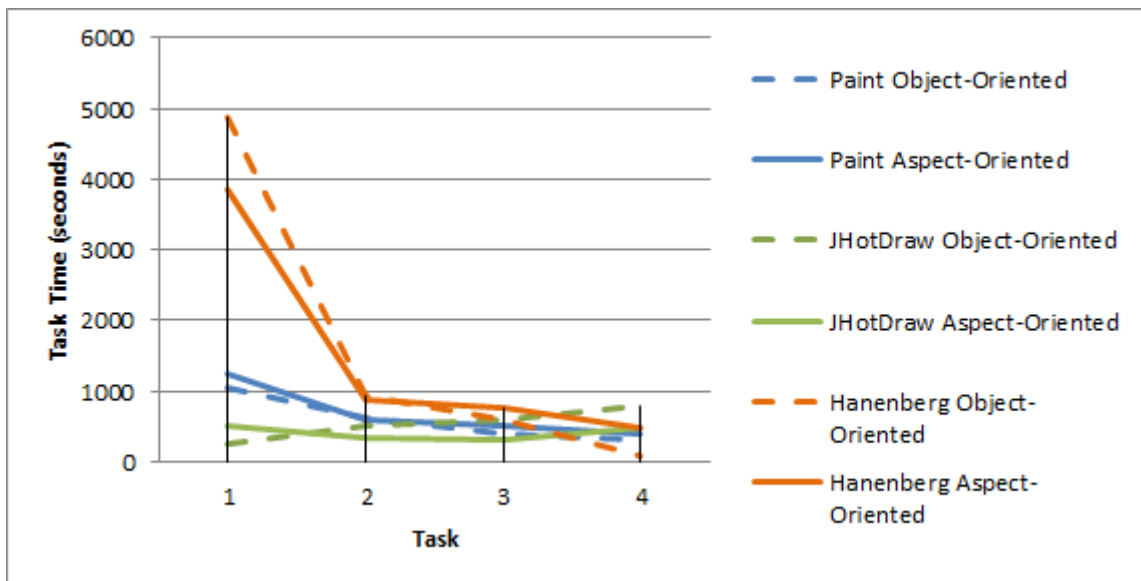


Figure 6.1.: Task effort with Hanenberg, et al Included(seconds)

The data from the experiment also indicates that a relationship exists between the complexity of the application, the methodology chosen for implemented concern capability, and the effort required to transition from bottom-up program comprehension to top-down program comprehension. For the Paint application the average effort time decreased for each subsequent task for both aspect-oriented and object-oriented approaches even though each subsequent task was designed to be more difficult than the previous task.

Additionally, the average task times for the Paint application decreased at similar rates and the difference between task times remained within a minimal range. Conversely, for the JHotDraw application the aspect-oriented development effort indicated an initial decline for the second task followed by a leveling off for the subsequent tasks while the object-oriented development showed an increase in effort for each subsequent task. A possible cause of this discrepancy can be attributed to the complexity of the application experiments. The Paint application contains a total of nine classes while the JHotDraw application contains a total of three hundred fifty classes. Based on the increased complexity of the JHotDraw application it can be reasoned that the object-oriented developers continued to perform bottom-up program comprehension for all tasks in the experiment, taking increased time for each subsequent task as the complexity of the task increased. This application complexity did not impact the aspect-oriented developers who transitioned after the first task to top-down program comprehension and were able to complete each subsequent task expending a relatively minimal constant amount of effort. For the Paint application, the complexity of the object-oriented implementation and the complexity of the aspect-oriented implementation were relatively similar resulting in the task times for each approach showing minimal differences between the two methodologies.

Hypothesis 5. *The effort to transition from bottom-up program comprehension to top-down program comprehension of concern related functionality in an aspect-oriented development software system is less than the effort to transition from bottom-up program comprehension to top-down program comprehension of the same concern related functionality utilizing an object-oriented methodology.*

This hypothesis assumes that the aspect functionality is relatively less complex and has fewer code operations than the systems core functionality.

6.3. Studying How People Organize Crosscutting

Concerns

Based on the post experiment formation of Hypothesis 5 future research should prove to be beneficial in exploring the benefits aspect-oriented development provide in facilitating the transition to top-down program comprehension. Based on the results of this thesis experiment the path forward should be to investigate better methods for modeling aspects than are currently available. Specifically, from the results it appears that comprehension of aspects can better be understood when viewed in isolation from the system's core func-

tionality. This approach conflicts with the current accepted practices of aspect modeling which tightly couples the aspect representation to the impacted core functionality. Future research in aspect modeling should focus on identifying a modeling nomenclature of aspects that is independent and decoupled from the object-oriented representation. While future research should base the modeling approach on familiar nomenclatures, the modeling approach does not necessarily need to re-utilize the current object-oriented nomenclatures such as UML.

