The appropriate separation of concerns is a fundamental engineering principle. A concern, for software developers, is that which must be represented by code in a program; by extension, separation of concerns is the ability to represent a single concern in a single appropriate programming language construct. Advanced separation of concerns is a relatively recent technique in software development for dealing with the complexity of systems that contain crosscutting concerns, namely those individual concerns that cut across programs. Aspect-oriented programming (AOP), which is the area of this dissertation, offers a form of advanced separation of concerns in which primary and crosscutting concerns can be separated during problem solving. An aspect gathers into one place a concern that is or would otherwise be scattered throughout an object-oriented program or system. The primary aim of this dissertation—the AOPy project—is to investigate the usefulness of advanced separation of concerns that aspect-oriented programming offers. In other words, the AOPy Project determines whether the potential usefulness of aspect-oriented programming is currently actualized in practice. In determining its current practical usefulness, this dissertation also determines characteristics of and obstacles
to usefulness of aspect-orientation in software development. Perhaps the most important contribution to understanding and addressing the problem of complexity in software systems that this dissertation makes is that the AOPy research project establishes a definition of compatibility of aspect-orientation and provides an analysis of sample instances during problem solving that indicate evidence of compatibility between object-orientation and aspect-orientation. Compatibility, as defined by the AOPy Project, exists when aspect-oriented ideas, terminology, and techniques are appropriately employed in the experimental problem-solving session. The primary scientific contribution of this dissertation, therefore, is a narrative description of the actual use of aspect-oriented programming in a series of controlled, problem-solving scenarios. Theories describing the use of aspect-oriented ideas, terminology, and techniques are generated and refined by means of Grounded Theory, a qualitative data analysis technique. Because this dissertation 1) analytically explores areas of compatibility of aspect-orientation with object-orientation and 2) defines areas of compatibility thwarted in practice, this research project can serve as a foundation for the development of aspect-oriented programming-based design methodologies that encourage compatibility and discourage non-compatibility. Therefore, the AOPy Project establishes a foundation for future research in both its methodology and its results and for future software development in practice. By contributing a definition of aspect-oriented compatibility and a framework within which it can be understood, this dissertation fosters the progression toward a seamless use of aspect-orientation between developer and task.
Advanced Separation of Concerns and the Compatibility of Aspect-Orientation

by

Douglas R. Dechow

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented March 18, 2005
Commencement June 2005

APPROVED:

Redacted for Privacy
Major Professor, representing Computer Science

Redacted for Privacy
Associate Director of the School of Electrical Engineering & Computer Science

Redacted for Privacy
Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Redacted for Privacy
Douglas R. Dechow, Author
ACKNOWLEDGEMENTS

The author expresses sincere appreciation to those individuals and institutions that supported work on this dissertation. The author's dissertation director, Dr. Timothy Budd, allowed students in his graduate course to perform the main experiment for this research. Dr. Budd oversaw and participated in the reading group that, in the course of its work, responded to this dissertation's chapter drafts; gratefulness for their responses is probably most evident in the revisions made to the dissertation. Dr. Budd also provided helpful suggestions as the dissertation took shape and is heartily thanked for that contribution.

The guidance of the entire dissertation committee helped shape this dissertation. Dr. Paul Cull, Dr. Jonathan Herlocker, Dr. Prasad Tadepalli, and Dr. Bella Bose contributed a variety of responses and suggestions for which the author is grateful.

With the help of Prof. John Dooley, whose enthusiasm was greatly appreciated, Knox College allowed the author to teach a course on aspect-oriented programming; the students in that course performed the preliminary experiment.

The author is especially grateful to those readers who provided both important feedback and encouragement throughout the research and writing of this dissertation: Eric Helvey, Dethe Elza, and Dr. Anna Leahy. Dr. Leahy's editing skills are much appreciated.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>1 Introduction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Rationale</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Experimental Methodology and Design</td>
<td>14</td>
</tr>
<tr>
<td>1.4 Case Study: Observer Design Pattern</td>
<td>17</td>
</tr>
<tr>
<td>1.5 Experiment Overview</td>
<td>18</td>
</tr>
<tr>
<td>1.6 Conclusion</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 Literature Review</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Introduction</td>
<td>23</td>
</tr>
<tr>
<td>2.2 Separation of Concerns</td>
<td>24</td>
</tr>
<tr>
<td>2.3 Introduction to Aspect-Oriented Programming</td>
<td>27</td>
</tr>
<tr>
<td>2.4 The Law of Demeter</td>
<td>30</td>
</tr>
<tr>
<td>2.5 Adaptive Programming</td>
<td>31</td>
</tr>
<tr>
<td>2.6 Subject-Oriented Programming</td>
<td>34</td>
</tr>
<tr>
<td>2.7 Dimensionality and Separation of Concerns</td>
<td>39</td>
</tr>
<tr>
<td>2.8 MultiDimensional Separation of Concerns</td>
<td>40</td>
</tr>
<tr>
<td>2.9 Hyperspaces</td>
<td>41</td>
</tr>
<tr>
<td>2.10 More Aspect-Oriented Programming</td>
<td>45</td>
</tr>
<tr>
<td>2.11 Adaptive Programming vs. Aspect-Oriented Programming</td>
<td>45</td>
</tr>
<tr>
<td>2.12 Hyperspaces vs. Aspect-Oriented Programming</td>
<td>46</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.13 Aspect-Oriented Programming &amp; Design Patterns</td>
<td>49</td>
</tr>
<tr>
<td>2.14 Conclusions</td>
<td>49</td>
</tr>
<tr>
<td>3 Materials and Methods</td>
<td>51</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>51</td>
</tr>
<tr>
<td>3.2 Support for Aspect-Orientation</td>
<td>51</td>
</tr>
<tr>
<td>3.3 The Weaver</td>
<td>63</td>
</tr>
<tr>
<td>3.4 Completing the Weave</td>
<td>66</td>
</tr>
<tr>
<td>3.5 Experiment Design</td>
<td>66</td>
</tr>
<tr>
<td>3.6 Experiment Methodology: Grounded Theory &amp; Verbal Reports</td>
<td>78</td>
</tr>
<tr>
<td>4 Case Study: Observer Design Pattern</td>
<td>83</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>83</td>
</tr>
<tr>
<td>4.2 Observer Design Pattern</td>
<td>85</td>
</tr>
<tr>
<td>4.3 The Observer Design Pattern Using Aspect-Orientated Techniques</td>
<td>97</td>
</tr>
<tr>
<td>4.4 Discussion</td>
<td>106</td>
</tr>
<tr>
<td>4.5 Related Work</td>
<td>106</td>
</tr>
<tr>
<td>4.6 Conclusion</td>
<td>107</td>
</tr>
<tr>
<td>5 Experiment Results</td>
<td>108</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>108</td>
</tr>
<tr>
<td>5.2 Framework for Experiment Results</td>
<td>116</td>
</tr>
<tr>
<td>5.3 Experiment Results</td>
<td>122</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.4 Summary of Experiment Results</td>
<td>130</td>
</tr>
<tr>
<td>6 Case Study: Analysis of Experiment Results</td>
<td>131</td>
</tr>
<tr>
<td>6.1 Analytical Framework for Experiment Results</td>
<td>131</td>
</tr>
<tr>
<td>6.2 Categorical Framework for Analysis of Experiment Results</td>
<td>138</td>
</tr>
<tr>
<td>6.3 Application of Compatibility Framework to Mention Sequences</td>
<td>141</td>
</tr>
<tr>
<td>6.4 Summary of Analysis</td>
<td>149</td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>156</td>
</tr>
<tr>
<td>7.1 Overview</td>
<td>156</td>
</tr>
<tr>
<td>7.2 The Usefulness of Aspect-Oriented Programming</td>
<td>158</td>
</tr>
<tr>
<td>7.3 The Future of Aspect-Orientation</td>
<td>160</td>
</tr>
<tr>
<td>7.4 Conclusion</td>
<td>166</td>
</tr>
<tr>
<td>Bibliography</td>
<td>168</td>
</tr>
<tr>
<td>Appendices</td>
<td>181</td>
</tr>
<tr>
<td>Appendix A: CS 180 Context for Subject Backgrounds</td>
<td>182</td>
</tr>
<tr>
<td>Appendix B: CS 582 Context for Subject Backgrounds</td>
<td>194</td>
</tr>
<tr>
<td>Appendix C: CS 180 Experiment Problems</td>
<td>200</td>
</tr>
<tr>
<td>Appendix D: CS 582 Experiment Workshop Syllabus</td>
<td>212</td>
</tr>
<tr>
<td>Appendix E: CS 582 Experiment Problems</td>
<td>219</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

2.1 Class dictionary graph .................................................. 32
3.1 Interceptor object ....................................................... 61
4.1 Observer Class Diagram .................................................. 86
5.1 Data Collection Process ................................................... 108

LIST OF TABLES

3.1 Subject Education and Experience .................................. 68
4.1 GoF-to-Python Method Name Mapping ................................ 87
5.1 Subject education and experience .................................. 115

LIST OF LISTINGS

2.1 Customer Account demonstrating tangled concerns ............... 30
2.2 Shipping subject ........................................................ 36
2.3 Shipping subject’s subject specification file ....................... 36
2.4 Transportation subject .................................................. 37
3.1 An AOPy join point declaration ...................................... 55
3.2 Join points for TraceAspect.py ....................................... 56
3.3 Python reflection: changing an object’s class .................. 59
3.4 The AOPy weave() method ............................................. 63
4.1 Module Observer.py ..................................................... 91
4.2 Point and Screen example using the Observer pattern .......... 96
4.3 Aspect-oriented ObserverProtocol in Python ................... 100
LIST OF LISTINGS (Continued)

Page

4.4 Point example using AOP Observer pattern .................. 102
4.5 Screen example using AOP Observer pattern .................. 103
4.6 Using the ObserverAspect in an example .................... 104
1.1 Overview

This dissertation documents the AOPy research project conducted by Douglas R. Dechow. This research project explores the usefulness of aspect-oriented programming in practice.

The AOPy project uses a qualitative research methodology to understand aspect-oriented programming within a frame of reference—the developers' skills and backgrounds and the problem domain—that replicates the professional development environment. The methodology in the AOPy project involved the following general steps:

1. identifying a setting and problem
2. establishing a theoretical framework through a literature review
3. choosing data collection methods
4. designing the experiment and defining the research project as a process in which the researcher can learn and adapt the experimental questions and the experiment design
5. considering how these steps can be documented for others.
In this way, the AOPy research project fit into the larger context of qualitative research more generally and asserts valid conclusions that can be incorporated into future research and into software development practice.

1.2 Rationale

In order to understand the reasoning behind the AOPy project, one must consider the increase in software complexity and the development of this project's focus on aspect-oriented programming.

1.2.1 Complexity: A Software Crisis

In the late 1960's, the NATO Science Committee held two conferences on the subject of software engineering. The proceedings of these conferences were published as *Software Engineering* (Naur and Randell, 1968) and *Software Engineering Techniques* (Randell and Buxton, 1969). One of the driving factors for convening the conferences was the perception that software engineering was facing a software crisis. In his 1972 Turing Award Lecture, Edsger Dijkstra summarized his view of the software crisis with the following statement:

the major cause [of the software crisis] is [...] that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.
Although Dijkstra’s position dates from the era of mainframe computers, this quote resonates with the developer of a modern distributed, network-enabled, n-tiered software system. At the heart of Dijkstra’s assertion is the notion of the complexity of creating software systems.

The kind of software design and development complexity that Dijkstra describes is notoriously difficult to define. The authors Alan Shalloway and James Trott (2002) give the following advice: “Good design requires keeping the big picture in mind.” And yet, most software is created in an act of narrow focus—one statement at a time. The disconnect that results in the mind of the developer is the source of many software design and development problems.

In the era that Dijkstra’s quote addresses, several notable software engineering attempts were made to address the problem of software complexity. At the 1968 NATO conference, Doug McIlroy presented his vision of mass-produced software in the form of software components (McIlroy, 1968). In 1972, David Parnas’s paper “On the Criteria To Be Used in Decomposing Systems into Modules” presented modular programming and “modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time.” This work was a way to understand the so-called big picture by understanding some of the complexity of the whole by addressing parts or modules that, presumably, complement each other to form the whole.

Still, as the requirements for our software systems increase, complexity continues to be an increasingly important problem that must be addressed. A
requirement is a capability that the system must deliver (Leffingwell and Widrig, 2000). Requirements result from the process of analysis of the real-world problem and are established before the software is built. According to Kiczales, et al (1997), in a software system, concerns are “the important design decisions the program must implement.” As was stated earlier, each of these design decisions, or concerns, must be translated into code that satisfies the requirement. This notion can be restated as the following: a concern is anything that can be traced back to a requirement.

Software systems can grossly be divided into primary concerns and crosscutting concerns, which are defined by Kiczales et al. (1997). Primary concerns are variously known as core concerns (Laddad, 2002) and basic concerns (Hursch and Lopes, 1995). Crosscutting concerns are variously called non-core special-purpose concerns (Hursch and Lopes, 1995) and system-level concerns (Laddad, 2002a). For the purpose of this dissertation, the two types of concerns are primary and crosscutting.

An example will help to clarify this terminology. In current software development thinking, the principle of separation of concerns is used in order to address the problem of complexity. Czarnecki and Eisnecker say the following about the principle separation of concerns:

The principle acknowledges that we cannot deal with many issues at once, but rather with one at a time. It also states that important issues should be represented in programs intentionally (i.e., explicitly, declaratively, and with little or no “extra noise”) and should be well localized. (2000)
Using the example of a banking system, a requirement might be something like the need for all users to use a password to log into an account. In the same system, Laddad (2002a) lists the following primary concerns that need to be addressed: customer and account management, interest computation, interbank transactions, statement generation, customer care, etc. This same banking system may also have concerns related to transaction management, persistence, synchronization, authentication, security, etc. Because these latter concerns touch many of the modules in a system, they are said to cut across, or crosscut, the system and are, therefore, termed crosscutting concerns.

Advanced separation of concerns is a relatively recent progression in separation of concerns techniques for dealing with the complexity of systems that contain crosscutting concerns. (Hursh and Lopes, 1995) Aspect-oriented programming (AOP), which is the area of this dissertation, offers a form of advanced separation of concerns in which primary and crosscutting concerns can be separated during problem solving. In the aspect-oriented programming model, primary concerns are typically represented as components (Kiczales et al, 1997; Czarnecki and Eisenecker, 2000). These components are generally understood to be objects. Exact definitions of objects range from the machine oriented, such as a region of storage (Stroustrup, 1991), to the terse equation, such as object = state + behavior (Blaschek, 1994). For the purposes of this dissertation, an object is defined to be a computational entity whose state is hidden behind an interface, an interface that presents the object’s available behaviors.
Aspects represent such crosscutting concerns. An aspect is a modularized and encapsulated crosscutting concern (Kiczales et al., 1997). Both modularization and encapsulation are facilitated by language support. Aspect-orientation, through advanced separation of concerns and the representation of crosscutting concerns by aspects, can be paired with object-oriented problem solving.

1.2.2 Object-Orientation as Dominant and Aspect-Orientation as Newcomer

Object-orientation is currently the dominant programming methodology (Elrad et al., 2001). Object-orientation offers a system of communicating agents and a decomposition technique that favors the real-world object—i.e., concept, thing, or model—and then provides linguistic support for programming said objects. These objects are analogous to their real-world counterparts and are represented as blueprints called classes. From these classes, any number of replicated objects can be created, each of which models the properties and behaviors that define the real-world concept or thing. Numerous advocates of object-oriented programming have asserted that object-orientation has become the dominant methodology because it is a good fit for human thinking; it is based on the natural language we use to describe the world and identifies objects (i.e., nouns) and behaviors (i.e., verbs) in ways analogous to ways we identify them in everyday life. For instance, in a banking system, customers, accounts, and money can be encapsulated as objects.

In addition, the decomposition and modularity properties of software objects are a good approximation for the way people divide and classify things in the real
world (Taivalsaari, 1996). The history of programming languages research has been characterized, in fact, by the search for the right modularity mechanism. Object-orientation proposes that its modularity mechanism is most analogous to the ways in which we categorize things in everyday life (Taivalsaari, 1996).

While object-orientation offers a feasible and useful modularity mechanism, an object is, however, constrained by the necessities of the medium; models or objects must be translated into software that runs on a computer. So, object-orientation does not account well for such things as databases, synchronization of resources, security, or other concepts that are not distinct, discrete things in everyday life (Elrad et al., 2001). In other words, each thing that is encapsulated in an object is a concern, but objects encapsulate some of these concerns well and others poorly.

Those concerns that are encapsulated less well can be recognized by the way in which they are present in a system: they are inconveniently located in many, disparate places. These poorly encapsulated concerns are often tightly coupled or intertwined with the concerns that are better encapsulated, such as customers and accounts in the banking example. Aspect-orientation offers an alternative way to account for these concerns, such as security protocols, that are scattered throughout our systems.

While object-oriented systems can and do work without aspect-orientation, aspect-oriented programming specifically addresses the modularization that object-orientation apprehends less effectively. An aspect gathers into one place an idea that is or would otherwise be scattered throughout an object-oriented program or system.
Whereas objects represent things, an aspect represents something about these things that recurs in the program or system. In other words, aspect-orientation fills in the blanks or catches what falls through the cracks in an object-oriented system. Aspect-orientation, then, addresses the problem of hunting down a concern that recurs throughout our classes or in many different places. Therefore, an aspect better encapsulates a scattered concern, such as a security protocol or a database access in the banking example, than does an object (Elrad et al., 2001).

Aspect-orientation neither replaces nor is integrated into object-orientation. Instead, aspect-orientation proposes that programmers separate concerns within a system. Object-orientation includes primary concerns that are well encapsulated by objects and also secondary concerns that are not well encapsulated by objects. These secondary concerns are well represented by aspects. Aspect-orientation, then, can be specifically designed to work in conjunction with and be compatible with object-oriented programming.

Given this overview of object-orientation and aspect-orientation, the AOPy research project that is documented in this dissertation works from the following premises: that object-orientation is the dominant programming methodology, that object-orientation apprehends some real-world things less well than it does others, and that aspect-orientation has been specifically designed to encapsulate well those scattered concerns that objects poorly encapsulate. From this foundation, the AOPy project establishes the usefulness of aspect-orientation in practice.
1.2.3 Research Focus within Aspect-Orientation

The primary aim of this dissertation—and the AOPy project it documents—is to investigate and provide a narrative description of the usefulness of advanced separation of concerns that aspect-oriented programming offers.

The use of dynamic, lightweight languages—more colloquially known as scripting languages—is an increasingly important area of industrial practice and academic research. As described in van Rossum (1997) and Martin (2001), the use of scripting languages for software development has exploded during the web era. These languages have proven very attractive in the web development arena because of the flexibility and rapid-development cycle using these languages (Ousterhout, 1998; McMillan, 1998).

In the area of academic research on scripting languages, MIT has sponsored the Lightweight Languages Workshop for the past three years (http://ll3.ai.mit.edu/; http://ll2.ai.mit.edu/; http://ll1.ai.mit.edu/). Work on languages such as Python, Perl, Javascript, and Ruby have been well represented in each year’s proceedings. Python is the language used in the AOPy project.

Similarly, aspect-oriented programming is an emerging area of software development. The October 2001 issue of the Communications of the ACM (CACM, 2001) was dedicated to aspect-oriented programming. Aspect-orientation (Kiczales, et al., 1997) has emerged as a post-object model and has garnered increasing attention from developers and researchers because it offers a controllable, modular mechanism for describing the separation of concerns (Hursch and Lopes, 1995) that
crosscut the objects that are used to structure and implement object-oriented software. The post-object model of modularity that aspect-oriented programming proffers is the aspect.

Again, consider the banking example mentioned above. Crosscutting concerns such as persistence, security, and logging will appear in many different implementation objects. As a post-object model, the incorporation of aspect-orientation into a system, such as the example for banking, allows for concerns that crosscut the system, such as security, to be separated and addressed systemwide, rather than individually as in the typical object model.

A bedrock precept of object-oriented design is that each class in a system has a well-defined responsibility (Wirfs-Brock et al., 1990). Due to their crosscutting nature, some concerns in the system—such as persistence, security, and logging—affect parts of many classes. As such, these concerns cannot be viewed as the responsibility of an individual class. So, traditionally, the code that implements the crosscutting concerns is added to each class separately. However, this practice violates the previously mentioned bedrock principle that each class has a well-defined responsibility. An aspect-oriented solution, such as the one proposed for the banking system, modularizes all of the information necessary to address the crosscutting concern in a new language construct, the aspect.

The short history of aspect-oriented programming has focused largely on the use of aspects to extend and enhance systems programming languages, in particular
Java (Czarnecki and Eisenecker, 2000; Kiczales et al., 2001; Lieberherr, 1996; Ossher and Tarr, 1999; Tarr et al., 1999).