While the results of this thesis indicate that developers are able to form mental models of aspect-oriented programs it did not discover what form that mental model takes. Future research should attempt to identify the aspect mental models developers and other system stakeholders form to understand the system. This can potentially be accomplished by taking the approach this thesis took and extending the experiment to include non-technical participants. One of the strengths of object-oriented analysis and design utilizing UML is a reuse of the nomenclature during the both analysis and design processes. By reusing the nomenclature, communication between the developers and the non-technical stakeholders of the system is greatly facilitated. This can only be accomplished if the nomenclature is capable of being understood by the non-technical stakeholders. Future aspect modeling nomenclatures

should attempt to duplicate this capability and future empirical studies of how subjects form mental models of aspects should include non-technical participants.

To illustrate how future research in aspect modeling can leverage the approach utilized in this experiment, this thesis provides a framework for a future aspect modeling experiment. For the potential aspect modeling experiment, consider a knowledge domain such as a hypothetical library. This library contains two rooms. In one room is a collection of books containing all known knowledge about the animal kingdom. In the second room is a collection of books containing all known knowledge about carpentry tools.

In the animal kingdom room, there is large number of shelves, with each shelf dedicated to collecting the books of a specific phylum. On the shelves are dividers which separate the books by the order classification. Each individual book in the animal library is dedicated to containing all known knowledge about an animal family. The chapters of these books are divided into a specific genus, with each genus chapter containing the knowledge for the individual species. In this manner, the library of animal knowledge is structured very similar to how an object-oriented software system is organized. Corollaries can be drawn, comparing the high level structure such as phylum to abstract classes utilized as the basis for further refined classes with a hierarchy of

inheritance ending at the species which corresponds to final concrete class implementations.

With the hypothetical domain defined, the experiment would require subjects to answer questions relevant to the domain. For instance, the experiment could task a subject to provide information on frogs. In this case the subject should navigate to the Chordata shelf and examine all books contained in the Anura divider. Or the experiment could ask the subject to provide information on scorpions with thick tails. Here the subject should navigate to the Arthropoda shelf and in the divider section for Scorpions find the book for Buthidae. Similarly, the subject could be asked to find information relevant to a specific layer of the animal kingdom hierarchy such as find animals with fur or hair, find animals with feathers, find animals with vertebrae, or find information on snails. Because this requested information is a specific point in the animal kingdom hierarchy, it is reasonable that the subject should be able to identify the knowledge location.

Now for the hypothetical experiment, suppose the subject is tasked with verifying the statement “Only animals with sharp teeth can eat meat”. The experiment should observe how the subject reasons or forms the intent to perform this task. Does the subject look for a specific phylum. Would the subject be able to identify all instances of animals with sharp teeth, i.e. did

the subject identify dogs; cats; humans (but not chimpanzees); some, but not all, snakes; sharks; piranhas; etc. Insight into how the subject reasons about performing this task will provide the evidence for development of more appropriate aspect modeling methodologies. In this sense, “Animals with sharp teeth” is the scattered concern to our system.

6.4. Threats To Validity

This section identifies the threats to the validity of this experiment which need to be explicitly communicated for software engineering empirical studies[KAKB⁺06].

6.4.1. Internal Threats

Due to scheduling constraints of the experiment participants, the experiment was performed in two different sessions. While the pre-experiment activities provided the subjects the same training material and tutorial exercise, the open discussion with different questions asked by the subjects in the different sessions introduces one threat to the validity of the experiment.

The experiment provided the tasks to the subjects in what the experiment designers considered simpler tasks first followed by subsequently more complex tasks. This approach was followed to reduce the risk of a subject becom-

ing too frustrated with a task and being unable to continue. A randomized approach to task ordering would have eliminated this threat but the small subject pool size and risk of subjects being unable to complete the task prevented the approach.

While the application and exact task implementation were different in both phases of the experiment, the task types were repeated in each phase. Subjects unfamiliar with the capability concept utilized during the first phase of the experiment (i.e. had no prior experience with logging or profiling) would have acquired that concept during the second phase introducing a reduction in the evaluation effort in the second phase. While the concepts of the maintenance task were not relatively complex to the overall task, any effort in understanding the concept for the first time would contribute to elevated development effort times in only the first experiment phase.