In light of this, it is appropriate to investigate the use of aspect-oriented techniques in scripting languages. Other work in the area of aspect-oriented scripting languages, such as A-TOS—an aspect-oriented TCL variant—(Pawlak, et. al., 1999), Pythius (Pythius, 2003), and AspectR (AspectR, 2002), has largely focused on implementation techniques and issues.

This dissertation research project is an ongoing dialogue concerning those considerations that must be accounted for when aspect-oriented techniques are introduced into an object-oriented software development environment. The goal of the AOPy research project is to investigate, the usefulness of advanced separation of concerns—specifically aspect-oriented programming.

1.2.4 Aspect Orientation and the AOPy Infrastructure

As stated above, the goal of the AOPy Project is to investigate, by means of the Python (Python, 2003) scripting language, the usefulness of aspect-oriented programming. Before the experiment-based part of this research that investigates usefulness could take place, it was necessary to build an aspect-oriented infrastructure for Python. The aspect-oriented infrastructure is implemented as a framework. A framework is a collection of classes meant to be extended by inheritance (Budd, 1997). A framework is differentiated from the closely related idea of a class library in that its primary control loop is determined by the problem
domain (Johnson and Foote, 1988) that the framework was designed to address. The full range of aspect-oriented support that is provided by the AOPy framework is demonstrated in the case study described in Chapter 4.

The convergence of design patterns (Gamma et. al., 1995), advanced separation of concerns techniques, and aspect-oriented programming is an active area of inquiry (Hanenberg, 2002; Hannemann, 2002; Noda, 2001a).

Additionally, the research project validates a specific implementation of an aspect-oriented tool—AOPy. The AOPy research project investigates the design and construction of software systems that facilitate the development of aspect-oriented software, thereby defining issues of experimental appropriateness and validity.

As is discussed in Becker's "The Epistemology of Qualitative Research" (1996), qualitative researchers performing observational experiments frame experimental validity within the context of the appropriateness of the questions being asked. Performing an extended case study based on an aspect-oriented solution for the Observer design pattern (Gamma et al., 1995) enhances the understanding of the chosen problem domain for the experimental portion of the AOPy project. Using Becker’s epistemology, this dissertation’s case study of the Observer design pattern establishes the appropriateness of the project’s research questions and, by extension, the validity of the observational experiments.

An aspect-oriented weaver is built and demonstrated to further this line of inquiry. An aspect weaver is a software tool that composes objects and aspects in a prescribed fashion (Böllert, 1999).
The AOPy project investigates the appropriate concerns/aspects that are scattered and tangled throughout a software system. Hence, the AOPy project is an in-depth exploration of when or where incorporation of aspect-oriented programming approaches is most appropriate. In other words, when and how is aspect-orientation useful?

The experiments carried out during the AOPy project are used to evaluate whether developers effectively identify new crosscutting concerns in a specific problem. If and when they do so, their use of aspect-oriented programming ideas, terminology, and techniques is examined. The AOPy Project, then, contributes definitions of and analysis of compatibility of aspect-orientation with existing object-orientation that is observed during the experiment. In other words, the research project defines and analyzes effective identification of new concerns, use of aspect-oriented terms to describe those concerns, and/or the employment of aspect-oriented techniques (i.e., the creation of aspects) during problem solving. Evidence of these aspect-oriented activities by subjects during problem solving is observed when subjects verbalize a shift between concerns represented by objects and concerns represented by aspects in the application of advanced separation of concerns technology.

Assessing and evaluating the incorporation of aspect-oriented programming into existing object-oriented software development practices is the primary goal of this research project.
The software portion of the AOPy project consists of two pieces: an aspect weaver and a set of aspects. The design of the weaver is similar to the general-purpose, runtime aspect weavers described in Böllert (1998; 1997). The system is similar in that it is inheritance based and non-invasive.

As is described in Kiczales et al. (2001), the primary functionality of an aspect-oriented system or language rests upon three criteria: “a join point model, a means of identifying join points, and a means of affecting implementation at join points.” The join point model used in AOPy is based on method-calls. In Ossher and Tarr (1998), it is argued that method-call join points are sufficient for an aspect-oriented system. Like the aspect-oriented extension to the Squeak (Korienek, Wrensch, and Dechow, 2002) Smalltalk system—AspectS (Hirschfeld, 2002)—AOPy does not introduce any new language constructs to support aspect-orientation. The join point language of AOPy is Python itself. At runtime, method-interception is used to enable the insertion of advice at join points. Advice—a means for affecting implementation—is provided in the form of mix-ins.

1.3 Experimental Methodology and Design

Chapter 3 discusses the materials and methods used in this research project in greater depth. That chapter discusses the use of aspect orientation and the experiment design. An overview of the materials and methods used in the AOPy research project (Dechow, 2003) follows below.
1.3.1 The Qualitative Research Process

Qualitative research—and Grounded Theory (Glaser and Strauss, 1967) in particular—was structured in the AOPy project as follows:

1. identifying an area of interest, in this case aspect-orientation and the role of crosscutting concerns in programming scenarios
2. structuring a setting, in this case a simulation of industry software development teams
3. collecting initial data, in this case through thinking-aloud sessions
4. interpreting the data overview, in this case recognizing the failure of subjects to identify crosscutting concerns in the preliminary experiment
5. exploring the phenomenon of interest, in this case through the main experiment of CS 582 during thinking-aloud sessions
6. determining and analyzing results, in this case by using three-point criteria to establish categories of compatibility and noncompatibility
7. dissemination, in this case through several conference presentations during the AOPy project and, ultimately, in this dissertation.

The AOPy research project, therefore, fits into well-accepted practices for conducting qualitative research.

The cornerstone of the AOPy project is to represent the dynamics of professional development teams so that the final analysis provides workable information and approaches for industry practices. Subject backgrounds, workshop preparation, and the design of experiment tasks were considered in relation to this
cornerstone. The background and experiences of the subjects in the CS 582 workshop—with a mixture of academic coursework and industry knowledge and practice—resembled that of the professional software development teams that the author has been associated with in industry.

The qualitative research of the AOPy project involved recording how subjects articulated the experiment problem as they were working on it. The sources for the results were the videotaped recordings of problem-solving sessions that approximated the professional development process and the field notes taken during those sessions.

1.3.2 Experiment Design

Among the artifacts developed for this dissertation is an experiment design. Based upon advice and comments that I received from a dissertation committee member and another researcher, my claims are validated by designing an experiment based on the thinking-aloud method (Lewis, 1982; Jorgensen, 1989). This is an extension of the ‘Simplified Thinking-Aloud’ method that is a component of discount usability engineering (Nielsen, 1989). The use of a thinking-aloud study provides an effective qualitative data generation method from which specific areas of the usefulness of aspect-orientation can be judged.

A preliminary experiment during the planning phase for the main experiment took place in a programming languages seminar, CS180. This work is described in a paper by this dissertation’s author (2004) and in Chapter 3.
The main experiment is indicated by the course number with which the experiment was associated: CS582. The experiment design is described more extensively in Chapter 3. The experiment results are presented in Chapter 5 and analyzed in Chapter 6.

1.4 Case Study: Observer Design Pattern

The application of advanced separation of concerns techniques to design patterns (Noda, 2001a), in order to improve the modularity and composibility of pattern implementations (Hannenmann, 2002), is an active research area. Additionally, other investigators (Hanenberg and Costanza, 2002) have been studying aspect-oriented patterns in their own right.

A case study involving the implementation of the Observer design pattern (Gamma et al., 1995) using the AOPy framework provides an excellent vehicle for further investigation of aspect-oriented programming in the context of design patterns. The described intent of the Observer design pattern is as follows: “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.” More simply, the Observer design pattern is used to decouple code behavior. The rationale for studying Observer, in particular, is quite straightforward and is discussed in Chapter 4, which describes the case study for the framework used in this research project.
1.5 Experiment Overview

Two separate thinking-aloud experiments were performed as a part of the AOPy research project. They are each briefly described below.

1.5.1 Preliminary Experiment: CS180 Programming Languages Seminar

The main experiment for this research project was preceded by a preliminary experiment. The preliminary experiment was used to develop material, techniques, and the working hypothesis of compatibility as usefulness for the main experiment, which is introduced in Section 1.5.2. The preliminary experiment was part of the planning phase for the main experiment and was not designed to produce results for analysis. Instead, the preliminary experiment served to generate the focus, namely on conceptualization of modularity as it is expressed during incorporation of aspect-oriented ideas, terminology, and techniques, for the main experiment.

The preliminary experiment was part of CS180 Programming Languages Seminar, taught at Knox College in Galesburg, Illinois, in Fall 2003. Seven undergraduate students served as subjects for the preliminary experiment.

The Python scripting language served as the implementation language. The Pythius aspect-oriented toolkit was used to facilitate aspect-oriented programming. The development of web applications was selected as a problem domain (McMillan, 1998), and the Python-based Snakelets web server (Snakelets, 2003) was used as a representative implementation of a web applications server.
Based on observations during the preliminary experiment, the main experiment was developed. Because the preliminary experiment was designed as part of the planning stage and not for analysis of the research area itself, it is only briefly described here and in Chapter 3 and mentioned briefly in Chapter 5, which presents the results of the main experiment.

1.5.2 Experiment: CS582 Aspect-Oriented Programming Workshop

The research experiment—CS582 Aspect-Oriented Programming Workshop—was performed in March 2004 at Oregon State University in Corvallis. CS582 is a graduate-level course on object-oriented programming, analysis, and design. Eight graduate students in that course served as subjects who participated in the AOPy workshop and experiment.

The experiment consisted of three stages: the initial survey, the experiment, and the follow-up survey. The experiment design is discussed more extensively in Chapter 3. The initial survey that subjects completed is included in the Appendix, and results of subjects' backgrounds from these surveys are included in Chapter 5. The results of the experiment are discussed in Chapter 5.

The experiment elicited recorded natural language in an aspect-oriented programming problem-solving session. Analysis of the experiment focused on compatibility, which this researcher defines as points in time during problem-solving when subjects incorporated aspect-oriented programming ideas, terminology, or techniques into the object-oriented programming problem. Thinking-aloud
assessment was used to observe the problem-solving session and determine these instances of evidence of compatibility of aspect-orientation with object-orientation. Chapter 5, in the results it presents, further defines the evidence of compatibility in the problem-solving sessions. Chapter 6 analyzes relevant sample evidence of compatibility that occurred within the experiment.

1.6 Conclusion

The work described in this dissertation is the AOPy project (Dechow, 2003; Dechow, 2004). The primary aim of the AOPy project is to investigate, the usefulness of advanced separation of concerns—specifically aspect-oriented programming ideas, terminology, and techniques.

In the AOPy project, the purpose of researching aspect-orientation and the development of aspect-oriented systems is communication—through a narrative description and analysis—with practitioners with the goal of understanding, modifying, and improving current object-oriented practice.

Assessing the incorporation of aspects into existing object-oriented software development practices is the primary goal of this research project and determines usefulness in specific areas of problem solving. A qualitative research methodology to address the issue of aspect-oriented programming's usefulness was constructed to develop insight into which ideas, terminology, and techniques of aspect-oriented programming might be especially important to developers. These insights were developed by observing a group of students in a series of thinking-aloud experiments.
that drew its subjects from a graduate workshop on aspect-oriented programming. In doing this observation of practitioners, this research project validates specific areas of usefulness of aspect-oriented programming by defining and analyzing areas of compatibility between aspect-orientation and object-orientation. This thinking-aloud experiment also indicates specific areas in which potential usefulness is thwarted by noncompatibility in actual practice.

Perhaps the most important contribution that this dissertation makes to understanding and addressing the problem of complexity in software systems is that the AOPy research project establishes a definition of compatibility and provides an analysis of sample instances during problem solving that evidence of compatibility between object-orientation and aspect-orientation. The primary scientific contribution of this dissertation, therefore, is a narrative description of the use of aspect-oriented programming in a series of controlled, problem-solving scenarios. A structured dissection of the use of the aspect-oriented methodology is presented by means of an analysis of those instances in the problem-solving experiment where the natural language used by the subjects indicates compatibility between discussing separation of concerns in object-oriented terms and discussing separation of concerns in aspect-oriented terms. Theories describing the use of aspect-oriented ideas, terminology, and techniques are generated and refined by means of the Grounded Theory methodology.

The results of the experiment and the conclusions of this dissertation indicate that the level of usefulness of aspect-orientation is related to how compatible aspect-
orientation is with object-orientation. When aspect-orientated ideas, terminology, and techniques are highly compatible with object-orientation, aspect-orientation is useful and can be used effectively to implement software systems. However, when aspect-oriented ideas, terminology, and techniques are feasible (i.e., a scattered concern exists) but are not compatible, aspect-orientation may be potentially useful but is not useful in practice. The AOPy project, then, not only establishes compatibility as usefulness but also suggests that non-compatibility—thwarted usefulness—be addressed by software developers so that potential usefulness of aspect-orientation can be actualized.

Because this dissertation 1) analytically explores areas of compatibility of aspect-orientation with object-orientation, 2) defines areas of compatibility thwarted in practice, and 3) establishes kinds of subject background that may foster usefulness, this research project can serve as a foundation for the development of aspect-oriented design methodologies.
CHAPTER 2 LITERATURE REVIEW

2.1 Introduction

Advanced Separation of Concerns techniques such as Aspect-Oriented Programming, Adaptive Programming, and Subject-Oriented Programming are often described as belonging to the family of Generative programming (Eisenecker, 1997; Czarnecki and Eisenecker, 2000; GCSE, 2001; GPCE, 2002; GPCE, 2003) techniques. Generative programming is a recent software engineering methodology that promotes the development of reusable families of software products chiefly via the automated production of source code.

That said, some methodologies and tools that are also considered to be GP techniques—components, domain engineering, feature analysis (Czarnecki and Eisenecker, 2000)—are not generally considered to be advanced separation of concerns technologies. As a result, this literature review is limited to those generative programming techniques that are also advanced separation of concerns techniques (see primarily Chapter 7 in Czarnecki and Eisenecker, 2000) and in particular, those techniques most closely related to aspect-oriented programming.

This review begins with a discussion of the history and description of separation of concerns ideas.
2.2 Separation of Concerns

Acknowledgment of the origination of separation of concerns generally point to two pieces of foundational work. On page 123 of Czarnecki and Eisenecker (2000), the authors cite Dijkstra’s *A Discipline of Programming* (1976) as the origin of separation of concerns. Dijkstra’s viewpoint is summarized as “dealing with one important issue at a time” (Czarnecki and Eisenecker, 2000). Other researchers, including Habra (2001), Kiczales et al. (1997), Ossher and Tarr (2001), and Pace and Campo (2001), cite Parnas’s influential paper “On the Criteria to be Used for the Decomposition of Systems into Modules” (1972) as the beginning of the study of separation of concerns. In this paper, Parnas identifies modular decomposition as “the technique for assigning responsibility to a module” (Parnas, 1972).

The original intent of separation of concerns is closely aligned with notions of functional decomposition and modularization (Kiczales et al., 1997). Kiczales et al. (1997) go on to describe functional decomposition as follows: “The exact nature of the decomposition differs between the language paradigms of course, but each unit is encapsulated in a procedure/function/object, and in each case, it feels comfortable to talk about what is encapsulated as a functional unit of the overall system.” As is stated in Czarnecki and Eisenecker (2000), “Most analysis and design notations and programming languages provide constructs for organizing system descriptions as hierarchical compositions of smaller, modular units.” Traditionally, the design decompositions—those that result in hierarchically composed software
systems—have been a reflection of the functional nature of the system (Kiczales et al., 1997). This tradition has led software development to a state of affairs that Tarr et al. (2001) refer to as the “tyranny of the dominant decomposition.”

In the mid-1990s, Hursch and Lopes claimed that research associated with separation of concerns was leading to “the emergence of a new paradigm.” Hursch’s and Lopes’ paper, “Separation of Concerns” (1995), summarizes the early research into methods of composition and decomposition that emphasize separating primary concerns (or basic concerns, in their terminology) from each other, and then further identifying and separating crosscutting concerns (or special concerns, in their terminology). Furthermore, Hursch and Lopes describe seven different, special concerns: algorithm, data organization, process synchronization, location control, real-time constraints, persistence, and failure recovery.

Hursch and Lopes discuss three techniques for separating the aforementioned special concerns from basic concerns: meta-level programming, pattern-oriented programming, and composition filters. Because of their relationship to two of the techniques discussed later in this chapter (aspect-oriented programming and adaptive programming), two of the three concern handling techniques—pattern-oriented programming and meta-level programming—are relevant to this discussion.

Pattern-oriented programming (Lieberherr, 1996) is an outcome of research on the law of Demeter (section 2.4) and, as is described in section 2.5, has led to the adaptive programming methodology (Lieberherr, Orleans, and Ovlinger, 2001). The
essence of pattern-oriented programming is based on two concepts: class dictionary
graphs and traversal strategies. Class dictionary graphs (Lieberherr, 1988) are
defined as follows:

Informally, class dictionary graphs express object-oriented class
hierarchies as mathematical graph structures. In a class dictionary
graph, instantiable and abstract classes are represented as vertices, and
part-of and inheritance relationships are expressed as edges.

Secondly, traversal strategies are a "set of constraints that declaratively define a path
through the class dictionary graph" (Lieberherr, Silva-Lepe, and Xiao, 1994).
Traversal strategies are used to compose a single instance of a program from a
possible family of programs. In the adaptive programming methodology, the
traversal strategies are referred to as propagation patterns (Lieberherr, 1992).

The subject of this dissertation, aspect-oriented programming, is a direct
descendant of reflective, meta-level programming techniques via Kiczales' work on
open implementations (Kiczales et al., 1997). Meta-level programming is also known
as meta-object programming. Programming languages—such as the Common Lisp
Object System (Bobrow, Gabriel, and White, 1993)—that support meta-object
programming do so via an architecture known as a meta-object protocol (Kiczales,
des Rivieres, and Bobrow, 1991). Meta-object programming and reflective
programming are related techniques that allow a software system to inspect,
evaluate, and possibly change the state or architecture of the running system.
2.3 Introduction to Aspect-Oriented Programming

In the paper "Aspect-Oriented Programming" (Kiczales et al., 1997), the authors detail how the tangled and crosscutting concerns that are present in software systems make the maintenance and modification of those systems extremely difficult, if not impossible. Code becomes tangled when design decisions addressing large numbers of concerns become scattered throughout an implementation. Tangled concerns tend to be functional notions/units that cut across—crosscut—several components.

According to Ossher and Tarr (2001), separation of concerns refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concern, defined as a concept, goal, or purpose. Advanced separation of concerns techniques are an attempt to address the problems associated with creating, maintaining, and modifying complex software systems. In other words, differentiating between separation of concerns and advanced separation of concerns acknowledges that separation of concerns techniques apply largely to those systems in which only one concern representation mechanism is available. As an example, an object-oriented system is comprised of objects and, therefore, uses only one mechanism to represent concerns. By contrast, a system using advanced separation of concerns techniques is comprised of both subjects and objects, in the case of subject-oriented programming, or aspects and objects, in the case of an aspect-oriented system. Aspect-oriented programming is an advanced separation of
concerns method that addresses the question of how to untangle code that is difficult to develop and maintain.

Aspect-orientation is characterized by its attempt to cleanly localize—modularize—those concerns of a software system that are orthogonal to the system’s functional components (Czarnecki and Eisenecker, 2000). These non-functional concerns are said to be crosscutting. This nomenclature and view of non-functional concerns is an acknowledgement of the fact these concerns are often scattered throughout the system’s functional components. Software systems created using aspect-oriented methods are then composed of a set of functional units, which are those components that encapsulate the system’s primary concerns, and a set of aspects, which are components that encapsulate the system’s non-functional, crosscutting concerns (Kiczales et al., 1997).

At the implementation level, aspects are concrete software artifacts that, for all practical intents and purposes, require language support. At the language level, an aspect is most similar to an object. At this level, aspects are units that encapsulate state, behavior, and behavioral enhancements in other components. These behavioral enhancements are known as advice (Kiczales et al., 1997).

The process of composing the system from functional components and aspects is called weaving. The rules that guide the composition process are based upon the insertion of advice at specific points of the program’s execution. These
specific points of execution are called join points (Kiczales et al., 1997). The tool used to insert advice is known as an aspect weaver (Kiczales et al., 1997).

Recall the banking system example from section 1.1. In that example, the primary concerns represented the following: customer and account management, interest computation, interbank transactions, statement generation, customer care, etc. This same banking system has crosscutting concerns related to transaction management, persistence, synchronization, authentication, security, etc.

Below is a Python-inspired pseudo-code listing that demonstrates how crosscutting concerns can become tangled with primary concerns. Note that only those statements that represent the primary concern are shown in code. The crosscutting concerns are denoted only by comments.

```python
class CustomerAccount(Account):
    def __init__(self, AcName, OpenDate, Balance=0.0):
        # primary data members
        self.OpenDate = OpenDate
        self.AcName = AcName
        self.Balance = self.OpenBal = Balance
        # other data members

    def processTransaction(self, amount, ...):
        # authenticate

        # design by contract constraints

        # lock for multi-threading

        # log beginning of operation

        # do the primary operation
```
The primary concern of this class is understood to be the representation of a customer's bank account. One of the primary responsibilities of this class is updating the account's Balance data member, which is encapsulated in the `processTransaction()` method. However, notice that crosscutting concerns such as authentication, design-by-contract hooks, and logging are also present in the method.

This is an example of how behaviors can become tangled. In an aspect-oriented solution, each of the crosscutting concerns (such as the synchronization, the authentication, and the logging concerns) which are tangled with the primary concern (updating the account's Balance) are be separated into their individual aspects.

2.4 Law of Demeter

The law of Demeter (Lieberherr and Holland, 1989) describes a methodology for implementing flexible, loosely coupled, object-oriented architectures. The
essence of the law of Demeter can be distilled as follows: do not directly invoke methods on an object returned as a value. Instead, create a new method that encapsulates this behavior.

By employing the law of Demeter, a resulting implementation is more structurally flexible. The creators of the law also claim that program comprehensibility is increased because of smaller method size and improved control flow. Ultimately, using the law of Demeter encourages a subtle generalization of information hiding that the creators of the law refer to as information restriction (Lieberherr, 1996).

Nonetheless, any claims of improved program comprehensibility must be balanced with the increase of program complexity that results from the proliferation of new methods.

It is through the Demeter Project (see Section 2.5) and its associated methodology, adaptive programming, that a part of the relationship between aspect-oriented programming and the law of Demeter can be seen.

2.5 Adaptive Programming

The law of Demeter has been integral to a long-term research project, the project from which the law derives its name, the Demeter Project. The core of the Demeter Project is the investigation of separation of concerns via adaptive programming.
As was described in section 2.2, adaptive programming is based on two concepts: class dictionary graphs and traversal strategies. An example of how to define class dictionary graphs and traversal strategies using the DemeterJ (Demeter for Java; DemeterJ website) tool is described below.

The DemeterJ tool has two ways to specify the class dictionary graph for a software system. One way uses a visual tool for creating a UML-like class graph. Class dictionary graphs can also be specified by a grammar (Lieberherr, 1988).

To demonstrate a traversal strategy, a simple class diagram is shown below:

![Class dictionary graph](image)

*Figure 2.1: Class dictionary graph depicting members visited (in gray) by a traversal strategy.*
An example of a traversal strategy for the resulting class dictionary graph is the following string: “from A via B bypassing C to D.” A visitor (an implementation of the visitor design pattern as described in Gamma et al. [1995]) is then used to perform the actual computation as guided by the traversal strategy.

The Demeter Project has resulted in the creation of a suite of software tools (Lieberherr, Orleans, and Ovlinger, 2001) that facilitate the development of object-oriented systems that separate their structural and behavioral concerns. As such, the law of Demeter can be seen as an early ad hoc response to the, at that time undeclared, need for the separation of concerns. It is this notion of separation of concerns that is also at the heart of aspect-oriented programming (Czarnecki and Eisenecker, 2000).

As adaptive programming was further developed, other concerns—including the seven described in Hursch and Lopes (1995)—were identified and investigated. Among the concerns that received the most attention were the synchronization and remote invocation concerns. The synchronization and remote invocation concerns were the subject matter of Lopes’ dissertation project, *D: A Language Framework for Distributed Programming*. This work also began the process of fitting adaptive programming into the aspect-oriented programming framework. A more formal description of the outcome of this work is given below in section 2.10.
Hence, Hursch’s and Lopes’s work (1995) establishes that the law of Demeter and aspect-oriented programming are related by the fact that they are both techniques for handling the separation of concerns.

2.6 Subject-Oriented Programming

Subject-oriented programming was a method for composing object-oriented software systems by means of subjects, sometimes referred to as subject views (Ossher et al., 1995; Foote, 1995). A subject is a set of classes and/or class fragments from a class hierarchy that describe a specific perspective, or context, of a solution domain (Ossher and Tarr, 1999a). In this case, the concept of perspective is analogous to the concept of view as described in Kruchten (1995). This analogy is also the motivation for the use of the phrase subjective view synonymously with subject (Ossher et al., 1995).

The following quote from Mili, Harrison, and Ossher’s paper “Supporting Subject-Oriented Programming in Smalltalk” (1996) further clarifies the definition of a subject:

Any program from which may be compiled successfully as a unit can be considered a subject. Subject-Oriented composition allows us to define the same class in two different subjects, possibly under a different name, with different ancestors, instance variables, and methods; instead of one definition overriding the other, as in Smalltalk, or flagging it as an error, as in C++, the two partial definitions are combined in interesting and semantically meaningful ways.
Individual or groups of subjects can be composed into enterprise applications.

As an example of how subjects represent different views of an object, the paper “Subject-Oriented Programming: Supporting Decentralized Development of Objects” (Ossher et al., 1995) asks the reader to contemplate a Truck object. Services provided by objects from the same Truck class might be required by a variety of enterprise applications: Shipping, Transportation, Scheduling, Billing, etc. Each of these applications will have a different view of what constitutes a Truck. Loosely coupled and decentralized development is facilitated by representing these views as subjects. A subject is able to incorporate the traditional object-oriented concepts of state and behavior via a named association with an implemented object (Ossher et al., 1995). The named association is termed a label. Each label gives a subject’s name and declares the portion of a class hierarchy that is mapped to the subject.

The example shown below demonstrates how a subject is comprised of class definitions for one or more classes.

class Truck():
    def __init__(self, Length, Width, Height):
        # Truck data members
        self.Length = Length
        self.Width = Width
        self.Height = Height

    def processTransaction(self, amount, ...): pass
The example shown above—again, in Python-like pseudo-code—demonstrates how the Shipping subject consists of a Box class and one possible view of what a Truck class should look like.

The next step in the creation of a Shipping subject makes use of a subject specification file. The contents of the subject specification file for the Shipping subject are shown below:

```
[Shipping]
...
composableClasses = Truck Box
```

Listing 2.3. Shipping subject’s subject specification file

Consider a second example:

```python
class Truck( ):
    def __init__(self, Route, GasCapacity, MPG):
        # Truck data members
        self.PlannedRoute = Route
        self.GasCapacity = GasCapacity
        self.TruckMPG = MPG
```
def SetRoute(self, Route, ...): pass

def Range( self ): pass

class City( ):
    def Name( self ): pass

Listing 2.4. Transportation subject

The second example demonstrates how the Transportation subject consists of a City class and a second, but significantly different, view of what a Truck class should look like. Again, the Transportation subject is described by a corresponding subject specification file.

Like other separation of concerns technologies, subject-oriented programming was able to identify and facilitate the modularization of crosscutting concerns. Subject-oriented programming was also able to integrate these concerns into a software system in a controlled and formal—as opposed to unplanned and ad hoc—manner.

A subject-oriented software system is composed of subjects by a specialized compiler, called a compositor. The composition process is carried out according to a declarative specification described in a set of composition rules (Ossher et al., 1996). The role of the compositor is analogous to that of the weaver in aspect-oriented programming.
As an example of a composition rule and the composition process, the example below shows how a Routing application is composed from the previously developed Transportation and Shipping subjects. It is necessary to construct a composition rule that will guide the subject compositor as the two subjects “are combined in interesting and meaningful ways” (Mili, Harrison, and Ossher, 1996). In this instance, a simple composition rule will suffice:

\[
\text{Routing} = \text{Merge} \{ \text{Shipping}, \text{Transportation} \}
\]

It is important to note that the composition methodology works on binary object code under the guidance of the label (Ossher et al., 1995).

The hyperspaces method, described later in section 2.9, is a generalization of subject-oriented programming. Before going any further, it is necessary to develop some more terminology to discuss separation of concerns in general and the specifics of the hyperspace model. Each of the following definitions can be found in at least one of the following references: Ossher and Tarr (1999b), Tarr et al. (2001), or Ossher and Tarr (2001).

The entire range of concerns that can be attributed to a software system is its concern space.

The dimension of a concern space is an attempt to quantify and organize the inhabitants of the space. The phrase “dominant dimension” recognizes the fact that
most current methodologies only allow concerns to be handled in a single dimension (Tarr et al., 2001). This recognition has resulted in the separation of concerns catchphrase, "the tyranny of the dominant decomposition."

Each separation of concern methodology has its own concern handling technology—object-oriented programming has objects (Budd, 1997), aspect-oriented programming has aspects (Kiczales et al., 1997), subject-oriented programming has subjects (Harrison and Ossher, 1993), and adaptive programming has adaptive methods (Lieberherr, Orleans, and Ovlinger, 2001). While other separation of concerns technologies exist, these four are relevant to this dissertation. Each concern handling technology has a dimensionality or degree association with it.

### 2.7 Dimensionality and Separation of Concerns

The dimensionality of an object-oriented system is one (Ossher and Tarr, 1999b). The only way that a developer can represent a concern in the system is by encapsulating the concern in an object. The result of this one-dimensional decomposition of object-oriented programming is that other concerns are scattered and tangled throughout the system.

The dimensionality of the other systems described in this chapter is two. They are additive systems in that they combine their own concerns technology with objects. So, in the case of aspect-orientation, aspects + objects yields a two-dimensional separation of concerns. This extra dimension affords aspect-oriented
developers the ability to separate the crosscutting concerns of the system, such as authentication, tracing/logging, persistence, remote method invocation, etc., from the system's primary concerns (Kiczales et al., 1997).

2.8 Multi-Dimensional Separation of Concerns

Some researchers have postulated that one or two degrees are not enough. Clearly, systems that contain multiple, overlapping concerns are being built everyday (and have been built for decades). But as is described in Hursch and Lopes (1995), these systems have scattered and tangled concerns. These systems struggle with the inability to provide a "clean" separation of concerns (Ossher and Tarr, 1999b). In a methodology that only allows for one-dimensional decomposition, it is still a straightforward process to identify and design for the separation of concerns. However, as the process moves on to implementation, the intermingling and overlapping of the concerns at the code level will lead to the following problems (Hursch and Lopes, 1995):

1. Since all of the concerns must be dealt with in the same dimensions (classes in the case of object-oriented programming, generalized procedures in structured techniques), writing tangled code is more complex than it should be.

2. The complexity of the tangled code reduces its comprehensibility.
3. The strong coupling of the tangled concerns hinders maintenance and impedes modification.

As is pointed out above, the overall result is that the "-ilities"—maintainability, comprehensibility, trace-ability, evolve-ability, etc.—of the software system are degraded (Ossher and Tarr, 1999b).

This situation is exactly that which an N-degree separation of concerns methodology is intended to address. The researchers have given the following name to the general problem space: MultiDimensionl Separation of Concerns. They argue that most problem domains contain MultiDimensional Concerns. And as such, most problem domains require a MultiDimensional Separation of Concerns handling technology. The model of hyperspaces is their specific response to the need for an MultiDimensional Separation of Concerns methodology.

2.9 Hyperspaces

As was the case with adaptive programming (and its predecessor, the law of Demeter), the concept of hyperspaces arose from work being done in the area of separation of concerns (Hursch and Lopes, 1995; Parnas, 1972). The primary hyperspaces researchers—Ossher, Tarr, and Harrison—first investigated the area of separation of concerns through a methodology known as subject-oriented programming (Harrison and Ossher, 1993).
Hyperspaces is a broad ranging methodology, encompassing the entire range of the software engineering lifecycle. It is necessary to provide here additional terminology that is specifically related to the hyperspace model. Again, each of the following definitions can be found in at least one of the following references: Ossher and Tarr (1999b), Tarr et al. (2001), or Ossher and Tarr (2001)

A software system is comprised of a discrete set of artifacts—RFPs, requirements, design documents, UML diagrams, working code, etc. In short, an artifact is any material, in some context-appropriate formalism, that is used to produce a working software system. The materials in an artifact are typically constructed of units. Units are classified according to whether they are primitive units or compound units.

In the hyperspaces model, primitive units are referred to as atomic, in the sense that they are indivisible. In the formalism of source code, a method and an instance variable are examples of a primitive unit. In the context of system documentation, a single requirement is a primitive unit. Compound units are groups of primitive units. Compound units are generally referred to as modules. Examples of modules include classes, packages, feature descriptions, and event-traces. All of these examples, though, are somewhat context sensitive.

Although, the preceding definitions were provided in order to develop a working vocabulary for the hyperspace model, each of the definitions thus far relates generally to the production of any software system. The notion of artifacts, units,
modules, and concerns should be intuitively familiar to an object-oriented architect, a web developer, or a COBOL programmer.

A hyperspace is "a concern space that facilitates multi-dimensional separation of concerns" (Ossher and Tarr, 1999b). A hyperslice is an ordinary set of modules. Ultimately, the modules can be implemented by any appropriate language construct—objects, aspects, methods, etc. A hyperslice represents the implementation formalisms of a single concern (Ossher and Tarr, 1999b). It is permissible to have overlap between hyperslices. That is to say, different forms of a specific module may appear in more than one hyperslice. This is a carryover from subjects and subject-oriented programming (Harrison and Ossher, 1993).

The specification used to create a software system can be found in a hypermodule. A hypermodule consists of a set of hyperslices and a composition rule set. The composition rule set serves as a guide for the compositor. The compositor is then able to compose a group of hyperslices into a functioning software system. This organization is also the result of the hyperspaces model's subject-oriented roots.

It has been pointed out on several occasions that the hyperspaces model owes much to its subject-oriented parentage. Hyperspaces are also a generalization of subjects. Not only can subjects be used as a concern composition technique in hyperspaces, but so can aspects, features, classes, etc. In short, any of the concern handling methods that have been mentioned thus far in this chapter can be given a concern mapping in the hyperspaces model.
The Hyper/J (Hyper/J website) integrated development environment is the first software tool that attempts to support the hyperspaces model. As indicated by the "J" in the name, Hyper/J is based on Java. The researchers place great importance on ensuring that Hyper/J is compatible with standard Java sources. As such, Hyper/J operates on class files. A direct result of this operation is that Hyper/J is able to work with systems for which the source code is unavailable.

The Hyper/J toolset requires that a developer provide three files in addition to the Java implementation files (Lai, Murphy, and Walker, 2000). The hyperspace file describes the Java class files that are to be manipulated. A concern mapping file provides the direct relationship between the system's concerns and the Java classes that implement those concerns. Finally, the hypermodule file describes how the hyperslices are to be composed into the working software system.

The separation mechanism in Hyper/J cannot provide total separation of concerns since a hyperslice may depend on some definitions that are made elsewhere. The composition rules are used to manipulate and resolve the dependencies between different concerns in the software system.

The Hyper/J tool is just the beginning step in getting software developers to experiment with the hyperspaces model of multidimensional separation of concerns.
2.10 More Aspect-Oriented Programming

In a recent *Communications of the ACM* article, Kiczales et. al. (2002), had this to say about aspect-oriented programming: “AO [aspect-oriented] languages have three critical elements: a join point model, a means of identifying join points, and a means of affecting implementation at join points.”

To support a join point model means to have an underlying model that denotes what program execution events are of interest to the aspect-oriented programming system. Identification of a join point can be handled in a variety of fashions: pre-processing, runtime method interception, class loading, or bytecode instrumentation. Affecting implementation—i.e., behavior—at join points is accomplished by advice. The advice statements in an aspect are analogous to the methods in a class.

AspectJ (Laddad, 2003), an aspect-oriented extension to the Java programming language, is the de facto aspect-oriented toolset.

2.11 Adaptive Programming vs. Aspect-Oriented Programming

As was mentioned above, the researchers behind the law of Demeter and the Demeter Project argue that adaptive programming is in fact a special case of aspect-oriented programming. The gist of their argument is that adaptive programming is a special case of aspect-oriented programming that is concerned with object graphs and algorithms for traversing those graphs. The argument is stated with great
Adaptive Programming (AP) is the special case of Aspect-Oriented Programming (AOP) where some of the building blocks are expressible in terms of graphs and where the other building blocks refer to the graphs using traversal strategies. A traversal strategy may be viewed as a partial specification of a graph pointing out a few cornerstone nodes and edges. A traversal strategy crosscuts the graphs it is intended for, mentioning only a few isolated nodes and edges. Traversal strategies may be viewed a [sic] regular expression specifying a traversal through a graph [...]. In summary, AP = AOP with graphs and traversal strategies. As an application of the earlier distinction, we can say that AP is a special case of AOP where some of the crosscutting is expressed adaptively using strategies to embed small graphs into large graphs.

Finally, it bears mentioning that the law of Demeter and aspect-oriented programming are deeply related in terms of their intended use and the outcome of their use. Both methods are intended to facilitate the development of software systems that are flexible, maintainable, and comprehensible (Lieberherr and Holland, 1989; Czarnecki and Eisenecker, 2000). Additionally, both methods are intended to achieve these goals by identifying and handling the concerns that more traditional object-oriented development methods deal with in an unsatisfying manner.

2.12 Hyperspaces vs. Aspect-Oriented Programming

As was mentioned above, aspect-oriented programming is also a response to the need for software development methodologies that support advanced separation
of concerns. As such, aspect-oriented programming and hyperspaces are more alike than they are different.

Both methods are meant to enable software developers to create flexible, coherent systems that can evolve gracefully. After all, it is the aim of every software engineering methodology to support the "-ilities" of software (Tarr et al., 2001).

Additionally, both hyperspaces and aspect-oriented programming support the ability to identify and encapsulate crosscutting concerns. In this same vein, both hyperspaces and aspect-oriented programming enable the software developer to describe the appropriate points—join points—in the system’s execution in order to introduce the behaviors associated with the now encapsulated concerns (Ossher and Tarr, 1998).

Finally, both hyperspaces and aspect-oriented programming are able to describe concerns in more than one dimension. Thus, both are able to break the tyranny of the dominant decomposition. Ironically, it is this same facility that leads to the greatest difference between the two methodologies.

As was noted earlier, aspect-oriented programming can identify and separate concerns in two dimensions—classes and aspects. The aspect-oriented programming literature (Kiczales et al., 1997) points out that aspects are generally orthogonal to the software system’s primary decomposition. While this may be true, it is also likely that many of a system’s concerns are likely to overlap (Tarr et al., 2001). This fact
limits the ability of aspect-orientation to separate the concerns in these areas of overlap.

On the other hand, as is made explicit in the name of the hyperspaces project, the hyperspaces model is meant to be an N-degree of freedom concern handling system, with no one concern taking precedence over another. The hyperspaces researchers would claim that, as a two-degree system, aspect-oriented programming is, by definition, valuing some concerns over others. This notion that some concerns are more valuable than others is primarily a legacy of the problem that aspect-orientation was meant to resolve.

Aspect-orientation is meant to facilitate the separation of functional from non-functional concerns. Hyperspaces are an attempt to separate and encapsulate all of a software system’s concerns from one another. Nonetheless, even the creators of hyperspaces admit that the ability of their model to achieve a “clean” separation of all of a system’s concerns from one another is still largely unproven (Tarr et al., 2001).

Finally, in both systems, the join points can be expressed in terms of operations—method invocations. In the hyperspaces model, method invocation is the only way to specify a join point. Some aspect-oriented systems, like AspectJ (AspectJ website) also allow join points to take place at the statement level. This difference greatly complicates the development of a general-purpose aspect weaver (Ossher and Tarr, 1998).
2.13 Aspect-Oriented Programming & Design Patterns

Design patterns are a literary form for describing the separation of concerns in certain well-understood problem domains (Gamma et al., 1995). Several researchers have recently addressed the issue of reformulating the Gang of Four (GoF) patterns with aspect-oriented techniques. One such recent aspect-oriented version of a GoF pattern—Observer—is especially relevant to this dissertation. In order to substantiate the validity of this dissertation's investigation of aspect-oriented programming, the Observer design pattern was implemented using aspect-oriented techniques (Dechow, 2004b).

In other instances, some patterns have been shown to be very useful in implementing full-blown aspect-oriented or aspect-oriented-like systems (proxy, interceptor, and decorator).

Finally, some recent work has been done in documenting aspect-oriented patterns (Hanenberg, 2002).

2.14 Conclusions

Separation of concerns is a software engineering principle that encourages the identification and modularization of the concerns of a software system.

The law of Demeter is an object-oriented rule of thumb for separating behavior from structure. The law of Demeter ultimately led to the development of
adaptive programming. Adaptive programming is characterized by traversal strategies and object graphs.

The law of Demeter and aspect-oriented programming are related by their attempts to separate concerns.

Hyperspaces is a methodology that supports multi-dimensional separation of concerns. The hyperspaces model is an outgrowth of work done on subject-oriented programming.