One final internal threat to validity of the experiment is the subjects presumption of proper methodology to utilize for task completion. After completion of the first task in the first experiment phase subjects knew which methodology to utilize for the completion of that experiment phase and which methodology they would utilize for the second phase. As a result, after the first task subjects knew whether the crosscutting concern was either entangled within the core implementation or was separated utilizing AspectJ. This

experiment does not address the issue of what the evaluation effort of identifying a crosscutting capability developers implement as an entangled object-oriented capability or as a potentially separated capability utilizing an aspect-oriented methodology.

6.4.2. External Threats

The external threats to the validity of the experiment arise from the subjects characteristics. The first characteristic is that all subjects were graduate students taking a course on advanced software engineering methodologies. Beyond that commonality the background of the subjects ranged from full-time to part-time graduate students with varying degrees of professional software engineering experience. The experiment is unable to assess the impact that the subjects' prior experiences with crosscutting domain capabilities that they potentially encountered through either academic or professional pursuits influenced the measurement results. Secondly, the psychological profile of the individual subject influences their capability to complete the experiment tasks. This experiment provided no mechanism for identifying how the subjects' intelligence, learning aptitude, and personality preferences influenced their ability to compete the experiment tasks.

7. Conclusions

7.1. Contribution to Research

This thesis contributes in identifying the impacts aspect-oriented methodologies have on developers' capabilities to comprehend software systems. The findings from this research imply that the aspect-oriented methodologies of decoupling separate concerns from the core capability impedes the developer's ability to perform bottom-up program comprehension, the primary intent of the empirical experiment conducted. A secondary finding not initially hypothesized is the positive impact aspect-oriented methodologies have on the developers' effort in transitioning from bottom-up comprehension to top-down comprehension.

7.2. Implications to Practice

Based on the findings from this research, software engineering practitioners utilizing aspect-oriented methodologies would be advised to design the separate concern capabilities independently and without undue bias or influence from the core capability. While the design of aspects in this manner may entail performing bottom-up comprehension with resulting negative impact to effort, the resulting packaging and structure of the resultant system design should facilitate comprehension to future maintainers through reducing the effort to transition to top-down comprehension. Note that this approach conflicts with many of the current approaches to modeling aspects through utilization of UML profiles that couples the aspect to the targeted core capabilities. Practitioners that follow this proposed approach must be cognizant of the fact that while they achieve benefit to understandability and modularity of the system design there is a risk of introducing defects by ignoring the principle of completeness.

8. Future Work

In addition to the future research path outlined in sec.6.2 the findings from this research can also be leveraged against other facets of ongoing research related to aspect-oriented development. This section identifies these tangential research topics on aspect-oriented development and describes how these research topics can utilize the findings from this thesis.

8.1. Aspect-Oriented Refactoring

A potential challenge developers utilizing traditional object-oriented methodologies face in comprehension of separated concern functionality is recognizing that the entangled concern is in fact a candidate for encapsulation utilizing aspect-oriented programming. In many cases this realization that an entangled concern has been replicated throughout the system does not occur until late in the development life cycle after the functionality has been

implemented multiple times. Aspect-oriented refactoring is a technique for improving modularity and reducing complexity of these existing systems not utilizing aspect-oriented development through a methodological modification of the system to an aspect-oriented system[YSY⁺11]. If developers can practice these aspect-oriented refactoring techniques early in the development life-cycle as the cross-cutting entanglement begins to emerge, not only can duplicate effort of future entangled concern implementation be avoided but identification of these entangled concerns will facilitate developer comprehension of aspect components of their developing system. Potentially this early identification of aspects will provide the mental model and design patterns necessary for developers to approach system design with a background necessary to encapsulating cross-cutting concern capabilities.

8.2. Aspect-Oriented Requirements Engineering

As previously discussed in sec.6.3, modeling of aspects during the analysis and design phases should utilize the same nomenclature as much as possible to ensure understandability by both technical and non-technical stakeholders. Chitchyan, et al [CGS⁺09], identify a challenge in aspect composition during requirements analysis that all stakeholders encounter that lead to the extensibility and usefulness of the implemented aspect. The major-

ity of aspect-oriented developments, including the techniques utilized in this research, compose the aspects utilizing syntactic references. These syntactic compositions utilize references to the core module or wild-card mechanisms to define the point-cut expressions which fail to express the actual meaning and intent of the aspect and lead to the problem of point cut fragility. Chitchyan, et al, propose a semantic composition technique based on natural language analysis that facilitates the understanding of aspects through utilization of a more expressive, human-oriented nomenclature than the prevailing syntactic approaches. Potentially, Chitchyan, et als', semantic composition aligns more closely to stakeholders mental model of cross-cutting concerns than current aspect-oriented modeling nomenclatures and lead to a reduction of effort in the "Gulf of Evaluation". Future research on extending this semantic composition through the aspect-oriented development life cycle and the impact this approach has on program comprehension could prove to be beneficial.