Both Hyperspaces and aspect-oriented programming are responses to the need for methods that support advanced separation of concerns. The hyperspaces technique is able to separate multiple concerns in multiple dimensions. Aspect-oriented systems modularize and separate concerns in two dimensions: functional concerns and aspects. This gives aspect-orientation two dimensions along which to separate concerns.
CHAPTER 3 MATERIALS AND METHODS

3.1 Introduction

There are two primary issues to be discussed in this chapter. The first is an aspect-oriented software system that was designed and implemented in order to establish the validity of aspect-oriented scripting languages and the AOPy Project's goals. The second is an experimental methodology to verify the usefulness of aspect-oriented techniques in the context of scripting languages in practice. The implementation of AOPy and the experimental methodology used to test it are discussed here.

3.2 Support for Aspect-Orientation

AOPy takes an object-based, runtime approach to its support for aspect-oriented programming. AOPy aspects are instantiated from ordinary Python classes. Like the aspect-oriented extension to the Squeak (Korienek, Wrensch, and Dechow, 2002) Smalltalk system, AspectS (Hirschfeld, 2001), no new language specific to aspect-oriented programming is required.

Using the AOPy framework is a straightforward task in client code.

In the case of a single aspect being woven into a single target object, the modifications to client code can be kept to a minimum of four steps:
1. create the objects and aspects
2. import the necessary modules—AOPy and the appropriate aspects
3. instantiate a weaver
4. weave the aspect into the target object

For the purposes of this example, the traditional development effort associated with designing and implementing the classes/objects—in this example, BarTargetObject—of a system is representing as a single action, Step 1. The task of creating the aspects—in this example, FooAspect—for a software system has a similar level of development effort. It is assumed that this is also included in Step 1. That said, due to incremental style of development that scripting languages afford, all of the tasks can be accomplished in an iterative manner.

The Python code for effecting the weaving process—corresponding to Steps 2, 3, and 4—the three steps is as follows:

1. `import AOPy, TraceAspect`
2. `myWeaver = AOPy.AspectWeaver( )`
3. `myT = myWeaver.weave( TargetObject, TraceAspect)`

To gain access to a module—in this case the AOPy and TraceAspect modules—you must first import the module. The import statement in Python creates a mapping between an internal module name and an external file. Then it is possible to access the names in the module using a dotted notation.
In this example, two files, AOPy and TraceAspect, are now accessible. The AOPy module contains the classes that define the aspect-oriented weaving framework. The TraceAspect module contains a single aspect for tracing the activities of the class that it will be woven into.

To begin the weaving process, an instance of the weaver must be created. The weaver instantiation is accomplished in step 2, the second line of Python demonstration code.

Finally, the TraceAspect aspect must be woven into the target object. This is carried out in step 3. When the weaving process is completed, the returned object preserves the message protocol of the original object. Additionally, no changes or additions to the Python code in the target class are necessary.

In the October 2001 issue of the Communications of the ACM, Kiczales et al. (2001) assert “AO [aspect-oriented] languages have three critical elements: a join point model, a means of identifying join points, and a means of affecting implementation at join points.” The remainder of this section describes how each of the elements in Kiczales’s definition is supported in AOPy.

3.2.1 Join Point Model

A join point is a well-defined point in the execution of a program (Gradecki, 2003). The join point model of AOPy is based on method calls. In Ossher and Tarr (1998), it is argued that method call join points—“operation join points” in Ossher
and Tarr's terminology—are sufficient for most situations. Additionally, method call join points have the added benefit of being "especially appropriate in an object-oriented context, because it adds the power of composition naturally within the object-oriented paradigm" (Ossher and Tarr 1998).

An AOPy join point is similar to an AspectJ method call join point (Laddad, 2003). Unlike the method execution join points supported in AspectJ, AOPy's join point model does not cross method call boundaries, thereby preserving the encapsulation of the function/method. This design decision, in addition to adhering to Ossher and Tarr's notion of a join point, is arguably conceptually simpler than competing join point models. The user does not have to consider the implications of having the encapsulation of the function/method broken.

Specification of a join point in AOPy requires four values: the class of the target object, the method name, the timeframe for the woven behavior to occur (before, after, around), and the associated advice. In the current version of AOPy, this information is specified as a Python dictionary. An example join point specification is shown in listing 3.1 for a target class name Account. The aspect's advice method, aopyAssertAfter, is to be woven after the execution of the target method processTransaction.

```python
{
    "cls": "Account",
    "method": "processTransaction",
    "time": "after",
```
"advice": "aopyAssertAfter"
}

Listing 3.1. An AOPy join point declaration.

In keeping with the aspect-oriented terminology that the AspectJ team has made largely standard, a set of join points that has been grouped together is called a crosscut. In the AOPy framework, each aspect contains a single crosscut. The crosscut is implemented as a dictionary of dictionaries. The dictionary is stored in a data attribute named joinpoints (in Python, the user is able to reference an attribute by using the self.attribute_name convention). The joinpoints attribute is a data member of the aspect that it describes. It is created at runtime as part of the aspect’s individual construction process. In Python, this type of construction is done in the __init__() method.

def __init__( self):
    self.joinpoints = {
        # This joinpoint is for tracing
        # the execution of
        # AccountClass().processTransaction()
        0 :
    {
        "cls" : "AccountClass",
    }
"method": "processTransaction",
"time": "before",
"advice": "aopyPreCondition"
},

# This joinpoint is for tracing
# the execution of
# AccountClass().print_me()

1:
{
  "cls": "AccountClass",
  "method": "print_me",
  "time": "after",
  "advice": "aopyTraceAfterAdvice"
}

Listing 3.2. Join points for TraceAspect.py.

The two join points that are shown above in Listing 3.2 are part of an aspect that is used to encapsulate the tracing concern for a hypothetical Python class named AccountClass. Each individual join point is a value in a key/value pair that is an item in the attribute joinpoints (which again, is a dictionary of dictionaries). In
AOPy, the individual join points are numbered beginning with 0. This is by convention only.

The 0 join point specifies that the method that it is associated with is the `processTransaction` method in the class `AccountClass`. Its interception timeframe is before the `processTransaction` method is invoked and executed. In this example, the advice that is to be executed is the method `aopyPreCondition`.

A more detailed description of the timeframe is provided in Section 3.2.3 Method Interception. The concept of advice is explored in greater depth in Section 3.2.4 Mix-Ins.

### 3.2.2 Reflection/Introspection in Python

A general definition of reflection can be found in Brian Smith's work (1990):

"An entity's integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on and deals with its primary subject matter."

A definition that is more specific to programming languages is as follows (Bobrow, Gabriel, and White, 1993):

Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession. Introspection is the ability for a program to observe and therefore reason about its state. Intercession is the ability for a program to
modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called reflection.

Python classes and objects have descriptive metadata available in the form of attributes. Python’s support for reflective programming is based on the manipulation of these same attributes. The set of attributes that Python provides to describe its classes (\_dict\_, \_name\_, \_doc\_, \_module\_, and \_bases\_) and class instances (\_dict\_ and \_class\_) can be arbitrarily modified at runtime. An example of modifying the class of an object at runtime is shown below.

```python
>>> class ClassA:
    ...
    pass
    ...

>>> class ClassB:
    ...
    pass
    ...

>>> a = ClassA()

>>> a
<__main__.ClassA instance at 0x10a7e8>

>>> a.__class__ = ClassB

>>> a
```
Listing 3.3. Python reflection: changing an object's class from ClassA to ClassB

The aspect weaver that is at the heart of the AOPy framework takes advantage of this facility in order to intercept method invocations and to create new inheritance hierarchies. The attribute modifications necessary to implement these techniques will be described in Sections 3.3.1 Implementing the Weaver: Mix-in Creation and 3.3.2 Implementing the Weaver: Method Interception.

3.2.3 Method Interception

The second criterion described in Kiczales’s (2001) definition is a “means of identifying join points.” AOPy accomplishes this by intercepting message sends to the intended receiver object. Techniques for method intervention are known by a variety names: wrappers (Brant, 1998), interceptors (Schmidt et al, 2000), and dynamic proxies (Horstmann, 2001). For the purposes of the AOPy Project, the technique is referred to as an interceptor and the process as method interception.

The methodology used by AOPy is a modification of the algorithm described in “The End of Inheritance: Automatic Run-time Interface Building for Aggregated Objects” from the Python Cookbook Online (2002).
The key to this technique rests with Python’s ability to add new member functions and data members to classes and objects at runtime.

An Interceptor object is created. This object will eventually replace the target object in the client code. Recalling the example from the beginning of Section 3.2, `BarTargetObject` is the object that will have the aspect, `FooAspect`, woven into it. As shown in the example code, the object returned by the `weave()` method is an instance of an Interceptor object. The newly created—and woven—Interceptor object preserves the method protocol of the `BarTargetObject` object (the object that `myBar` referred to prior to the weaving process). This is accomplished by creating individual proxy methods for the methods in the class of `BarTargetObject`. Now, all messages sent to the `myBar` object are first handled by the proxy methods in the Interceptor object.

The proxy methods of the Interceptor object then execute the original methods in the `BarTargetObject` object. It is at this point that the aspect’s advice methods can be inserted before, after, or around the original method.
More details concerning the creation of a method to be executed in a specific timeframe are provided in section 3.3.2 Implementing the Weaver: Method Interception.

3.2.4 Mix-ins

The third and final criterion expressed in Kiczales’s (2001) definition is “a means of affecting implementation at join points.” In AOPy, this means is accomplished through the use of multiple inheritance, specifically in the form of mix-in class techniques (Bracha, 1990). A quote from Esterbrook (2001) points to the fact that mix-ins by themselves are a powerful separation of concerns tool:

Mix-in programming is a style of software development where units of functionality are created in a class and then mixed in with other classes. This might sound like simple inheritance at first, but a mix-in differs from a traditional class in one or more or the following ways. Often a mix-in is not the “primary” superclass of any given class, does not care what class it is used with, is used with many classes.
scattered throughout the class hierarchy and is introduced dynamically at runtime.

The use of mix-ins is a relatively well-known technique in Python. Mix-in inspired multiple inheritance techniques have even made their way into the base Python distribution. The SocketServer module makes extensive use of mix-ins. Among the examples present in the SocketServer module are the classes ThreadingTCPServer and ThreadingUDPServer. A ThreadingTCPServer multiply inherits from a ThreadingMixIn class and a TCPServer class.

In the AOPy system, the mix-in technique is used in order to make the advice of an aspect available to the target object. The technique used in AOPy is derived from a recipe—“Modifying the Class Hierarchy of an Instance”—from the Python Cookbook (Martelli and Ascher, 2002).

When using the AOPy framework, all user-defined aspects must inherit from the AOPyAspect base class. Because aspects in AOPy share much in common with mix-ins, the care that one would take in creating a mix-in class must also be exercised when creating new aspects. Issues such as unintended overriding via method name clashes must be considered.

The technique of adding a mix-in class via attribute modification is explored in greater detail in section 3.3.2 Implementing the Weaver: Mix-in Creation.
3.3 The Weaver

The primary responsibilities of the weaver are illustrated by a high-level description of the four basic steps that it carries out to create a woven object:

1. Create an object (a mix-in) that inherits from both the aspect class and the class of the target object.

2. Create an Interceptor object (it's really a Proxy) that will process and forward message sends.

3. Wrap the identified methods (i.e., the join points) of the woven object.

4. Return the Interceptor object in place of the original target object.

The Python code that implements these steps resides in a method called `weave()`. The code is shown below in Listing 3.4.

```python
def weave( self, obj, aspect):
    obj = self.createMixinAspectObject( aspect, obj, 1 )
    int = Interceptor( )
    self.interceptMethodsOf( int, obj )
    return int
```

*Listing 3.4. The AOPy weave() method*
The more detailed activities that take place in support of the two nontrivial basic steps—1 and 3—are described in the next two sections.

3.3.1 Implementing the Weaver: Mix-in Creation

The first step in creating a mix-in object—the instantiation of an object that multiply inherits from the aspect and the class of the target object—is quite straightforward. This step takes advantage of the ability to manipulate Python attributes at runtime. The necessary changes are made to the attributes related to the target class's inheritance structure: \_bases\_, \_class\_, and \_class\_.\_name\_. Additionally, the runtime object instantiation facilities that are present in the Python new module are used.

3.3.2 Implementing the Weaver: Method Interception

In AOPy, the heart of the process of method interception is the creation of a method dictionary as a private data member in the Interceptor object. This method mapping contains the original method name as a key that maps to a value that is the newly created method. In order to ensure this mapping, the class dictionary attribute (\_dict\_) of the target object is searched for all of the attributes that correspond to the methods of the class. This can be done by making sure that a particular key (attribute) in the dictionary has a value that is of types FunctionType.
Once the original method name is available to be stored as a key in the method dictionary, a new method must be created as the value. The interceptor method creation process is described in the next section.

The interceptor method creation process has two key components that result from the definition of the join point (section 2.1). First, the appropriate method to weave must be identified. Second, a method call structure that corresponds to the advice execution timeframe defined in the join point must be created.

The join point in the target object is identified by searching the class dictionary of the target object. The process of creating a new method call structure begins only when an attribute that corresponds to the method name defined in the join point is located. This means that those target object methods that are not defined in a join point are left untouched. They are merely mapped to the correct attribute name in method dictionary.

The remaining methods, those that are defined in a join point, will receive a new method call structure that contains both the original method and the advice defined in the aspect. This is accomplished by an appropriate interlacing of the advice and the original method call within a try:finally: block. The original method to be executed always appears in the try: clause. The aspect advice to be executed is tailored to the appropriate timeframe. Advice that is supposed to be executed before the target method simply precedes the original method in the try: clause. Advice that is supposed to be executed after the target method is placed in
the `finally:` clause. Advice that is to be executed around the target method appears in both clauses.

### 3.4 Completing the Weave

Finally, the newly created interceptor object is returned by the weaving process. From this point on, when the client code sends a message to the method identified in the join point, the new method call structure—containing calls to the appropriate advice—is executed.

### 3.5 Experiment Design

The experimental phase of the AOPy project was implemented in a two-experiment sequence. Each experiment was conducted at the end of workshop that introduced aspect-oriented programming to the experiment’s subjects.

The thinking-aloud methodology was used to produce data for each experiment. The thinking-aloud methodology is described in detail in section 3.6.

At the suggestion of researchers familiar with the thinking-aloud methodology (Borning, 2002; Cook, 2003; Herrmann, 2003), a preliminary experiment, described in sections 3.5.1 and 3.5.2, was performed in an open-ended fashion.

The second experiment, outlined in section 3.6.3, was heavily dependent on the analysis of the outcomes of the first experiment. Hence, the first experiment can
be considered to be part of the background research and planning for conducting the main thinking-aloud experiment.

3.5.1 CS 180: Programming Languages and Tools Seminar

The CS 180 Programming Languages and Tools Seminar was taught at Knox College in the fall of 2003. The seminar concluded with an experiment. This experiment was a preliminary experiment that was to be used to inform and focus the planning and organization of the main experiment (Dechow, 2004a).

In order to establish a baseline for the study, the participants completed a survey of their relevant skills and experiences. In the next step, the subjects—a class of seven undergraduate students—participated in the aspect-oriented programming workshop.

The workshop material was presented in three sections: 1) an introduction to Python, 2) web applications development, and 3) aspect-oriented programming. Each section consisted of approximately six hours of instruction. The overarching theme for the workshop was separation of concerns. This theme was emphasized by demonstrating a progression of techniques for separating concerns: from functions to classes to multiple inheritance (using mix-ins). This progression culminated in the use of advanced separation of concerns techniques (aspects) using the Pythius aspect-oriented extension module for Python.
All subjects in the preliminary experiment were undergraduate computer science majors. While subject background is of less consequence in this initial discovery phase using grounded theory, the educational exposure—in terms of computer science coursework—is similar across subjects and summarized in the table below. None of the subjects had any significant prior industry experience.

<table>
<thead>
<tr>
<th>Subject #</th>
<th>Academic Experience</th>
<th>Industry Experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>141, 142, 201, 205, 206, 208</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>Opted out of experiment</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>141, 142, 201, 205, 206, 395</td>
<td>None</td>
</tr>
<tr>
<td>4</td>
<td>141, 142, 201, 205, 308, NET, SE, Java</td>
<td>Summer Intern (2)</td>
</tr>
<tr>
<td>5</td>
<td>141, 142, 201, 205, 206, 208&amp;AI, OS, SE, DBMS</td>
<td>Summer Intern (3)</td>
</tr>
<tr>
<td>6</td>
<td>141, 142, 205</td>
<td>None</td>
</tr>
<tr>
<td>7</td>
<td>141, 142, 201, 206</td>
<td>None</td>
</tr>
<tr>
<td>8</td>
<td>141, 142, 205</td>
<td>None</td>
</tr>
</tbody>
</table>

*Table 3.1 Subject education and experience.*

The superset of the courses taken by the subjects yielded a total of fourteen possible computer science courses. The course numbers are matched with the corresponding course title in the table below.

3.5.2 CS 180 Thinking-Aloud Experiment and Outcomes

Initially, it was not known exactly what would be analyzed in the preliminary experiment. Several researchers suggested, as described above in Section 3.5, that
the first group of subjects to participate in this experiment should be evaluated in an open-ended fashion. As a result of this approach, the subjects were allowed to interpret the experimental tasks for themselves. Additionally, they were free to solve the problems in the manner with which they felt most comfortable: on the computer, pen and paper, or by straightforward discussion.

The outcomes from this first group of subjects could then be used to shape the experiment diagnostic for the next group of test subjects. This process allows areas of interest to emerge from experimentation within the defined area of the research project. This process seems to be a reasonable and prudent course of action in planning a main experiment for a project such as the AOPy project.

As was mentioned in Chapter 1, the core concepts of an aspect-oriented language are defined as “a join point model, a means of identifying join points, and a means of affecting implementation at join points” (Kiczales et al., 2001). The ability to modularize these concepts in the form of a language construct—the aspect—is at the heart of aspect-oriented programming.

Two separate questions were investigated in the experiment. Each question relates to a single, different class of Python code from the Snakelets web server.

In the first question, the subjects are asked to apply what they have learned in order to diagnose the behavior of the Snakelet class TemplateProcessor (contained in the templateProcessor module). This diagnostic task is
analogous to instrumenting the class. The instrumentation concern embodies the primary aspect-oriented concepts

For the second question, the participants are instructed to examine a different Snakelets class: ThreadingServer (contained in the server module). The intent of this question is to have the subjects analyze the ThreadingServer class for concerns that can be separated out into aspects. This activity is clearly a more abstract task, and one that gets to the heart of modularization of concerns.

For this specific assessment in the preliminary experiment (performed on 6 Nov 2003), the seven students were divided into two groups of two, and one group of three. The students were given as much time as they needed to complete the tasks. Each session was videotaped. Additionally, significant verbalizations were recorded by hand as notes.

While the subjects were successfully able to master the syntactics of aspect-oriented programming, the more important ideas of “What is a concern?” and “How do I modularize a cross cutting concerns?” completely escaped them.

The first of these outcomes—the subjects’ ability to successfully use the aspect language—was confirmed by the fact that three out of three groups successfully satisfied the requirements of the problem’s instructions. In the first problem in the CS 180 experiment, each group of subjects demonstrated the ability to create a joinpoint that successfully defined the correct point in the program’s execution. Further, each group also formulated advice methods that would have
inserted the desired new behavior at the aforementioned joinpoint. The success of the groups in the first case could be attributed to the fact that they were adding a new concern.

In the second problem, none of the three groups was able to identify or articulate an appropriate concern. This problem asked subjects to identify and extant concerns. Subjects were unable to articulate an already-represented concern in a new context.

The second problem, therefore, provided the sort of unpredicted evidence that grounded theory fosters. The failure of all students to identify concerns provided an area of interest to further investigate and, therefore, became the driving factor in the formulation of the main experiment because of its relevance to real-world software development situations. The second problem more accurately represents the process of maintaining an existing software product and enhancing it with aspect-oriented techniques.

Additionally, the second problem had elements of software analysis that were not present in the first problem. In the first problem, the subjects were provided a complete analysis of the problem. This analysis involved a description of a new software concern—an instrumentation or tracing concern—to add to an existing Python program. This new software concern was a crosscutting concern. They were then asked to follow through with the design and implementation, if they could.
The second problem, on the other hand, tasked the subjects with three phases—analysis, design, and implementation. In the second problem, the subjects were faced with conceptualizing all of the extant concerns in the software system and deciding which one was a crosscutting concern. So, instead of being given a crosscutting concern to add to a program, subjects were given a program and asked to recognize a crosscutting concern that existed within it but that was not defined by the problem description. Because all subjects' success between the first and second problem differed markedly, this additional burden of analysis and identification of crosscutting concerns pointed the way to the second experiment.

These outcomes led the development of a second experiment—the main experiment for the research project—that focused on these issues. No salient changes were made to the methodology, but changes were made in the experiment design. These changes are described below in section 3.5.3.

3.5.3 CS 582 Aspect-Oriented Workshop

The CS 582 Aspect-Oriented Workshop preceded the main thinking-aloud experiment. Eight computer science graduate students, whose experiences are summarized in table 1 in section 5.1.4, were involved in this workshop and the subsequent experiment. The subjects were volunteers from the graduate object-oriented programming course—CS 582—here at OSU.
The workshop started with an introduction to the Python programming language. This introduction was done independently by each student before the group workshop began. The recommendation for this introduction was the first four chapters of www.diveintopython.org.

The subjects who participated in the CS 582 AOP workshop did so while concurrently enrolled in an object-oriented programming course at Oregon State University: CS 582. The instruction portion of the workshop took place during the 9th and 10th weeks of the winter 2004 term. The experiment was conducted during finals week, ensuring that all subjects had completed a course in object-oriented programming at the time of the AOPy project’s main experiment.

Although each subject brings a differing level of preparation and a disparate set of experiences to the experiment, this is in keeping with what one expects to find in an industrial software team. Once someone is a member of a team, whether or not she earned an A in operating systems becomes irrelevant. Each team member has the same opportunity and obligation to participate.

Two background factors are present in the subjects who participated in the experiment: each had an identical eight weeks of exposure to OOP (CS 582) and the two-week AOP workshop. The content covered during the eight weeks of CS582 prior to the beginning of the AOP workshop is described in the course schedule for CS 582 (see syllabus in appendix 2). Again, this situation closely follows what one could expect to find in an industrial setting. For instance, new information or
techniques might be presented to development teams in a one- or two-day seminar, after which teams would work as a group to develop skill in using that information and applying those techniques.

The idea of separation of concerns and the techniques of aspect-oriented programming were introduced in the group workshop. This portion of the workshop consisted of six two-hour meetings during the first two weeks of March. Material was introduced by lecture.

The workshop culminated with pairs of student subjects solving two separation-of-concerns problems in the thinking-aloud experiment. Although each experiment session was originally scheduled for two hours and the experiment was designed with this timeframe in mind, no group was ever forced to terminate a thinking-aloud session because of time concerns.

3.5.4 CS 582 Thinking-Aloud Experiment

During the course of the preparatory workshop, the process of participating in a thinking-aloud experiment was explained on two occasions: the first and last nights of the workshop.

On each occasion, the thinking-aloud methodology was characterized as a technique to facilitate the observation of a problem-solving session. The workshop students were told that they would be given a problem (or problems) to solve. As the subjects worked through the problems, they would need to verbalize their thoughts,
questions, and decisions. Hence, they would need to think aloud. The subjects were
told to think of the experiment as a collaborative design session that would be
videotaped. Finally, it was further explained that the experimental portion of the
workshop would be conducted using a modification of the thinking-aloud technique.
This modification is known as a constructive interaction experiment.

The benefits of the constructive interaction between multiple subjects who
cooperate on the experimental task is that this format reduces the involvement of the
experiment observer and that it better reproduces the collaborative environment that
exists in software development shops. The subjects were encouraged to work
together in pairs or trios during the computer-based sections of the workshop. These
hands-on problem sessions took place at the conclusion of every lecture during the
workshop.

The workshop participants were not involved in a ‘mock’ thinking-aloud
session prior to the actual experiment. The first time that the experiment subjects
participated—in the context of this experiment—in a thinking-aloud study was
during the experiment.

The two-problem set developed for the experiment is based on well-known
design pattern implementations: Chain-of-Responsibility and Visitor. The choice of
basing this part of the experiment materials on a pair of well-documented design
patterns was done for a variety of reasons. The chosen patterns were described in
literature discussing patterns and scripting languages (Savikko, 1997). As is
mentioned in section 1.2.1, a great deal of research into the area of design patterns and aspect-oriented programming is currently ongoing. The use of design patterns give a rich, natural language description of the concerns involved in the problem space that can be used in the analysis of the subjects’ verbal outcomes.

As was mentioned previously in Chapter 1, the connection between design patterns and aspect-oriented programming is an active area of research inquiry. While the notion of design patterns may have been somewhat radical ten years ago at the time (1994) of the publication of the GoF book, today patterns are well established in the practice of software development. As such, the Chain-of-responsibility and Visitor design patterns are well suited to providing a real-world basis for the experiment problem set.

The two specific patterns were excellent choices for a wide variety of reasons. Beyond the nature of design patterns as descriptions of acknowledged real-world problems, the visitor pattern is the subject of significant research in the aspect-oriented world and the chain-of-responsibility pattern is particularly relevant in the context of the implementation of graphical user interfaces, one of the original ‘killer-applications’ for object-oriented programming.

Every effort was made to preserve the conceptual integrity of the chosen design patterns. As a result, the source code that was presented to the subjects consisted of pattern representations that implemented only those behaviors and characteristics that define the pattern. This limiting of variety—or this focusing on
defining behaviors and characteristics—was done in order to make the experiment problems more comprehensible to the subjects and more amenable to analysis of focused results.

Finally, the experiment subjects had no workshop exposure to the chosen experiment design patterns. However, they were exposed to Proxy, Observer, and an aspect-oriented implementation of Observer as a part of the workshop’s course materials.

Each of the two problems consisted of a Python module implementing a version of the particular design pattern and a set of instructions for analyzing the module.

The instructions for each problem were to simply find and describe all of the concerns that were present. The exact language used to begin each problem statement is as follows:

Analyze the software by describing all of the concerns that are present in the module. Are any of the concerns crosscutting (tangled, orthogonal to the primary concerns, etc.)? If so, based on the techniques that were covered in the workshop, describe alternative solutions for separating the concerns.

The problem descriptions (in their entirety, as they were given to the subjects) are presented in the appendices.

In general, the experiment design and methodologies, ultimately, were designed and adapted to test for compatibility of aspect-oriented programming with object-oriented programming in the main thinking-aloud experiment. Discussion of
compatibility in the results and analysis are discussed in Chapters 5 and 6, respectively. The specific methodologies employed for organizing and observing both thinking-aloud experiments—CS 180 and CS 582—are described in section 3.6.

3.6 Experiment Methodology: Grounded Theory & Verbal Reports

The guiding methodologies for the preliminary and main experiment sequence are grounded theory and verbal reports generated by thinking aloud.

3.6.1 Grounded Theory as Methodology

Software development is a social phenomenon. In Chapter 22: Managing People of his text Software Engineering, Ian Sommerville has the following to say:

The people working in a software organization are its greatest assets. They represent intellectual capital [... M]y discussion of management [is based] on cognitive and social factors rather that any, currently fashionable, management theory. Software engineering is a cognitive and social activity, so these factors are important in developing an understanding of how people write software.

Because software development has this overriding social component, a methodology using the grounded theory (Glaser and Strauss, 1967) technique is useful in accounting for the role of the subject in the relationships between object-oriented programming and aspect-oriented programming. A grounded theory approach to experiment design and methodology fosters understanding of the assumptions of
compatibility that developers make between the two techniques. Grounded theory also allows the researcher to understand the experiment as a process.

In “Grounded Theory as Scientific Method,” Brian Haig (1995) gives the following summary of the grounded theory methodology:

Grounded theory is typically presented as an approach to doing qualitative research, in that its procedures are neither statistical, nor quantitative in some other way. Grounded theory research begins by focusing on an area of study and gathers data from a variety of sources, including interviews and field observations. Once gathered, the data are analyzed using coding and theoretical sampling procedures. When this is done, theories are generated, with the help of interpretive procedures, before being finally written up and presented.

Ultimately, grounded theory is a framework both for experiment design and for analysis of results because grounded theory defines experimental research as a process. In other words, grounded theory is an analysis of an experimental process—a narrative of sorts—instead of merely an analysis of the product of the experimental process. The manner in which the data is generated is as important as, if not more important than, the data itself.

In this research project, for example, data were gathered in the preliminary experiment—CS180—from both surveys and an open-ended, thinking-aloud experiment involving two problems. Once gathered, these data were loosely coded according to whether subjects used appropriate aspect-oriented terminology and articulated understanding of modularity and crosscutting concerns. The process of the preliminary experiment generated insights that were used to plan the main
experiment. So, though the preliminary experiment produced what might be considered open-ended results that were informally analyzed, this application of grounded theory is also part of the design and methodology for the main experiment—CS582—that produced the research results that are described in Chapter 5 and analyzed in Chapter 6.

3.6.2 Verbal Reports and Thinking Aloud

While grounded theory provides a unified methodology both for experiment design and for qualitative data analysis, another group of methodological approaches is necessary to generate the experimental data.

Verbal reports and protocols are experimental techniques that seek to describe the cognitive processes of participants (Ericsson and Simon, 1993). The participants in this type of a study verbally describe what they are thinking about as they attempt to perform a problem-solving task. Each participant's verbalization is recorded in one or more forms: audio-taping, videotaping, note-taking, etc. The thinking-aloud methodology and simplified thinking-aloud (Nielsen, 1989) are among the more common examples of verbal protocols. Their use is well established in computer science and software engineering.

Thinking-aloud experiments are based on three components: 1) an evaluation; 2) user participation; and 3) a diagnostic and summary ("Thinking Aloud," Usor: A Collection of User Oriented Methods, 2003). Thinking-aloud studies result in
qualitative data obtained via observing the subjects participation in a problem-solving session. It is important to note that thinking-aloud is not a statistical methodology. A slight modification to the thinking-aloud methodology involves using paired subjects in the diagnostic. This method is known as a constructive interaction session (O'Malley, Draper, and Riley, 1984). This modification is very beneficial when dealing with subjects from a wide variety of skill backgrounds. It allows for the testing of larger numbers of subjects in an allotted period of time.

In this research project, thinking-aloud was used with pairs of subjects in both the preliminary and main experiments. All three components of thinking-aloud methodology were employed. Subjects completed an initial survey and workshop. Subjects participated in videotaped problem-solving sessions during which the researcher also took field notes. The diagnostic and summary aspects of thinking-aloud involve the coding of the qualitative results (see Chapter 5) and the analysis and summary of findings (see Chapter 6).

3.6.3 Thinking-Aloud Constraints

When thinking-aloud methodology—or any participant-observer research methodology—is part of the data collection, sessions should not be burdensome for the subjects. Instead, results can be considered more valid when the thinking-aloud session is of reasonable length and convenient in time and place for the subjects. In addition, subjects should not be under pressure beyond the experiment's boundaries.
In the case of the AOPy project, these considerations meant that, though subjects were drawn from academic classes, their performance in the experiment and my observations or notes were not part of that class. Likewise, their performance in that class did not influence my observations and interpretations in the research. Also, the sessions should be kept to a comfortable length—in this project, roughly two hours—and the experiment was designed with that in mind. The sequencing of a preliminary experiment and a main experiment also gives the researcher practice so that the results of the main experiment are even less likely to be affected by any lack in the researcher’s experience with thinking-aloud methodology.

When collecting data through a method such as thinking aloud, the researcher must consider a variety of possibilities for designing the experiment, including the setting and problems. I considered the amount of structure the sessions would have, the benefits of informality between subjects, the situations in which the researcher is best unobtrusive and best interjective, the documentation through both videotape and field notes, the difference between conversational and interrogative sessions, the pointedness of experiment problems, and the benefits of small teams of subjects. In addition, I consciously designed the experiments as exploratory, rather than confirmatory of a narrow premise so that the qualities of thinking-aloud methodology could best be exploited.
CHAPTER 4 CASE STUDY: THE OBSERVER DESIGN PATTERN

4.1 Introduction

As stated in Chapter 1, this research project works from the following premises: that object-orientation is the dominant programming methodology, that object-orientation apprehends some real-world things less well than it does others, and that aspect-orientation has been specifically designed to encapsulate well those scattered concerns that objects poorly encapsulate (Elrad et al., 2001). The case study of the Observer design pattern, as documented in this chapter, establishes the validity of design patterns as a context for the investigation of aspect-orientation. As was described in Chapter 1, in qualitative data analysis, the validity of research is established and framed by the context of the questions that are asked.

As was mentioned in the introductory chapter, the application of advanced separation of concerns techniques to design patterns (Noda, 2001a), in order to improve the modularity and composibility of pattern implementations (Hannenmann, 2002), is an active research area. Additionally, other investigators (Hanenberg and Costanza, 2002) have been studying aspect-oriented patterns in their own right. The case study documented in this chapter, then, establishes a valid context and, by extension, frames the research question.
As a secondary goal, this chapter serves as a non-trivial documentation of how to use the AOPy framework.

Design patterns (Alexander et al., 1977; Gabriel, 1996) are literary descriptions of solutions for a particular problem in a specific context. In the realm of software development, design patterns are typically—though not specifically—concerned with the architecture of object-oriented systems. The emphasis on object-oriented software began with the so-called Gang-of-Four (GoF) book (Gamma et al., 1995), which is subtitled: Elements of Reusable Object-Oriented Software.

The GoF book suggests the following design idiom: “Favor object composition over class inheritance.” Unfortunately, some very useful design patterns show just how difficult this can be in practice. One such pattern is Observer (Gamma et al., 1995).

In the GoF book, the purpose of Observer is defined as follows: “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.” This makes Observer particularly useful in systems such as user interfaces.

As will be shown, the standard inheritance-based GoF implementation of the Observer pattern tightly couples the application code to the design pattern code.

A case study involving the implementation of the Observer design pattern using the AOPy framework provides an excellent vehicle for further investigation of these issues. The rationale for studying Observer in particular is straightforward. As
was shown in Hannenmann’s OOPSLA paper (2002), the notification relationship between an Observable (Subject in traditional GoF design patterns parlance) and its Observer can be characterized as a crosscutting concern.

Evidence of these crosscutting concerns can be seen in the implementation of the Observer design pattern in Listing 4.2 (section 4.1.2). The setter methods of class Point, setX() and setY() contain the tangled notification concern that is embedded in the two related methods:

\[
x\_coordNotifier.notifyObservers()
\]

and

\[
y\_coordNotifier.notifyObservers().
\]

By applying aspect-oriented techniques, the tangled notification concern is removed from the Point and Screen classes. The newly untangled concern representing the implementation specifics of the Observer design pattern is encapsulated in the ObserverProtocol aspect. The result is that each of the components involved in the design pattern is made more modular, i.e., “organized into more or less independent units” (Brown, 1997).

### 4.2 Observer Design Pattern

As was described above, Observer is commonly used in user interfaces. As an example, the Observer pattern is the basis for the Model-View relationship in a
Model-View-Controller architecture (Reenskaug, 1979). It also underpins the Publisher-Subscriber pattern (Buschmann, 1996).

A UML class diagram—adapted from an OMT diagram presented in Gamma et al., (1995)—describing the relationships of the classes involved in the observer pattern is presented in Figure 4.1 (Data & Object Factory, 2002):

![Figure 4.1. Observer Class Diagram](image-url)
4.2.1 Observer in Python

The standard implementation of the observer design pattern in Python is described below. This implementation is modeled on a solution found in the online draft of Thinking in Patterns (Eckel, 2003). The Thinking in Patterns book implements many of the 23 GoF patterns in the Python scripting language.

Object instances created from two classes—Observer and Observable—are used to implement the observer pattern in Python. The source code for the two classes—Observer and Observable—is shown at the end of this subsection in Listing 4.1.

As is shown in Table 4.1, the method names used in the Python implementation of Observer are slightly modified from the names that are described in the GoF book and the implementations that it has inspired.

<table>
<thead>
<tr>
<th>Class</th>
<th>GoF version</th>
<th>Python version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observer</td>
<td>Update()</td>
<td>update()</td>
</tr>
<tr>
<td>Observable</td>
<td>Attach()</td>
<td>addObserver()</td>
</tr>
<tr>
<td></td>
<td>Detach()</td>
<td>deleteObserver()</td>
</tr>
<tr>
<td></td>
<td>Notify()</td>
<td>notifyObservers()</td>
</tr>
</tbody>
</table>

Table 4.1. GoF-to-Python Method Name Mapping.
The primary responsibility of the Observer class is to provide a means for change notification. This behavior is encoded in the update() method.

The Observable class has two main responsibilities: 1) to maintain, in the attribute self.obs, a collection—in this case a list structure—of all of the interested Observer(s); and 2) to provide a set of behaviors—notifyObservers(), addObserver(), and deleteObserver()—for manipulating the Observer(s).

To complete the implementation of the Observer design pattern, it is necessary to implement ConcreteObserver and ConcreteObservable classes (as is shown Figure 4.1). These classes are application specific, and they will be described in the example section 4.1.2.

```python
#: util:Observer.py

# Class support for "observer" pattern.
from Synchronization import *

class Observer:
    def update(observable, arg):
        '''Called when the observed object is modified. You call an Observable object's notifyObservers method to notify all the
```
object's observers of the change.''

pass

class Observable(Synchronization):
    def __init__(self):
        self.obs = []
        self.changed = 0
        Synchronization.__init__(self)

    def addObserver(self, observer):
        if observer not in self.obs:
            self.obs.append(observer)

    def deleteObserver(self, observer):
        self.obs.remove(observer)

    def notifyObservers(self, arg = None):
        '''If 'changed' indicates that this object has changed, notify all its observers, then call clearChanged(). Each observer has its update() called with two arguments: this
observable object and the generic 'arg'.

```
self.mutex.acquire()
try:
    if not self.changed: return
    # Make a local copy in case of synchronous
    # additions of observers:
    localArray = self.obs[:]
    self.clearChanged()
finally:
    self.mutex.release()

    # Updating is not required to be synchronized:
    for observer in localArray:
        observer.update(self, arg)
```

def deleteObservers(self): self.obs = []
def setChanged(self): self.changed = 1
def clearChanged(self): self.changed = 0
def hasChanged(self): return self.changed
def countObservers(self): return len(self.obs)
4.2.2 Observer Example

The application domain for this case study is a simplified representation of a computer screen and items (points) that can be drawn on it. The Observer design pattern is a natural choice for modeling the notification process that results when location changes are made to points.

The example demonstrates how changes to the x or y attributes of the Point class are communicated to the Screen (Observer). The Point and Screen classes make use of implementations of ConcreteObservable and ConcreteObserver, respectively, as the Pythonic equivalent of inner classes. A brief description of all of the components involved in the process for handling notifications related to changes to self.x is provided below.

Firstly, it is obvious that the inner class—XcoordNotifier—is the implementation of ConcreteObservable. Most importantly, it contains an implementation of notifyObservers(). This method invokes the same method
in its base class (Observable). The base class implementation of notifyObservers() iterates through its collection of registered Observer(s) and directly invokes the update() method on each Observer. The actual event handling code is implemented in the update() method of the ConcreteObservable (in this example, a Point).

The complete code is provided below in Listing 4.2. At the end of this section, following Listing 4.2, is a summary of the messages exchanged when a change is made to the x coordinate.

```python
#: Point—observed—as in Java Subject interface
#: Screen(s)—observers—as in Java Observer interface
#: Demonstration of GoF Observer Design Pattern

import sys
from Observer import Observer, Observable

class Point:
    def __init__(self, _x, _y, color):
        self.x = _x
        self.y = _y
```
self.color = color

self.x_coordNotifier = Point.XCoordNotifier(self)

self.y_coordNotifier = Point.YCoordNotifier(self)

def setX(self, _x): # x-coord change
    self.x = _x

    self.x_coordNotifier.notifyObservers( self.x )

def setY(self, _y): # y-coord change
    self.y = _y

    self.y_coordNotifier.notifyObservers( self.y )

class XCoordNotifier(Observable):
    def __init__(self, outer):
        Observable.__init__(self)

        self.outer = outer

    def notifyObservers(self, x_coord):
        self.setChanged()
class YCoordNotifier(Observable):
    def __init__(self, outer):
        Observable.__init__(self)
        self.outer = outer
        def notifyObservers(self, y_coord):
            self.setChanged()
            Observable.notifyObservers(self, y_coord)

class Screen:
    def __init__(self, name):
        self.name = name
        self.x_coordObserver = Screen.XCoordObserver(self)
        self.y_coordObserver = Screen.YCoordObserver(self)

        # An inner class for observing x-coord
        class XCoordObserver(Observer):
def __init__(self, outer):
    self.outer = outer

def update(self, observable, arg):
    print "Screen: " + self.outer.name +
    " notified of change to x-coord: " +
    str(arg)

# Another inner class for y-coord
class YCoordObserver(Observer):
    def __init__(self, outer):
        self.outer = outer
    def update(self, observable, arg):
        print "Screen " + self.outer.name +
        " notified of change to y-coord: " +
        str(arg)

if __name__ == "__main__":
    print "Creating observers (s1, s2, s3) & observed (pp)"
    pp = Point(5, 5, "blue")
    s1 = Screen("s1")  # s1 observes coords
#s2 = Screen( "s2" )  # s2 also observes

coords

#s3 = Screen( "s3" )  # s3 observes

colors

print "Creating observing relationships"

pp.x_coordNotifier.addObserver(s1.x_coordObserver)

pp.y_coordNotifier.addObserver(s1.y_coordObserver)

print "Change that interest the observers"

pp.setX( 2 )

pp.setY( 4 )

# clean-up and go home

pp.x_coordNotifier.deleteObservers()

pp.y_coordNotifier.deleteObservers()

print "Changes that went unnoticed"

pp.setX( 2 )
Listing 4.2. Point and Screen example using the Observer pattern.