8.3. Aspect-Oriented and Verification

Krishnamurthi and Fisler[KF07] outline the unique challenges that aspect-oriented development has on the verification of the resulting systems. As with representation of all systems, the more engineers utilize the principles of abstraction in depicting the system the greater the degree of falsifiabil-

ity is introduced into the resulting representation [Pop72]. Essentially, these challenges from aspect-oriented representations arise from the inherent decoupling of the aspect from the core capability and the complexity of recombining the advice with the core capability to perform verification. This challenge is further exacerbated if the approach recommended in this thesis are followed which advocate for developers to model and develop aspects independently from the core capabilities. In addition, when the problem of aspect verification is researched in connection with the problems of point cut fragility described by Chitchyan, et al, the potential emerges that future modifications to the core capabilities may result in unintended system behaviors. As such, any benefits from research in early life-cycle aspect-oriented processes must be analyzed for the impact that utilization of the techniques have on the verification process. Ideally, research in aspect-oriented verification identifies substantial benefits that supplant any impacts early life cycle aspect-oriented research may impose on the verification effort.

8.4. Aspect-Oriented Languages

Additionally, future research can apply the recommendations from the previous sections in improving the languages that implement aspect-oriented programs. In one sense the AspectJ programming language can be compared

to the C++ programming language in that both languages are extensions to an existing programming language. Just as developers were slow to accept object-oriented development until the advent of a pure object-oriented programming language in Java, developers may be unwilling to perform aspect-oriented development because of the effort of integrating AspectJ with the ongoing Java development. The creation of a single, integrated aspect-oriented programming language that combines the features of aspect-oriented and object-oriented programming may facilitate the acceptance of aspect-oriented programming. The creation of this pure aspect-oriented programming language must be cognizant of the deficiencies identified by this research and the cited research topics. Ideally, a pure aspect-oriented language would address the issues of point cut fragility and program comprehension through utilization of semantic compositions as proposed by Chityan, et al. At a minimum, improvements to aspect-oriented languages can utilize the findings from this research and treat aspects as separate, decoupled entities by providing a separate, localized packaging structure unique to the aspect-implementation and independent from the class definitions. This aspect only package structure would force developers to design aspects independently from the core implementation and facilitate bottom-up program comprehension by providing the anchor point from which maintainers can begin their concept search of sepa-

rate concern functionality.

A. CITI Completion Report

**COLLABORATIVE INSTITUTIONAL TRAINING INITIATIVE (CITI)
HUMAN RESEARCH CURRICULUM COMPLETION REPORT**
Printed on 09/28/2013

LEARNER Jeffrey Steenbock (ID: 3700400)
DEPARTMENT UNO - Computer Science
PHONE (402) 882-4726
EMAIL jsteenbock@unomaha.edu
INSTITUTION University of Nebraska Medical Center (UNMC/UNO)
EXPIRATION DATE 08/31/2016

GROUP 6 - SOCIAL/BEHAVIORAL FOR STUDENTS

COURSE/STAGE: Basic Course/1
PASSED ON: 09/01/2013
REFERENCE ID: 11134097


REQUIRED MODULES	DATE COMPLETED	SCORE
Introduction	08/31/13	No Quiz
Students in Research	09/01/13	10/10 (100%)
History and Ethical Principles - SBE	09/01/13	5/5 (100%)
Defining Research with Human Subjects - SBE	09/01/13	5/5 (100%)
The Regulations - SBE	09/01/13	5/5 (100%)
Assessing Risk - SBE	09/01/13	5/5 (100%)
Informed Consent - SBE	09/01/13	4/5 (80%)
Privacy and Confidentiality - SBE	09/01/13	5/5 (100%)
Research with Prisoners - SBE	09/01/13	4/4 (100%)
Research with Children - SBE	09/01/13	2/4 (50%)
Research in Public Elementary and Secondary Schools - SBE	09/01/13	4/4 (100%)
International Research - SBE	09/01/13	3/3 (100%)
Internet Research - SBE	09/01/13	4/5 (80%)
Research and HIPAA Privacy Protections	09/01/13	4/5 (80%)
Conflicts of Interest in Research Involving Human Subjects	09/01/13	5/5 (100%)
University of Nebraska Medical Center (UNMC & UNO)	09/01/13	No Quiz

For this Completion Report to be valid, the learner listed above must be affiliated with a CITI Program participating institution or be a paid Independent Learner. Falsified information and unauthorized use of the CITI Program course site is unethical, and may be considered research misconduct by your institution.