The entire messaging sequence is in listing 4.3, which describes resulting method/function invocations when a change is made to the x attribute of a Point.

1. pp.setx(2)
2. self.x_coordNotifier.notifyObservers(self.x)
3. Observable.notifyObservers(self, x_coord)
4. observer.update(self, arg)
5. print "Confirmation of received update..."

An identical process is used to handle notifications related to changes in self.y.

4.3 The Observer Design Pattern Using Aspect-Oriented Techniques

The case study implementation of the Observer design pattern is modeled after the ObserverProtocol solution described in Hannenmann’s paper (2002) and the book Mastering AspectJ (Laddad, 2003). However, some significant changes have been made. In this solution, no specific attempt was made to improve the compositibility of the solution. All of the improvements offered by this solution were designed to enhance modularity. This is reflected by the implementation of advice specific to the x- and y-coordinates in the body of the aspect. A more composable
solution might have abstracted all generic behaviors into a new aspect and then have used that aspect in an inheritance relationship. This would be more in keeping with the ObserverProtocol implementation described in the above references.

The Python code shown below in Listing 4.3 demonstrates how it is possible to increase the solution’s overall modularity by encapsulating the x- and y-coordinate change notification code in the ObserverAspect.

```python
import AOPy
from Observer import Observer, Observable

class ObserverAspect( AOPy.Aspect ):
    def __init__( self ):
        self.joinpoints = {
            # This joinpoint is for observing the change in x-coord
            0 : {
                "cls":"Point",
                "method":"setX",
                "time":"after",
                "advice":"aopyAfterSetX"
            }
        }
```
self.x_coordNotifier = ObserverProtocol.XCoordNotifier(self)

# Make sure to init the Super class
AOPy.Aspect.__init__(self)

def aopyAfterSetX(self, *moreArgs, **evenMoreArgs):
    self.x_coordNotifier.notifyObservers(moreArgs[0])

class XCoordNotifier(Observable):
    def __init__(self, outer):
        Observable.__init__(self)
        self.outer = outer

    def notifyObservers(self, x_coord):
        self.setChanged()
        Observable.notifyObservers(self, x_coord)
# An inner class for observing x-coord
class XCoordObserver(Observer):
    def __init__(self, outer, name):
        self.outer = outer
        self.name = name
    def update(self, observable, arg):
        print self.name, "x-coord change: " + str(arg)

    def addObserver(self, screen):
        screen.x_coordObserver =
            ObserverProtocol.XCoordObserver(
                self, screen.name)
        self.x_coordNotifier.addObserver(
            screen.x_coordObserver)

    def deleteObservers(self):
        self.x_coordNotifier.deleteObservers()

As Listing 4.3 shows, now both of the inner class objects instantiated from the Observable and the Observer are encapsulated in the aspect. By modularizing this concern in an aspect, the tangled nature of the concern’s representation in previous solution has been resolved.

4.3.1 Features of the ObserverAspect

As was described in section 2.1, a crosscut in AOPy is implemented as a dictionary of dictionaries. A join point is represented by a single dictionary entry in that data structure. For the ObserverAspect, a single join point is used for demonstration purposes:

```json
{
    "cls":"Point",
    "method":"setX",
    "time":"after",
    "advice":"aopyAfterSetX"
}
```

In this example join point, the weaver is instructed to identify the method `setX` as the method call of interest. Anytime that the a change to the x-coordinate is
made by invoking the `setX` method, the weaver has inserted advice—
aopyAfterSetX—to be executed after the `setX` method executes.

4.3.2 Aspect-Oriented Observer Example

For this aspect-oriented Observer example, the aspect-oriented versions of
the `Point` (shown below in Listing 4.4) and `Screen` (shown below in Listing 4.5)
classes are described first.

```python
class Point:
    def __init__(self, _x, _y, color):
        self.x = _x
        self.y = _y

    def setX(self, _x):  # x-coord change
        self.x = _x

    def setY(self, _y):  # y-coord change
        self.y = _y

Listing 4.4. Point example using AOP Observer pattern.
```
class Screen:
    def __init__(self, name):
        self.name = name

    def display(self, somestring):
        print self.name + " : " + somestring

Listing 4.5. Screen example using AOP Observer pattern.

The most fruitful investigation of the two new classes comes by directly comparing them with the Point and Screen classes presented in Listing 4.2. The most obvious difference is dramatic reduction in size of the Point and Screen classes in the aspect-oriented version. Clearly, the two new classes are easier to comprehend than their earlier versions.

Additionally, by removing the tangled implementation concern associated with the Observer pattern code from Point and Screen classes, modularity and encapsulation in all of the components is improved.

Below, in Listing 4.6, an example of using the new Point and Screen classes with the ObserverAspect is shown.

    pp = Point.Point( 5, 5, "blue" )
# Initiate weaving...

myWeaver = AOPy.AspectWeaver()

pp = myWeaver.weave(pp, AOPyObserverAspect.ObserverAspect)

sl = Screen.Screen("sl") # sl observes

print "Creating observing relationships"

pp.addObserver(sl)

print "Changes that interest the observers"

pp.setX(2)

pp.setX(44)

# clean-up and go home

pp.deleteObservers()

pp.setX(55)

Listing 4.6. Using the ObserverAspect in an example.
The entire messaging sequence for event notification in the aspect-oriented solution can be seen in the following list. The list describes the method/advice invocations that result when an update is made to the x attribute of a Point object (which is referred to by the variable pp in this example):

1. pp.setX( 2 )
2. self.aopyAfterSetX( ) # THIS IS WOVEN ADVICE
3. self.x_coordNotifier.notifyObservers( self.x )
4. observer.update(self, arg)
5. print self.name, "x-coord change: " + str( arg )

When compared with the messaging sequence for the GoF implementation of the Observer design pattern (in section 4.1.2), it is only in the second method invocation (highlighted in bold) that a difference in the sequence appears. This extra method call is the result of weaving the aspect’s advice into the target object (here, a Point object).

An identical process to handle notifications related to changes in self.y is a straightforward modification of the ObserverAspect.
4.4 Discussion

Again, it is clear when making a direct comparison between the original Point and Screen classes and the aspect-oriented version of these classes that the aspect-oriented versions offer a cleaner implementation of the behaviors that Point(s) and Screen(s) are intended to model. The enhanced simplicity that results by removing the code that implements the Observer design pattern makes the usage of the Point and Screen classes much more obvious. The orthogonal representation of the notification concern that is present in the GoF implementation of the Observer design pattern in Python—in the form of the XCoordNotifier, YcoordNotifier, XCoordObserver, and YcoordObserver inner classes—is a complication that distracts from the primacy of the concerns represented by the Point and Screen classes.

4.5 Related Work

Of most obvious interest to Python developers is the Pythius aspect-oriented toolkit (Pythius, 2003). Pythius’s approach differs greatly from that of AOPy. Pythius’s support for aspect-orientation is invasive because of its use of the Python _metaclass attribute (Martelli, 2003) in the target class. In general, aspect-oriented systems view making source level changes in the target class as an unacceptable implementation approach.
In a similar vein to the AOPy system, Böllert’s early weaver description (1999) was also an inheritance-based system. Böllert’s description of the AOP/ST system motivated the initial design (adding aspect-specific code by the use of inheritance) and goals (non-invasive) of the AOPy system. The AOP/ST was implemented in Smalltalk. As such, it did not use mix-in classes to weave new behaviors into application objects. AOP/ST used a system that Böllert referred to as dynamic-method-rewriting.

4.6 Conclusion

The aspect-oriented infrastructure that is a part of the AOPy Project is an attempt to extend the Python scripting language by means of an ordinary Python module. AOPy’s support for aspect-oriented programming is based on method call join points, method interception, and mix-in programming techniques.

Following the completion of the software case study that is documented in this chapter and that demonstrated the validity of design patterns as an area of inquiry and documented how to use AOPy, the AOPy framework was used in an experiment to determine the usefulness of aspect-orientation in the context of scripting languages. The experiment was performed in March 2004.
5.1 Introduction

The following chart displays the process of determining results through the experimental process:

- **field notes are taken during the thinking-aloud sessions to characterize interactions**
- **points of interest in field notes are determined by looking for rich combination of AO and OO terminology**
- **points in notes are analyzed in videotapes to determine appropriate vs. inappropriate AO usage**
- **pinpointed mention sequences (data that indicate phenomena of interest) are coded and categorized**
- **category definitions are refined as compatibility & noncompatibility using specific characteristics**

*Figure 5.1: Data Collection Process*
The key idea that underlies the theories developed in this dissertation is the notion that the natural language that developers, in this case subjects in the experiment, use during the problem-solving session indicates the incorporation of aspect-oriented ideas, terminology, and techniques. Because natural, spoken language is relatively predictable and consistent within the problem-solving session and also spontaneously articulated, it provides key words or phrases that serve as indicators of level of ease of compatibility between aspect-orientation and object-orientation in software design. When aspect-orientation lacks compatibility, the natural language used is either unnatural, inconsistent, or nonexistent.

As an example scenario, consider the analysis and design discussions that will take place in software development shops that extend object-oriented systems with aspect-oriented practices. These groups of developers will inevitably need to account for both systems of design and decomposition simultaneously. That is to say that design sessions will need to roam freely from discussion of primary concerns (object/components) to crosscutting concerns (aspects). Discussion of these design problems will be best facilitated when the discussion proceeds back and forth between these ideas in a manner that suggests there are no glaring incompatibilities that impede the conversations of either design system. The question, then, is whether the ideas and techniques of aspect-oriented programming are a natural, easy, or good fit for the ideas and techniques of which object-oriented developers are already aware.
In other words, is aspect-orientation compatible with object-orientation? Compatibility implies that no discontinuities or disconnects occur to thwart extension of object-orientation by using aspect-orientation. Early literature on aspect-orientation indicates that object-orientation was not a prerequisite for aspect-orientation and that aspect-orientation could work equally well with, for instance, functional programming languages (de Meuter, 1997; de Moor, 1999). Yet, the reality is that the bulk of aspect-oriented research and practice has focused upon extensions to and enhancements of existing object-oriented programming languages. This reality dictates that the usefulness of aspect-orientation depends heavily upon its compatibility with object-orientation. Compatibility can be better understood in relation to notions of complementarity and symbiosis.

5.1.1 Aspect-orientation and object-orientation: compatibility in complementarity?

Complementarity implies wholeness and completeness. Two systems are complementary if they complete each other or, together, form a balanced whole.

Physics offers a model of complementarity in the understanding of the relationship between particle and wave in the understanding of light. This relationship is duality. The physical principle of complementarity suggests a blending of the wave and particle nature of matter. Scientists can learn or accomplish some things by considering light as particle and other things by considering it as wave. One approach, however, does not negate the other; both particle and wave are
always already existent. The following quote from *Encyclopedia Britannica Online* provides an insight into the principle of complementarity:

> Depending on the experimental arrangement, the behaviour of such phenomena as light and electrons is sometimes wavelike and sometimes particle-like; *i.e.*, such things have a wave-particle duality. It is impossible to observe both the wave and particle aspects simultaneously. Together, however, they present a fuller description than either of the two taken alone.

Perhaps this degree of complementarity can exist at the source code level, with woven statements not apparently attributable to either primary or crosscutting concerns. Nonetheless, the software developer does not seek a conceptual blurring in which a single whole is achieved. Instead, the developer seeks a blending in which aspect-orientation is incorporated smoothly into the work when appropriate and useful.

Object-orientation is largely considered complete or whole in and of itself. However, inherent in the definition of aspects (Kiczales et al, 1997) is the notion of crosscutting a primary concern—most likely an object. This primary-ness of an object gives an aspect the quality of the so-called other. As previously discussed in both the introduction of this dissertation and the introduction to this chapter, research in aspect-orientation has largely been done in relation to object-orientation.

Compatibility, rather than complementarity, defines the usefulness of aspect-orientation. The results of the AOPy project, therefore, are described in terms of compatibility.
5.1.2 Aspect-orientation and object-orientation: compatibility in symbiosis?

The notion of symbiosis is, ultimately, a better model for describing the compatibility relationship between aspect- and object-orientation.

Symbiosis can be defined as a relationship of mutual advantage or dependence. This dissertation accepts that the usefulness of aspect-orientation is dependent upon object-orientation and that both orientations benefit when compatibility occurs. The benefits of compatibility for object-orientation are in the traditional "-ilities"—maintainability, comprehensibility, trace-ability, evolvability, etc.—of software engineering. The benefit of compatibility for aspect-orientation is usefulness. The mutually beneficial relationship between aspect- and object-orientation provides a stronger overall solution, though the relationship, in current practice, remains one of dependence as well. The relationship between aspect- and object-orientation is examined in the experiment results in order to understand this compatibility and usefulness of aspect-orientation.

5.1.3 Aspect-oriented acts in response to an object-oriented worldview

The subjects for this experiment arrived with a background that was largely object-oriented in nature. The subject's backgrounds are summarized below and described fully in section 5.1.4. This existing background of the subjects gives the best reason for accepting symbiosis as a better model for compatibility. The
conceptual blurring that is implicit in complementarity would possibly require a rejection and certainly require a redefinition of object-orientation by the subjects. Compatibility offers a theory by which subjects might build upon or extend their expertise in object-orientation, rather than replacing it, by using aspect-orientation.

Nothing in the experiment's problem description mentions aspects or aspect-orientation. The terminology used, including *module, classes, concerns*, and so on, define the problem as object-oriented, with the possibility that aspect-oriented techniques might be incorporated.

The problems are written in an object-oriented manner using the class-based notation of Python. The problems are working object-oriented programs, and they are based on rather run-of-the-mill implementations of the Chain-of-Responsibility and Visitor design patterns (Gamma et al., 1995). A complete description of the problems and the experiment design is provided in Chapter 3.

Aspect-orientation is not written directly into the problems. Instead, its inclusion during the thinking-aloud sessions provides the experiment results.

The subjects have varying degrees of experience with object-oriented techniques. The table in section 5.1.4 summarizes their backgrounds. As a group, their professional software development experience ranged from none to four years of experience. All subjects were simultaneously enrolled in a graduate course on object-oriented programming. The subjects' more deeply ingrained, object-oriented, software worldview makes aspect-oriented techniques the less familiar and,
therefore, less likely choice during problem solving. In other words, the problems fit
the object-oriented training and mindset of the experiment subjects.

Aspect-oriented techniques were introduced to subjects as part of the
experiment workshop. After completing the workshop—consisting of 12 contact
hours—that discussed advanced separation of concerns and aspect-oriented
programming, the subjects participated in the experiment. Because the subjects’
backgrounds were in object-orientation and because the problems were written in an
object-oriented fashion, the subjects every recorded use of aspect-oriented ideas,
terms, or techniques was evidence of potential or actual compatibility.

This object-oriented mindset, when contrasted with their limited exposure to
aspect-oriented programming, indicates that each time a subject used a concept from
aspect-oriented programming the subject was actually participating in a transition.
This transition was a movement away from an object-oriented solution in their hands
and away from their education and experiences to a less familiar approach and
toward compatibility. The results presented in this chapter, then, are drawn from
selected, meaningful uses of and absences of aspect-oriented ideas, terms, or
techniques to determine compatibility.

5.1.4 Subject Backgrounds

Background surveys provide an understanding of the group of subjects and
provide a context in which the qualitative results might be considered. Surveys are
especially valuable in the earliest stages of qualitative research in order to establish an understanding of the demographic so that the researcher broadly understands the make-up of the group in relation to the setting and experiment problem. In the case of the AOPy project, the surveys validate the simulation of the professional development environment. However, the goal of qualitative research is particulars. In the AOPy project, therefore, the particulars are represented in the results of the problem-solving session.

As mentioned above, the subjects had a wide range of backgrounds. As can be seen in table 5.1, seven of the eight subjects were candidates for the master's degree, and one was a doctoral candidate.

<table>
<thead>
<tr>
<th>Subject #</th>
<th>Academic Experience</th>
<th>Industry Experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MS + 1</td>
<td>Yes (3 yrs.)</td>
</tr>
<tr>
<td>2</td>
<td>BS + 1</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>BS</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>BS</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>BS + 1</td>
<td>Yes (5 yrs.)</td>
</tr>
<tr>
<td>6</td>
<td>BS</td>
<td>Yes (4 yrs.)</td>
</tr>
<tr>
<td>7</td>
<td>BS + 1</td>
<td>Yes (1 yr.)</td>
</tr>
<tr>
<td>8</td>
<td>BS + 1</td>
<td>Yes (6 mo.)</td>
</tr>
</tbody>
</table>

*Table 5.1: Subject education and experience.*

Of these seven master's candidates, three were first-year graduate students, and four subjects had completed at least one year of graduate coursework. The one remaining subject—subject #1—had completed a BS, an MS, and one year of PhD work.
Five of the subject had between six months and five years of professional software development experience. Three of the subjects had no previous industry experience.

5.2 Framework for Experiment Results

Experiment results were determined by using categories, keywords, and mentions and mention sequences.

5.2.1 Categories

A category is an abstract description of an experimentally observed phenomena.

Categories are developed through a process known as coding (Charmaz, 1983). A code is a shorthand, a naming convention for denoting a specific category. Seidel and Kelle, in their paper “Different Functions of Coding in the Analysis of Textual Data” (1983), describe two types of codes and coding practices: 1) codes that denote a specific region of text, and 2) codes that denote specific facts.

The relationship of the codes and coding to the data generated by the experiment is established as follows (Seidel and Kelle, 1983):

[C]odes represent the decisive link between the original ‘raw data’, that is, the textual material such as interview transcripts or field notes on the one hand, and the researcher’s theoretical concepts on the other[.]
In a similar vein, Charmaz (1983) says the following:

In qualitative coding, researchers develop codes out of their field notes, interviews, case histories, or other collected materials (these could include diaries by participants, journals, interactional maps, historical documents, and so forth).

In this research project, coding represents the link between the data—contained on videotapes and in field notes from thinking-aloud experiments—and the concepts of compatibility and non-compatibility.

Ultimately, after an iterative process of working with and refining the codes and the concepts to which they refer, it is possible to generate categories. This process is outlined in the following quote from Charmaz (1983):

Codes may be treated as conceptual categories when they are developed analytically. This means the researcher defines them carefully, delineates their properties, explicates their causes, demonstrates the conditions under which they operate, and spells out their consequences.

A thorough overview of the application of this coding process to qualitative data analysis of the experimental data generated for this dissertation is presented in Chapter 6. That said, the result of this iterative process of coding led to the development of two conceptual categories.

Compatibility is one category. Compatibility, in terms of determining experiment results, is the natural-language utterance, during the videotaped thinking-aloud experiment, of an aspect-oriented idea, term, or technique. Compatibility, as defined in sections 5.1.1 and 5.1.2, is distinct from complementarity but akin to
symbiosis. Compatibility implies that similarities exist between aspect- and object-oriented or that differences are reconciled. Compatibility points out that, at the current time, things are working in ways conducive to aspect-orientation. Compatibility acknowledges proficiencies and usefulness in aspect-orientation.

Non-compatibility is the other category. Non-compatibility, in terms of determining experiment results, is the lack of using an aspect-oriented idea, term, or technique at a point when it could have or should have been incorporated. Non-compatibility is not designated merely as the absence of aspect-orientation in mention sequences but, rather, as thwarted compatibility.

This is distinct from incompatibility. Incompatibility implies that differences are irreconcilable. Non-compatibility points out that, at the current time, things are not working in ways conducive to incorporation of aspect-orientation. Nonetheless, by drawing attention to these areas, future research can improve upon the acknowledged deficiencies in compatibility.

5.2.2 Sources of Results

Though I initially considered that transcription might be representative of all data, I discovered even during the thinking-aloud sessions that the combination of the videotapes and my field notes could provide a more thorough and usable record of the data. Transcription includes meaningless details such as the articulation of uh and inserts punctuation, while at the same time losing the intonation of questions or
emphatic assertions as well as a sense of varying pauses. Transcriptions of specific results discussed and analyzed are included in this dissertation to offer the readers the available evidence.

In using the videotapes and field notes to determine relevant results, I noted relationships between articulations as well as noting unexpected articulations, all with a focus on issues of crosscutting concerns. Because thinking-aloud sessions were conversational, they were analyzed as such. For instance, because conversations consist largely of words, aspect-oriented terminology, particularly when used appropriately, marked a mention of possible interest. Likewise, because the sessions simulated the professional development environment, successful use of an aspect-oriented technique was of interest in determining results.