Paul Braunschweiger Ph.D.
Professor, University of Miami
Director Office of Research Education
CITI Program Course Coordinator

Figure A.1.: CITI Completion Report

B. Consent



Page 1 of 1

IRB PROTOCOL # 642-13-EX

NARRATIVE CONSENT

Title of this Research Study
Empirical experiment on Aspect-Oriented Program Comprehension

You are invited to participate in this research study because of your enrollment in a post-graduate computer science course concerning software engineering topics. The information in this consent form is provided to help you make an informed decision whether or not to participate. If you have any questions, please ask.

If you agree to take part in this study, you will complete a short survey concerning your level of experience with different software engineering techniques. Then you will receive a brief tutorial on software engineering techniques necessary to perform the experiment. Finally, you will be asked to perform basic software maintenance activities utilizing different software engineering paradigms.

Participating in this research is voluntary, and you may decide not to participate in this study. Your decision about participating will have no influence on your grade for the software engineering course from which you were recruited for this study.

By signing this document, you are saying that the information on this consent form has been explained to you, that you have read and understood the consent form, that your questions have been answered, and that you have decided to participate. If you have any questions, please inform the investigator leading the research study.

Signature of Subject _____ Date _____

Signature of Person Obtaining Consent _____ Date _____

Authorized Study Personnel:
Principal Investigator: Jeffrey Steenbock - jsteenbock@unomaha.edu
Faculty Advisor: Harvey Siy, Ph.D. - hsiy@unomaha.edu

Version 1
Subject's Initials

IRB Approved
Valid until 10/23/2018

Figure B.1.: Consent Form

C. Survey

The scale of 1 through 5

1. Having no experience
2. Have heard of the technology but have never actively used the technology.
3. Occasionally used the technology.
4. Regularly uses the technology.
5. Expert level use of the technology. Would feel comfortable mentoring others on the use of the technology.

The survey questions

1. Experience with Object-Oriented development
2. Experience with the Unified Modeling Language (UML).

3. Experience with the Java programming language
4. Experience with Object-Oriented programming languages besides Java.
(C++,C#,etc).
5. Experience with Aspect-Oriented development.
6. Experience with AspectJ
7. Experience with other Aspect-Oriented technologies besides AspectJ (JEE
Interceptors, Spring, etc).
8. Experience with using the Eclipse IDE.
9. Experience with other IDEs besides Eclipse.

Metrics	Paint	JHotDraw
Packages	7	18
Lines of Code	785	41,051
Compilation Units	14	290
Concrete Classes	11	280
Abstract Classes	2	22
Interfaces	1	48
Enums	0	0
Static Fields	1	163
Instance Fields	44	522
Static Methods	3	0
Instance Methods	65	2,651
Constructors	7	357
Static Initialization Blocks	0	4
Static Initialization Block LOC	0	22
Instance Initialization Blocks	0	1
Instance Initialization Block LOC	0	8
Method LOC Average	5	7
Method LOC Std. Deviation	5	9
Method Statement Average	2	3
Method Statement Std. Deviation	3	5

Table D.1.: Detailed Target Application Metrics



Figure D.2.: JHotDraw Application Inheritance

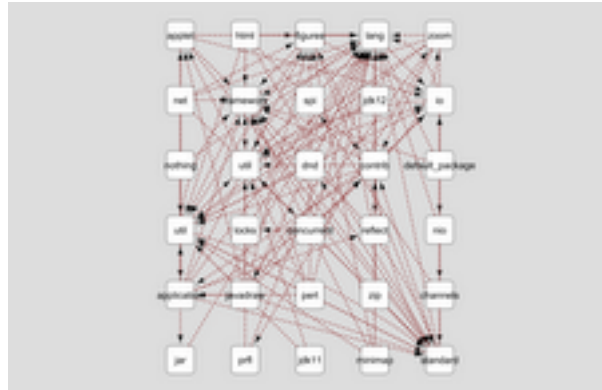


Figure D.3.: JHotDraw Application Package Dependencies

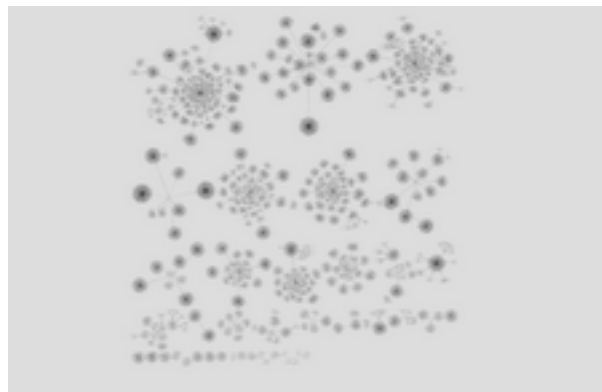


Figure D.4.: JHotDraw Application Package Type Member Structure

E. Tasks

E.1. Paint Application

1. The Application currently writes a log statement when entering every method at the *trace* level. Modify the application to write the log statement for *non-public* scoped methods at the *debug* level.
 - a) Aspect solution: `GgngPct::lgNnPblcNtr()` - 1 line mod
 - b) Object-oriented solution:
`PaintObjectConstructor::makeHoveringPrototype()` - 1 line mod
2. The Application currently profiles the methods in the `PaintWindow` class (i.e. the time to execute the method is recorded). Using the same technique as applied to the `PaintWindow` class, profile the `EraserPaint` class.
 - a) Aspect solution: `RflrSpct` Aspect annotation. - 1 line mod

b) Object-oriented solution: EraserPaint - 3 new lines

3. The Application currently checks for null arguments being passed into the setter methods and if the argument is null throws a `NullPointerException`. Using the same technique used for the setter methods, add a check for null arguments in methods with the name *define* and if the argument is null throw a `NullPointerException`.

a) Aspect solution: `SttrLdtr::hchPrm()` - 1 line mod

b) Object-oriented solution: `PencilPaint::define()` - 2 new lines

4. The application currently checks if public `int` fields are being assigned a value less than zero. If the value being assigned is less than zero the field is instead assigned the value of zero. Using the same technique as checking *int* fields, ensure that the assignment of *public double* fields is also greater than or equal to zero.

a) Aspect solution: `SttrLdtr` new aspect similar to `chkSet()` - 5 new lines.

b) Object-oriented solution: `PaintWindow::PaintWindow()` 1-2 mod or new lines

E.2. JHotDraw Application

1. The Application currently writes a log statement when entering a *public* method in the `org.jhotdraw.samples.net` package at the *debug* level. Modify the application to write the log statement for *protected* scoped methods in the `org.jhotdraw.samples.net.package` at the *trace* level.
 - a) Aspect solution: Add advise for protected methods in the `GgngPct`.
- 4 lines.
 - b) Object Oriented solution: `NetApp::createTools()` - 1 line.
2. The Application currently profiles the *public* methods in the `org.jhotdraw.contrib.zoom` package that have a `MouseEvent` as a passed in parameter (i.e. the time to execute the method is recorded). Using the same technique as applied to the public methods in the package, profile the *protected* and *private* methods in the `org.jhotdraw.contrib.zoom` package that have a `MouseEvent` parameter passed in.
 - a) Aspect solution: Remove the public scope from join point in `PrflPct::profile()` - 1 line modified.
 - b) Object oriented solution: `ZoomDrawingView::createScaledEvent()`.
3. The Application currently checks for null arguments being passed into setter methods of parameter type `Font` and if the argument is null throws

a `IllegalArgumentException`. Using the same technique used for the `Font` setter methods, add a check for null arguments in setter methods with parameter type of `StorageFormat` and if the argument is null throw an `IllegalArgumentException`.

a) Aspect solution: Add advice similar to `VldtrSpct::chchPrm()` but using `StorageFormat` as parameter. - 4 new lines.

b) Object Oriented solution: Add check in

`StorageFormatManager::setDefaultStorageFormat()` - 2 new lines

4. Field validation. The application currently checks to ensure the int fields `fOriginX`, `fOriginY`, `fWidth`, and `fHeight` in the `TextFigure` class are assigned positive values. If the value being assigned is negative the fields are instead assigned a zero value. Using the same technique as the check for `TextFigure` ensure that the assignment of fields `fLastX` and `fLastY` are also only assigned values greater than or equal to zero