5.2.3 Keyword list

Keywords are derived from the terminology of the domains of object-oriented programming, advanced separation of concerns, and aspect-oriented programming. As is described in section 5.2.3, a mention—an experiment result—is denoted by a keyword.

The natural language domain that is available to the subjects is constrained by the areas of object-orientation, advanced separation of concerns, and aspect-orientation, all of which have been provided to the subjects in the form of their individual backgrounds, the graduate course in which they are enrolled, the
experiment workshop, and the problems themselves. The subjects' use of object-oriented terminology is influenced by the graduate-level course in object-oriented programming taught by Dr. Timothy Budd and by *An Introduction to Object-Oriented Programming* (Budd, 1997), the text for that course. The subject's terminology for aspect-orientation was introduced to them via the aspect-oriented workshop that preceded the experiment. The keywords are drawn from the shared experiences of the graduate course, the experiment workshop, and experiment problems.

While any descriptive terminology from the above fields could potentially be a keyword, the keyword list is not exhaustive. For example, some terms that are quite common to the domains—such as polymorphism and late binding—never appear in the experiment tapes. The list of keywords is limited to distinguish potential examples as compatible and non-compatible in relation to the incorporation of aspect-orientation.

Though any mention of aspect orientation, either in terms of notions such as crosscutting concerns or in terms specific to aspect orientation, indicate potential compatibility, the list of keywords was developed to help identify mention sequences in relation to compatibility and noncompatibility. The keywords are behavior, class(es), derives, encapsulating, events, function, method(s), message(s), object(s), overrides, concern(s), crosscutting, crosscutting concern(s), orthogonal, separate, separation, aspect(s), join, and join point(s).
5.2.4 Mentions and Mention Sequences

A mention is denoted by a keyword. Keywords are explained and listed in section 5.2.2.

If we allow that the length of a mention is one sentence—no matter how many keywords appear in that sentence—then there literally hundreds of possible events of interest that were recorded during the experiments. It would not make any sense analytically to take such an isolationist approach because it would be difficult to understand compatibility and the usefulness that it implies for aspect-orientation. It is necessary to develop examples as mention sequences in order to cut up the dialogue into meaningful pieces.

Mention sequences are sections during the thinking-aloud experiment that begin with a sentence that includes a keyword, end at an appropriate time, and define a coherent structural context.

The process of distinguishing mention sequences from the larger text is driven by several forces. Some individual mentions do actually live in isolation, contributing nothing meaningful to the contexts that came before or after. Some mentions begin a new context, i.e., a mention sequence. Some mentions come in the middle of an established context. This process of distinguishing mention sequences results in a coherent structural context.
The mention sequences are the primary means of event identification for the experimental results. Each mention sequence from a tape is identified by the group/tape number that the sequence comes from, the problem number, and the beginning time on the tape. For example, described below in section 5.3.1 is mention sequence 1-1-0:10:40. The mention sequence numbering scheme means that the mention sequence occurred during the problem sessions of group 1 (and is recorded on tape 1); the sequence occurred while the subjects were working on problem #1; and the mention sequence began at tape time 0:10:40 (hours: minutes: seconds).

5.3 Experiment Results

Results are presented to indicate categories. Along with objective measurement of mention sequences, examples are designated by category: compatibility or non-compatibility.

Each example of a mention sequence is described using the following experimental attributes: time on the videotape that the sequence started, subjects involved in the problem-solving session, keywords, summary of utterance and activity, ending phenomenon, category (either compatibility or non-compatibility), and relevant description of the sequence.

5.3.1 Mention sequence: 1-1-0:10:40

Time: 0:10:40 – 0:11:20
Subjects: 4 and 7

Keywords: concerns, crosscutting, classes, objects (as GUI things: Frames, Panels, Buttons)

Summary: The sequence centers on the identification of a crosscutting concern present in a set of classes.

Ended by: Subject 4 agrees with assessment of subject 7. Subject 7 then changes direction of conversation.

Category: Compatibility

Description: Subject 7 begins this mention sequence by indicating that the behavior encapsulated by the method `handleRequest()` should be considered a crosscutting concern. Subject 7 goes on to describe that the concern in question is present in the classes Frame, Panel, and Button. Subject 7 says, “in all of these things [Frame, Panel, and Button] you have to handle the request.” After considering such remarks, subject 4 replies, “Yes. Yes. Please go on.” This mention sequence terminates when subject 4 agrees with subject 7, after which subject 7 takes the conversation in a different direction.

5.3.2 Mention sequence: 2-1-0:48:20

Time: 0:48:20 – 0:50:11

Subjects: 1 and 8
Keywords: encapsulating, events, concern, join point, behavior, method(s), crosscutting concern(s), classes

Summary: Subject 8 describes the handling of events as a crosscutting concern. Subject 8 proposes to encapsulate the event handling behavior.

Ended by: Subject 1 articulates that subject 8’s treatment of the problem is much more general and substantively more correct than his own. Subject 1 then pauses.

Categories: Compatibility

Description: Subject 8 identifies event handling as a crosscutting behavior. Subject 1 focuses on the two lines of code that are present in the handleRequest() method. Subject 1 hypothesizes a situation in which the handleRequest() method contains 100 more lines of code beyond the two that are present in the problem implementation. Subject 1 suggests that these lines would not be a part of the same concern. Subject 8 extends this discussion by pointing out that whatever lines of code are present in the method is secondary and that the salient fact is that the concern is event handling. Subject 8 also correctly describes the event handling concern as cutting across the classes of the simulated GUI hierarchy. Subject 1 agrees that subject 8’s treatment is more correct than his own.

5.3.3 Mention sequence: 4-1-0:34:23

Time: 0:34:23 – 0:35:47
Subjects: 3 and 5
Keywords: concern, aspect, join points, method
Summary: Request/message handling concern is identified as an appropriate aspect by subject 3.

Ended by: The end of the statement of identification of the concern ends the mention sequence, which is followed by a lengthy pause.

Category: Compatibility
Description: Subject 3 begins the mention sequence by stating the need to create an aspect to "handle all requests." After encapsulation of this behavior in an aspect, subject 3 hints at reusability of the aspect by expanding the range of possible join points. Subject 5 restates that the concern is currently encapsulated in the method handleRequest() and that it will be necessary to specify a set of join points for the aspect.

5.3.4 Mention sequence: 1-1-0:13:00
Time: 0:13:00 – 0:13:20
Subjects: 4 and 7
Keywords: aspect, weave, classes, join [point]
Summary: The sequence centers on the modularization of a behavior in order to separate it out as an aspect.

Ended by: Subject 7 takes dialogue in a different design direction.
Categories: Compatibility.

Description: Subject 4 begins this mention sequence by suggesting a behavior that could be extracted as an aspect. Subject 4 suggests weaving the aspect back into three classes by means of a join point.

5.3.5 *Mention sequence: 2-1-0:14:32*

Time: 0:14:32-0:16:30

Subjects: 1 and 8

Keywords: crosscutting concern, orthogonal, aspect, functions, methods

Summary: Subjects 1 and 8 discuss repeated code in methods as indication of a scattered concern, specifically event handling.

Ended by: Subjects 1 and 8 turn to discussion of how to implement a method that is unimplemented in the problem system.

Category: Compatibility

Description: After briefly discussing the fact that SWT components handling their own events could be a crosscutting concern, subjects 1 and 8 discuss why they think that. Subject 1 states, "the best candidates for an aspect are the functions, the methods that are [...] repeated inside—." Subject 8 interrupts to affirm this assertion. Shortly thereafter, subject 8 states, "along these lines, there's this `add()` method that happens in both `Frame` and `Panel` because they're both container classes." The discussion, then, centers on the fact that there are methods that appear in each
component that implement the same behavior and that this behavior is not the primary responsibility of the SWT component. Nonetheless, although the add() method appears in both Frame and Panel, it was there only to suggest necessary behaviors for each class. As such, the method was unimplemented in each class. Subjects 1 and 8 went on to describe an implementation unrelated to separation of concerns.

5.3.6 Mention sequence: 3-1-0:42:30

Time: 0:42:30 – 0:47:55

Subjects: 2 and 6

Keywords: events (OO: message sends and AO: point in program execution), constructor, method, override

Summary: The sequence centers on event handling discussed in an anthropomorphic format and event handling as a concern.

Ended by: Conversation stops for an extended period.

Categories: Non-compatibility

Description: Subject 6 begins a long sequence that can primarily be characterized by subject 6’s use of an informal anthropomorphic design language. This begins with 6’s utterance of the phrase “Who gets the event after me?” This anthropomorphic statement about event handling is followed by a discussion about taking and gathering up all the event code that is present in each class. The gathered
event code would be decoupled, according to the statements made by subjects, from the classes for which it handles events. The gathered event code would only know when it needed to execute, according to the subjects.

5.3.7 Mention sequence: 1-1-0:13:23

Time: 0:13:23 – 0:14:35

Keywords: aspect, join (as a verb), message, method

Summary: Subject 7 describes a series of design decisions that would lead to a more obvious, more intuitive solution.

Ended by: Subject 7 attempts to elicit a solution from subject 4. There is not response.

Categories: Non-compatibility.

Description: Subject 7 begins this mention sequence by posing two questions concerning possible join points: 1) “What would you join to?” and 2) “How would you specify it?” Subject 7 then proceeds to outline a possible design. The proposed design includes creating a smarter handleRequest() and placing it in class Handler. This smarter version solution, according to the subjects, would be based upon collecting the variety of behaviors that handleRequest() implements in other classes—notably for the analysis, classes Panel and Button—and then moving this collection of behaviors into Handler. Subject 7 then points out that
while this collection of behaviors is a crosscutting concern, an aspect would not be
the intuitive solution.

5.3.8 Mention sequence: 4-2-1:03:43

Time: 1:03:43 – 1:05:10

Subjects: 3 and 5

Keywords: concern, method, classes, separate

Summary: Subject 5 identifies an appropriate concern, but the subjects fail to
implement an aspect-oriented solution.

Ended by: Subject 3 begins to sketch out pseudocode, after which dialogue
diminishes significantly.

Category: Non-compatibility

Description: Subject 5 begins this mention sequence by identifying code that
facilitates the use of double-dispatching (Budd, 1997) and the visitor design pattern.
Subject 3 asks what the concern is. Subject 5 poses this code as a concern. This
concern is identified as being encapsulated in the accept() method. Separating the
concern from the Node class is proposed by Subject 5. The feasibility of placing the
concern into an existing solution class is described. Subject 3 suggests making it part
of NodeTraveller.
5.4 Summary of Experiment Results

Results from eight mention sequences have been presented. Five of the mention sequences—1-1-0:10:40, 2-1-0:48:20, 4-1-0:34:23, 1-1-0:13:00, and 2-1-0:14:32—have been determined to indicate compatibility between aspect- and object-orientation. Three of the mention sequences—3-1-0:42:30, 1-1-0:13:23 and 4-2-1:03:43—demonstrate non-compatibility between aspect- and object-orientation.
CHAPTER 6 CASE STUDY: ANALYSIS OF EXPERIMENT RESULTS

6.1 Analytical Framework for Experiment Results

As asserted in Chapter 3 of this dissertation, software development is a social phenomenon. In his chapter "Managing People" in Software Engineering, Ian Sommerville (2001) proffers the following about the importance in considering social factors in software development:

The people working in a software organization are its greatest assets. They represent intellectual capital [... .M]y discussion of management [is based] on cognitive and social factors rather than any, currently fashionable, management theory. Software engineering is a cognitive and social activity, so these factors are important in developing an understanding of how people write software.

This important social component in software development makes analysis using grounded theory techniques particularly useful in understanding specific relationships between object-oriented programming and aspect-oriented programming. To understand the results of the AOPy project presented in Chapter 5, grounded theory can be used to explore the assumptions of compatibility and non-compatibility that developers make between aspect- and object-orientation.

Grounded theory—though not previously used specifically for evaluating software development techniques—has been used in a similar manner for evaluating design tools and methodologies (Calloway and Ariav, 1991). Additionally, grounded
theory has a history of application in information science (Toraskar, 1991) and empirical software engineering research (Orlikowski, 1993).

6.1.1 Grounded Theory Overview

Grounded theory is the qualitative data analysis approach used for the AOPy research project. The essence of grounded theory is the inductive, evolutionary development of a theory that describes a particular phenomenon. It was not used as part of the experimental framework to plan the preliminary experiment. Instead, it has been used since the preliminary experiment to generate a theoretical model that has explanatory powers for describing the aspect-oriented phenomena that were observed and recorded during both sets of experiments.

In “Grounded Theory as Scientific Method,” Brian Haig (1995) gives the following summary of the grounded theory methodology:

Grounded theory is typically presented as an approach to doing qualitative research, in that its procedures are neither statistical, nor quantitative in some other way. Grounded theory research begins by focusing on an area of study and gathers data from a variety of sources, including interviews and field observations. Once gathered, the data are analyzed using coding and theoretical sampling procedures. When this is done, theories are generated, with the help of interpretive procedures, before being finally written up and presented.

More information about grounded theory can be found in Charmaz (1983). Charmaz has the following to say:

The grounded theory method stresses discovery and theory development rather than logical deductive reasoning which relies on
prior theoretical frameworks. These two aspects of the method lead the grounded theorist to certain distinctive strategies. First, data collection and analysis proceed simultaneously[...]. Since grounded theorists intend to construct theory from the data itself, the need to work with solid, rich data that can be used to elicit thorough development of analytic issues [...]. Grounded theorists shape their data collection from their analytic interpretations and discoveries, and therefore sharpen their observations. [...] These strategies serve to strengthen both the quality of the data and the ideas developed from it.

As is described above, the ultimate goal of a grounded theory inquiry is the generation of a theory (or theories). The theory is the result of an iterative process that seeks to generate categories of phenomenon description and the patterns of relationships that link and, in some sense, help to define the categories (i.e., the observed phenomena).

It is important to keep in mind the significance of the difference between experimental phenomena and experimental data. Data only exist as a tangible indication that a phenomenon has been observed (Bogen and Woodward, 1988).

As Herbert Simon said in “Creativity in the Arts and in the Sciences” (2001):

Phenomena must be observed, recorded, and analyzed objectively; theories describing and explaining the phenomena must be constructed; the theories must be matched with the phenomena at some level of detail and to some degree of accuracy.

This approach is the intended use of the grounded theory technique in this dissertation.
6.1.2 Application of Grounded Theory to Experiment Results

This investigation did not start out as an explicit evaluation of the compatibility between aspect-orientation and object-orientation. Instead, it began as a loosely defined comparison. Only after the initial discovery of and the ensuing iterative application of grounded theory techniques was the notion of evaluating the compatibility of aspect-orientation with object-orientation articulated.

The original direction for this research was provided as a comment by Alan Borning during a presentation (Dechow, 2002) made by the author. Dr. Borning suggested adopting the simplified thinking-aloud technique from discount usability engineering (Nielsen, 1989). The thinking-aloud technique is particularly good technique for data generation.

A later meeting with Curt Cook also suggested that a thinking-aloud study would be useful technique for the type of evaluation used in the AOPy project. As was previously mentioned in Chapter 3, Dr. Cook suggested that the first experiment be designed in such a way that it addressed the first principles that are claimed as aspect-orientation. This experimental approach for data generation and exploration of a research area resulted in a preliminary experiment—CS180—that sought to evaluate aspect-oriented language constructs (aspects, join points, and advice) and the modularization of crosscutting concerns.

Shortly after the preliminary experiment was completed, grounded theory provided the underpinnings for the research process that ultimately is reflected in the
analysis of results. Field notes and videotapes from the thinking-aloud experiment of CS180 were analyzed. These preliminary experiment results were presented elsewhere (Dechow, 2004) and then led to a working hypothesis or theory that could be tested in the main experiment, CS582. These preliminary results indicated that mastering the syntax and semantics of an aspect-oriented language—Pythius (2003) in this case—was relatively easy for software developers. However, identifying and modularizing crosscutting concerns was difficult.

The preliminary experiment was not designed to focus on the relative ease and difficulty subjects had in mastering ideas, terminology, and techniques of aspect-oriented programming. Instead, CS180 was designed to generate data from which a theory could—and did—emerge. Grounded theory fosters this coordinated back-and-forth between data collection and analysis. Therefore, once a meaningful theory emerged from observation of the data on the videotapes and in field notes, that hypothesis needed to be tested in the main experiment with more focus in order to generate additional data, useful results, and more meaningful, detailed analysis.

The main experiment, including its workshop, was focused on the consequences of the second part of the working hypothesis: the subjects of the CS180 experiment had great difficulty in identifying and separating crosscutting concerns. With a minimalist approach to the implementation of aspect-oriented support in Python, the focus of the main experiment therefore, is most clearly on developers’ ability to identify and separate concerns.
Through developing the software, implementing the workshop, and conducting the experiment, a further refinement of the theory emerged. This hypothesis can be expressed as follows: Separation and identification of all concerns, not just crosscutting concerns, is not merely a programming language activity. It also takes place cognitively and linguistically in the activities—in the articulation of ideas, the use of problem-specific terminology, and the employment of techniques—of developers.

The main experiment was not designed to elicit aspect-oriented ideas, terminology, and techniques per se that could be used as results, though it is logical that this sort of evidence would be exist in the thinking-aloud experiment. The videotapes of the experiment sessions were coded, sorted, and analyzed for sequences of interest regarding the working experiment question and theory. The experiment hypothesis or theory evolves further through this initial observation of the videotapes.

Questions that emerged included the following: What about the language that developers use as they transition from object-orientation to aspect-orientation, and vice versa? Are there parallel structures, or repeated structures? Previously designated sequences of interest were further coded and reviewed with these new theories in mind.

 Eventually, through this back-and-forth analytical process of observing the data to generate theories and using the theories to further observe the data and so on,
the theory of compatibility as outlined in sections 5.2 through 5.4 emerged. The process made it clear that usefulness of aspect-oriented programming could be most clearly established by categorizing results as compatible or non-compatible.

Once categories were established, mention sequences were evaluated for aspect-oriented ideas, terminology, and techniques, because when evidence of any of these three elements existed, developers were engaging in actual or potential compatibility of aspect-orientation with the object-orientation already present in the experiment problem itself. By analyzing the ideas, terminology, and techniques of particular mention sequences, experiment results were addressed. Research questions related to usefulness of aspect-oriented programming were developed. The questions addressed through analysis of results include: Are object-oriented programming and aspect-oriented programming a good fit for each other? If so, what indicates this good fit, or compatibility, in practice? Are there disconnects and discontinuities that we can foresee? If so, what indicates this inhibition of compatibility in practice?

The questions relate to the emergence of object-oriented programming as the currently dominant industrial paradigm and to the position of aspect-oriented programming in relation to this dominant paradigm. When object-orientation first emerged from labs into the mainstream, it was touted as a breakthrough in software reuse. People like Brad Cox (1986) started to talk about software ICs, or integrated circuits. After more experience with object-orientation was gained, ensuing studies (Kamath, Smilan, Smith, 1993) showed that object-oriented reuse is challenging.
This work went on to show that even competent and well-trained developers can take years to reach rewarding levels of object-oriented reuse.

In relation to the dominant object-oriented paradigm, aspect-oriented programming is a relative newcomer to software development. Aspect-oriented programming is being delivered by tool-builders or researchers as a tool (a paradigm) that is implicitly compatible with object-orientation (Kiczales et al, 1997). For this assumption to be true in the practice of software development, aspect-orientation must also be compatible in the minds and practices of the tool-users or developers. In other words, aspect-orientation is only effectively compatible with object-orientation in as much as it is compatible in its actual use of ideas, terminology, and techniques by software developers. Usefulness of aspect-oriented programming, then, can be defined, at least in part, upon its compatibility with object-oriented programming in practice.

Once grounded theory analysis is applied as this section describes, a categorical framework for evaluating the traits associated with that compatibility, which establishes usefulness of aspect-oriented programming, is necessary for analysis of the AOPy experiment results.

6.2 Categorical Framework for Analysis of Experiment Results

As was described in section 5.2.1, a category is a descriptive name that suggests something about the underlying phenomena that it denotes. For instance, in
the case of aspect-orientation compatibility, the category name is meant to designate sequences of natural language that the subjects used while engaging in a narrative description that exemplifies ideas, terms, or techniques of either object-orientation or aspect-orientation.

6.2.1 Definition of Compatibility Category

Compatibility occurs when aspect-oriented ideas, terms, or techniques are put into practice by subjects during problem solving. The experiment results indicate that compatibility has the following traits:

1. An aspect-oriented idea, such as the identification of a concern that cuts across classes, is employed during problem solving.

2. Aspect-oriented terminology, such as crosscutting or join point, is used to describe part of the problem or process of solving the problem.

3. An aspect-oriented technique, such as the creation of an aspect to address a crosscutting concern, is proposed in problem solving.

While this list of traits is not exhaustive, it is consistent and also comprehensive enough to provide indicators during the experiment that establish compatibility. This list denotes characteristics of the results in the compatibility category. These characteristics, in turn, indicate usefulness of aspect-orientation.
Mention sequences that have all of these traits are deemed to be compatible. Mention sequences that do not have all of these characteristics or have characteristics that thwart these traits are deemed to be non-compatible.