F. Results Database Description

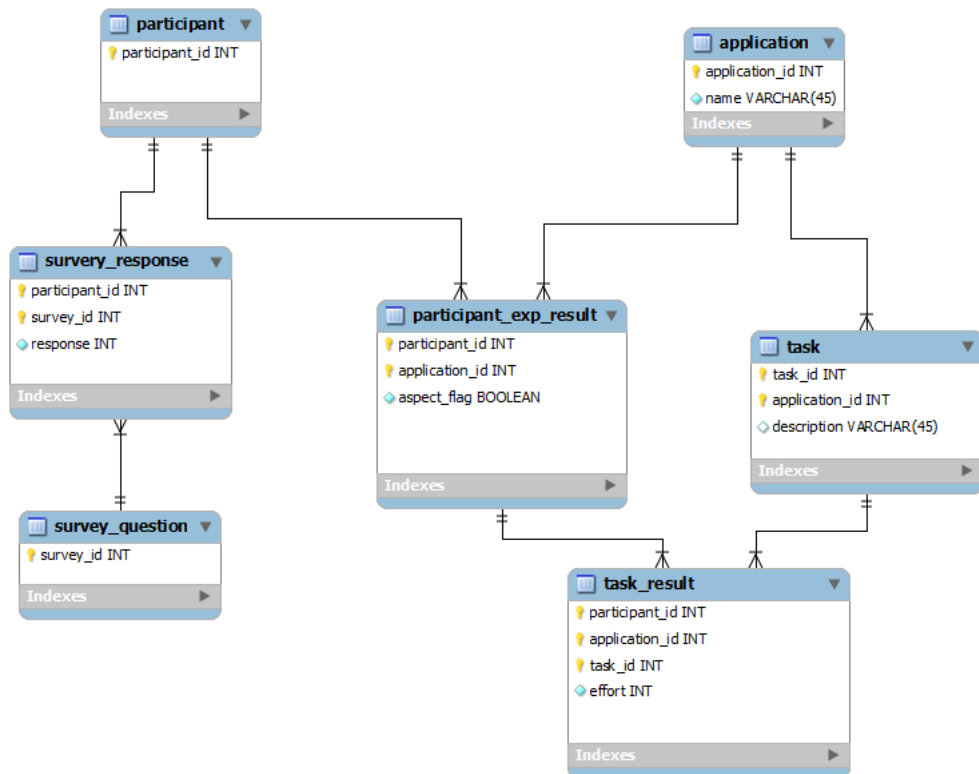


Figure F.1.: Results Database Schema Definition

G. Experiment Results

Participant Identifier	1. Experience with OO development	2. Experience with UML	3. Experience with Java	4. Experience with other OO languages	5. Experience with AO development	6. Experience with AspectJ	7. Experience with other AO technologies	8. Experience with Eclipse	9. Experience with other IDEs
1	4	3	4	3	3	3	4	4	2
2	3	3	3	4	3	2	2	3	4
3	4	4	5	3	3	3	3	4	3
4	4	3	4	4	2	2	2	4	3
5	4	3	4	3	2	1	2	4	2
6	5	3	3	5	2	1	1	3	4
7	4	3	4	3	2	2	3	4	3
8	3	2	3	3	2	2	1	3	3
9	3	3	4	3	2	2	2	3	2

Table G.1.: Survey Responses

Participant Identifier	1. Logging	2. Profiling	3. Null Check	4. Validator
2	985	744	511	457
4	325	840	120	180
6	1066	254	328	248
8	1765	630	630	DNF

Table G.2.: Paint Application Object-Oriented Participant Results

Participant Identifier	1. Logging	2. Profiling	3. Null Check	4. Validator
1	505	900	540	240
3	1080	DNF	480	1105
5	405	140	552	108
7	2940	720	420	120

Table G.3.: Paint Application Aspect-Oriented Participant Results

Note: Participant 3 did not perform the tasks in the assigned order. Participant 3 performed the tasks in order of task 4, task 3, then task 1, and did not finish with task 2.

Note: Participant 9 participated in the Aspect-Oriented Paint application experiment but did not record the task times.

Participant Identifier	1. Logging	2. Profiling	3. Null Check	4. Validator
1	255	360	660	900
3	195	1140	540	DNF
5	175	440	634	546
7	381	389	392	747
9	255	180	660	1020

Table G.4.: JHotDraw Application Object-Oriented Participant Results

Participant Identifier	1. Logging	2. Profiling	3. Null Check	4. Validator
2	865	710	540	420
4	195	120	120	480
6	172	153	265	512
8	795	DNF	DNF	DNF

Table G.5.: JHotDraw Application Aspect-Oriented Participant Results

H. Links

- University of Nebraska Medical Center Institutional Review Board - <http://www.unmc.edu/irb>
- Collaborative Institutional Training Initiative - <http://www.citiprogram.org>
- Paint from Carnegie Mellon - <http://www.cs.edu/~marmalade/studies.html>
- JHotDraw - <http://jhotdraw.org>
- Git - <http://www.git-scm.com>
- SonarQube - <http://www.sonarqube.org>
- Eclipse IDE - <http://eclipse.org>
- AspectJ - <http://www.eclipse.org/aspectj>
- MySQL - <http://www.mysql.com>

- Sextant - http://faculty.ist.unomaha.edu/winter/ShiftLab/Sextant_web/Sextant_index.html

Bibliography

- [BMW93] T.J. Biggerstaff, B.G. Mitbender, and D. Webster. The concept assignment problem in program understanding. In *Software Engineering, 1993. Proceedings., 15th International Conference on*, pages 482–498, May 1993.
- [Bos14] B. Bos. Design guide, April 2014. <http://www.w3.org/People/Bos/DesignGuide/simplicity.html>.
- [CGS⁺09] Ruzanna Chitchyan, Phil Greenwood, Americo Sampaio, Awais Rashid, Alessandro Garcia, and Lyrene Fernandes da Silva. Semantic vs. syntactic compositions in aspect-oriented requirements engineering: an empirical study. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 149–160. ACM, 2009.

-
- [EH11] Stefan Endrikat and Stefan Hanenberg. Is aspect-oriented programming a rewarding investment into future code changes? a socio-technical study on development and maintenance time. *International Conference on Program Comprehension*, 0:51–60, 2011.
- [HKJW09] Stefan Hanenberg, Sebastian Kleinschmager, and Manuel Josupeit-Walter. Does aspect-oriented programming increase the development speed for crosscutting code? an empirical study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 156–167, Washington, DC, USA, 2009. IEEE Computer Society.
- [KAKB⁺06] Barbara Kitchenham, Hiyam Al-Khilidar, Muhammad Ali Babar, Mike Berry, Karl Cox, Jacky Keung, Felicia Kurniawati, Mark Staples, He Zhang, and Liming Zhu. Evaluating guidelines for empirical software engineering studies. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 38–47, New York, NY, USA, 2006. ACM.
- [KF07] Shriram Krishnamurthi and Kathi Fisler. Foundations of incre-

-
- mental aspect model-checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2):7, 2007.
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97 - Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.
- [KMCA06] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, December 2006.
- [Nor02] Donald A. Norman. *The Design of Everyday Things*. Basic Books, New York, reprint paperback edition, 2002.
- [Pop72] Karl Raimund Popper. *Objective knowledge: An evolutionary approach*. Clarendon Press Oxford, 1972.
- [Raj11] Václav Rajlich. *Software Engineering: The Current Practice*. CRC Press, 2011.
- [RGI75] D.T. Ross, J.B. Goodenough, and C. A. Irvine. Software engineering: Process, principles, and goals. *Computer*, 8(5):17–27, 1975.

-
- [TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering, ICSE '99*, pages 107–119, New York, NY, USA, 1999. ACM.
- [vMV97] Anneliese von Mayrhauser and A. Marie Vans. Program understanding behavior during debugging of large scale software. In *Papers presented at the seventh workshop on Empirical studies of programmers, ESP '97*, pages 157–179, New York, NY, USA, 1997. ACM.
- [WRG13] Victor Winter, Carl Reinke, and Jonathan Guerrero. Sextant: A tool to specify and visualize software metrics for java source-code. In *Emerging Trends in Software Metrics (WETSoM), 2013 4th International Workshop on*, pages 49–55. IEEE, 2013.
- [WSK⁺11] Manuel Wimmer, Andrea Schauerhuber, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger, and Elizabeth Kapsammer. A survey on uml-based aspect-oriented design modeling. *ACM Comput. Surv.*, 43(4):28:1–28:33, October 2011.
- [YSY⁺11] Reishi Yokomori, Harvey Siy, Norihiro Yoshida, Masami Noro,

and Katsuro Inoue. Measuring the effects of aspect-oriented refactoring on component relationships: two case studies. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 215–226. ACM, 2011.