6.2.2 Aspect-Oriented Ideas

This trait—employment of aspect-oriented ideas—is primarily associated with important aspect-oriented first principles such as identification of crosscutting concerns and join points. The natural language of mention sequences is primarily conceptual. When sequences show evidence of this trait, the subjects demonstrate aspect-oriented concepts, such as identification of concerns that crosscut the system.

6.2.3 Aspect-Oriented Terminology

Though most superficial of the categories, when appropriate aspect-oriented terminology is paired with the appropriate aspect-oriented idea, understanding of aspect-orientation is demonstrated. In addition, demonstrating this understanding of aspect-orientation through appropriate terminology allows the developer to more clearly communicate about the aspect-oriented ideas with another developer.

6.2.4 Aspect-Oriented Techniques

As opposed to the trait of aspect-oriented ideas defined in section 6.2.2, the aspect-oriented techniques trait is concerned with application of the aspect-oriented
concepts. Identification of a crosscutting concern shows that the subject understands the concept. Use of appropriate aspect-oriented terminology to talk about the concept indicates increasing understanding of aspect-orientation. When they employ an aspect-oriented technique, subjects use that understanding in practice. For example, a discussion of weaving the aspect into the existing objects demonstrates the ability to apply aspect-orientation to the problem.

6.3 Application of Compatibility Framework to Mention Sequences

The following is an analysis of eight mention sequences. Each sequence is described in the subsections of section 5.3 and characterized here by its adherence to the compatibility evaluation framework.

6.3.1 Mention Sequence 1-1-0:10:40

This mention sequence—1-1-0:10:40—is described in section 5.3.1. This mention sequence is an example of compatibility. The phenomena exhibited in this example generate each of the compatibility criteria described in the evaluation framework.

1. Aspect-oriented ideas were used: subjects identified and separated the crosscutting concern. Subjects get to heart of modularization of a crosscutting concern. Subject 7 was able to identify the same behavior present in three classes: Frame, Panel, and Button.
2. The correct terminology was used: subject 7 described the behavior as a crosscutting concern.

3. The appropriate aspect-oriented technique was employed by the subjects: aspect-orientation is posited as a viable solution. In identifying behavior in three classes, subject 7 was able to demonstrate a principle of aspect-oriented programming: modularization of a crosscutting concern. In this case, the concern was associated with request handling in a simulated windowing toolkit.

6.3.2 Mention Sequence 2-1-0:48:20

This mention sequence—2-1-0:48:20—is described in section 5.3.2. This mention sequence is an example of compatibility. The phenomena displayed by this example of compatibility exhibit all three of the traits in the compatibility category of the analysis framework.

1. Aspect-oriented ideas were used: the encapsulation of crosscutting concerns is described in a way that is consistent with aspect-orientation.

2. The appropriate aspect-oriented terminology was used. Event handling is described as a concern that cuts across several classes. The term join points is used to describe a specific location in the control flow of a program.
3. Aspect-oriented techniques were employed. Subjects 1 and 8 both articulated their understanding of how to use join points to identify particular points of program execution as a means to manipulate program behavior. Further, both subjects articulated that it is possible to apply multiple behaviors to a single join point.

6.3.3 Mention Sequence 4-1-0:34:23

This mention sequence—4-1-0:34:23—is described in section 5.3.3. This mention sequence is an example of compatibility. The phenomena exhibited in this example generate each of the compatibility criteria described in the evaluation framework.

1. Aspect-oriented ideas were used in the identification of the message handling concern.

2. The correct terminology was used. Aspects are described as the appropriate concern handling technique. Join points are used to describe the locality of event occurrences in classes.

3. The appropriate aspect-oriented technique was employed by the subjects: the subjects plan a viable aspect-oriented solution. The tangled concern is removed. An aspect is used to encapsulate the concern. Join points are specified to locate the behavior.
Subject 3’s description of aspect reusability by extension of the set of possible join points is particularly interesting in this example. Because reusability was not discussed in the workshop to prepare the students for the experiment problems, this articulation by subject 3 is evidence that the subjects can use the aspect-oriented ideas, terminology, and techniques to move beyond basic concepts.

6.3.4 Mention Sequence 1-1-0:13:00

This mention sequence—1-1-0:13:00—is described in section 5.3.4. This mention sequence is an example of compatibility. The phenomena displayed by this example of compatibility produce all three of the traits in the compatibility category of the analysis framework.

1. Subject 4’s description of a join point reveals the trait of aspect-oriented ideas.

2. In describing the join point, subject 4 identifies a method, handleRequest(), as encapsulating a crosscutting concern. The aspect-oriented terminology of aspect and join point are both appropriately used.

3. The behavior encapsulated in the method handleRequest() is the correct crosscutting behavior to identify as an aspect. The process of using a weaver to weave the aspect into the three other classes is an employment of aspect-oriented technique.
6.3.5 Mention Sequence 2-1-0:14:32

This mention sequence—2-1-0:14:32—is described in section 5.3.5. This mention sequence is an example of compatibility. The phenomena displayed by this example of compatibility exhibit all three of the traits in the compatibility category of the analysis framework.

1. Several aspect-oriented ideas were used. Subject 1’s description of repeated code statements—encapsulated in functions or methods—as being orthogonal to a component’s main concern is an acknowledgment that the code segments are a tangled concern, though the subject did not use the term tangled concern. Additionally, the subjects recognized that the appearance of the same repeated code in a variety of components indicated that the code was, in the subject’s word, scattered.

2. The subjects used aspect-oriented terminology appropriately. Crosscutting concerns are named as such. Crosscutting concerns are denoted by the term orthogonal in defining their relationship to a component’s primary—or as subject 1 said, main—concern.

3. The appropriate aspect-oriented technique was employed. The subjects describe the need to place the crosscutting concerns in an appropriate concern handling technique: an aspect.
6.3.6 Mention Sequence 3-1-0:42:30

This mention sequence—3-1-0:42:30—is described in section 5.3.6. This mention sequence is an example of non-compatibility.

Subject 6 articulates a long description of the problem in an almost completely anthropomorphic manner. In some object-oriented design descriptions, anthropomorphism is a hallmark characteristic (Wirfs-Brock, Wilkerson, and Wiener, 1990).

Anthropomorphism—not merely the treating of something as if it were human, but the discussing of oneself as an object as if one is part of the software—creates a situation in which the experiment subjects remain in object-oriented terminology. When subjects employ anthropomorphism, they do not employ aspect-orientation. Anthropomorphism, in this mention sequence, prevents subjects from transitioning into aspect-oriented ideas, terms, and techniques even when the transition may be appropriate and is potentially useful. The anthropomorphism encouraged in object-orientation reduces, or even eliminates, the compatibility of aspect-orientation in this case.

Subjects 2 and 6 identify the essential nature of crosscutting without acknowledging it. Here exists a situation in which a set of behaviors—the event code—is in each of the “thing” classes (the UI widgets). The subjects do not, though, in any indication through natural language acknowledge the potential usefulness of
aspect-orientation nor incorporate aspect-oriented terms or techniques. Time elapses to 0:44:22.

At 0:45:44, subject 6 says of gathering together all the event code, “Really nice goal: all it needs to know is WHEN.” This utterance gets to the heart of the definition of a join point: “A join point is a well-defined point in the execution of a program (Gradecki, 2003).” Again, while the researcher acknowledges this point as one of potential compatibility, the subjects themselves make no articulation that they are proffering aspect-oriented ideas.

6.3.7 Mention Sequence 1-1-0:13:23

This mention sequence—1-1-0:13:23—is described in section 5.3.7. This mention sequence is an example of non-compatibility.

In this example of non-compatibility, the intuitiveness of object-orientation—its seemingly natural truthfulness or appropriateness without much reasoning out or analyzing—inhibits the incorporation of aspect-orientation. Subjects’ verbalizations indicate object-orientation as the dominant and intuitive paradigm to the extent that aspect-orientation seems burdensome in comparison. In this example, aspect-oriented ideas, terms, and techniques are described as unintuitive and dismissed, even though the subjects convey an understanding of those ideas, terms, and techniques and discuss aspect-orientation as a plausible option.
Potential compatibility is verbalized by the subjects, but non-compatibility defines this example. When object-orientation is viewed as intuitive by the subjects, compatibility of aspect-orientation is thwarted.

6.3.8 Mention Sequence 4-2-1:03:43

This mention sequence—4-2-1:03:43—is described in section 5.3.8. This mention sequence is an example of non-compatibility.

Subjects appropriately use aspect-oriented ideas when they propose and discuss the separation of an entangled concern. Subjects also appropriately use aspect-oriented terminology such as concern, method, class, and separation, but subjects did not use terms in consistently aspect-oriented ways. In fact, subject 5 may be using concern and aspect interchangeably. Moreover, no aspect is proposed, no join points are discussed, and no advice method is employed.

Potential compatibility is displayed in the articulations of the subjects, but non-compatibility defines this example. Subjects verbalize only abstraction and modularity via encapsulation of a separated concern, but they do not employ an aspect, do not identify join points, and do not suggest the use of the advice method. The subjects propose removing a concern from one object and suggest putting it in another existing object, rather than creating an aspect. Even though advanced separation of concerns and aspect-oriented programming are discussed by the
subjects, the example represents non-compatibility because the final proposed solution relies only on object-oriented techniques.

6.4 Summary of Analysis

6.4.1 Overview

Analysis of the experiment results establishes usefulness by defining compatibility and non-compatibility of aspect-orientation with object-orientation. Compatibility and non-compatibility are terms to describe the relative usefulness of two-dimensional, aspect-orientation in specific mention sequences during problem-solving sessions involving subject pairs. While feasibility of aspect-orientation in this problem is established in Chapter 4, the mention sequences, which were determined by their potential for aspect-orientation, establish its relative usefulness as exhibited by the subjects’ articulations during thinking-aloud sessions involving object-oriented problems that were conducive to aspect-orientation.

6.4.2 Analysis of Compatibility

The following mention sequences are examples of compatibility: 1-1-0:10:40; 2-1-0:48:20; 4-1-0:34:23, 1-1-0:13:00; 2-1-0:14:32. Compatibility is established in the AOPy project by the subjects’ use of aspect-oriented ideas, terms, and techniques during thinking-aloud problem-solving sessions. Just as Kiczales et
al. (2002) suggest that aspect-orientation requires a way of thinking about it, talking about it, and doing something about it, the criteria developed to analyze the example mention sequences in this experiment investigate the aspect-oriented ideas, terminology, and techniques, particularly as they are related to crosscutting concerns, that the subjects articulated during the thinking-aloud, problem-solving sessions.

The most common, appropriate use of aspect-oriented ideas in this sampling of mention sequences was the identification of crosscutting concerns or a join point. Sometimes things like join points are described in the absence of a larger discussion of crosscutting concerns. The identification of a crosscutting concern or join point—even if the appropriate terms such as crosscutting, orthogonal, and join point were not initially used in the descriptions—is an essential element of compatibility. Logically, unless subjects are able to articulate aspect-oriented ideas in the recognition of such things as concerns that cut across classes, they cannot use aspect-orientation to solve problems or consider two-dimensionality. Therefore, articulation of aspect-oriented ideas indicates compatibility and the potential for using aspect-oriented terminology and techniques.

While the extent of the use of appropriate aspect-oriented terminology varied, compatibility was not confirmed unless aspect-oriented terminology was used. Aspect-oriented keywords present in more than one example include the following: concern(s), crosscutting, crosscutting concern(s), join point(s), aspect(s), method(s), and class(es). Logically, unless subjects are able to use terminology consistently and
in ways consistent with aspect-orientation, compatibility cannot be firmly documented. Appropriate use of aspect-oriented terms indicates compatibility and the ability to communicate with others about aspect-orientation.

Because the use of aspect-oriented techniques, particularly coupled with the appropriate terminology, demands the application of aspect-oriented ideas, the employment of aspect-oriented techniques is the cornerstone of the compatibility category. All compatibility examples show developers using aspect-orientation as a solution to address what they identify as crosscutting concerns. Therefore, articulation of aspect-oriented techniques indicates compatibility and establishes usefulness because it demonstrates that subjects can do something about those crosscutting concerns that they identify and name.

Based on analysis of the experiment results, aspect-orientation is most useful when it is compatible with object-orientation in the implementation of ideas, terminology, and techniques. Only then can aspect-orientation simplify tangled code, thereby making it more comprehensible, more easily maintained, and more conducive to evolution. In other words, the usefulness of aspect-orientation depends upon the extent to which aspect-orientation fosters the developers' ability to reconfigure, to some extent, the problem as a two-dimensional separation of concerns with objects + aspects.

Often, when a developer pair identified a crosscutting concern appropriately, they discussed it in a reinforcing manner and built each other's certainty that it was a
crosscutting concern. The subjects articulated a way of thinking about it, talking about it, and doing something about it. They also reinforced each other in discussing two-dimensional separation of concerns. This manner of discussing an appropriate crosscutting concern and what to do about it indicates that the pair was in agreement and that they found aspect-orientation a good fit or that, in the language of this research project, they found it compatible.

6.4.3 Analysis of Non-Compatibility

Based on analysis of the experiment results, aspect-orientation is least useful when characteristics fostered by object-orientation and one-dimensional separation of concerns inhibit the incorporation of aspects. Characteristics that are fostered by object-orientation but not by aspect-orientation, in its current state, include anthropomorphism, intuitiveness, and an object-centric approach based on an existing worldview.

Non-compatibility, then, may occur for a variety of reasons. However, experiment results for non-compatibility identify and define a few particular phenomena that inhibit the incorporation of aspect-oriented ideas, terms, and techniques when aspect-orientation is a logical, useful option during problem solving. The meaningful characteristics of non-compatible mention sequences are as follows:
1. Anthropomorphism—or the discussing of an object as if it were oneself—occurs in mention sequence 3-1-0:42:30. This anthropomorphism prevents the incorporation of aspect-orientation when the potential for its use exists. Anthropomorphism is a characteristic that object-orientation fosters, but that aspect-orientation, in its present configuration, does not encourage. Therefore, anthropomorphism inhibits compatibility of aspect-orientation; aspect-orientation is less useful when anthropomorphism, encouraged by object-orientation, exists in problem solving.

2. Aspect-orientation is articulated as unintuitive in an implied or direct comparison with object-orientation, which is taken as intuitive by the subject(s). Intuitive ideas, terms, and techniques are those that, in the subjects' conversation, seem true or useful without much reasoning or analysis. In mention sequence 1-1-0:13:23, the dominance of the object-oriented paradigm and its one-dimensional separation of concerns is indicated by subjects' designation of object-orientation as intuitive. In relation to the dominant paradigm, aspect-orientation seems less intuitive or even unintuitive. Subjects tend to proceed with the more intuitive—i.e., object-oriented—step in problem solving. Therefore, though aspect-orientation may be potentially useful, the relative lack of intuitiveness that aspect-orientation currently encourages in practice inhibits its compatibility with object-orientation.
3. Aspect-orientation is excluded as a solution when an object-centric approach—viewing the world as objects—dominates development. This object-centric approach includes the bias of the subjects toward object-oriented ideas, terms, and techniques and toward one-dimensional separation of concerns based on such factors as their backgrounds and the problem description. When developers define everything in relation to object-orientation and objects, they leave no room for aspect-oriented ideas or techniques. In fact, as in mention sequence 4-2-1:03:34, two-dimensional separation of concerns terms are employed when the ideas and techniques articulated remain one-dimensional. In an object-centric approach, the ideas, terms, and techniques of aspect-orientation are either excluded, used inappropriately, or reconfigured in favor of object-oriented ideas and techniques. This object-centric approach thwarted compatibility because subjects became stuck or centered on object-orientation instead of reconfiguring the separation of concerns in aspect-orientation.

These characteristics of non-compatibility indicate that object-orientation is the powerful default for ideas, terms, and techniques during certain problem solving sessions. While aspect-orientation offers viable, useful solutions for crosscutting concerns, certain characteristics such as anthropomorphism, intuitiveness, and an
objectcentric approach that are otherwise fruitful characteristics can thwart developers’ full use of aspect-oriented programming.

Where potential usefulness exists, actual usefulness of aspect-orientation is inhibited by its current inability to encourage or incorporate anthropomorphism; by its current demands on subjects for the conscious reasoning out or analysis of aspect-oriented ideas, terms, and techniques; and by its current subordinate position to objectcentric approaches in talking about problems and implementing solutions.

Logically, increased relative usefulness of aspect-orientation depends upon increasing compatibility and decreasing non-compatibility. Characteristics of non-compatibility, such as anthropomorphism, intuitiveness, and objectcentric approaches, thwart potential compatibility.
CHAPTER 7 CONCLUSION

7.1 Overview

The AOPy project is, in the end, a valid and valuable study because

1. it is consistently qualitative in intent, design, and analysis;

2. it uses the appropriately qualitative subject-observer method of data collection in thinking aloud;

3. it approximates the professional software development environment and dynamics of development teams;

4. it uses a preliminary experiment to focus on an area of interest and a main experiment to generate results within that area for analysis;

5. it analyzes by consistently using categories of compatible and non-compatible to interpret results of interest;

6. it makes well-supported conclusions that can be used by software project managers and administrators to improve the usefulness of aspect-orientation.

In the AOPy project, the purpose of researching aspect-orientation and the development of aspect-oriented systems is communication with practitioners with the goal of understanding and improving current aspect- and object-oriented practices. In
order to pursue this line of inquiry in the AOPy research project, the following tasks have been accomplished and documented in the dissertation.

The context of software complexity and the research goals of the AOPy project were stated and outlined in Chapter 1. The motivating circumstances and rationale for this research work were provided in that chapter.

An extensive survey of the relevant advanced separation of concerns literature led to a focus on the practice of aspect-oriented programming. The literature survey is presented in Chapter 2. This survey establishes the larger scientific context for the AOPy research project.

The AOPy framework was built to support Kiczales’ three-point definition of an aspect-oriented system. The AOPy aspect-oriented framework is described in Chapter 3. The AOPy framework was demonstrated in the case study presented in Chapter 4. This case study evaluates the use of aspect-oriented programming techniques in the implementation of the observer design pattern. This case study demonstration of the AOPy framework establishes the validity of aspect-orientation in the context of this research project.

The experiment methodology used to explore the usefulness of aspect-orientation in practice is presented in the second half of Chapter 3. The preliminary and main experiments—CS180 and CS582—used thinking-aloud techniques to generate qualitative data. An analysis of the results of the preliminary experiment—CS180—was communicated in Dechow (2004). This work was used in conjunction
with Grounded Theory (Glaser and Strauss, 1967), a qualitative research methodology, to plan the main experiment, CS 582. The selected results of the main experiment—CS 582—are described in Chapter 5.

The results of the CS582 experiment are analyzed in Chapter 6. The iterative application of grounded theory techniques gave rise to the development of the aspect-orientation compatibility framework. The compatibility framework serves as the basis for the analysis presented in Chapter 6.

In addition to the preceding summary of the dissertation's accomplishments, Chapter 7 presents, in the following two sections, conclusions about the usefulness of aspect-oriented programming and applies the research documented in the dissertation to the future of aspect-oriented programming.

7.2 The Usefulness of Aspect-Oriented Programming

For practical purposes, aspect-orientation can be viewed as an extension of and enhancement of object-orientation with advanced separation of concerns techniques. The current practice of aspect-oriented programming reflects this view in the unstated assumption that aspect-orientation will be paired with object-oriented programming. The compatibility framework that was developed for the AOPy research project and that is described in this dissertation can be used by others to evaluate how well aspect-orientation fits the field of software development currently dominated by object-orientation. In other words, project managers can use the
compatibility framework to assess the usefulness of their teams’ aspect-oriented practices and discover where potential compatibility is being used successfully and where it is being thwarted. Thus, the compatibility framework is a significant contribution to the field as it is practiced.

This dissertation establishes that, for greatest usefulness of aspect-oriented programming, developers must find the two-dimensionality of aspect-orientation to be a good fit with the one-dimensionality of object-orientation and move with relative ease between these as is mutually beneficial to each approach. As such, the future success of aspect-oriented programming depends upon its compatibility with object-oriented programming. This assertion, supported by the analysis of experiment results using the compatibility framework, is the primary contribution of the AOPy research project. In other words, the primary contribution of this dissertation is a qualitative analysis and evaluation of the ways in which aspect-oriented programming currently works with object-oriented programming.

Perhaps an equally important contribution to the field is the development and use of qualitative techniques to generate, document, and analyze data. The use of thinking-aloud sessions was important to the generation of qualitative data. In particular, too, the use of grounded theory techniques in the evaluation of a programming methodology is new to the field. Whereas other approaches in the field have been valuable in viewing software development in relation to particular scientific methods, grounded theory techniques provide an especially good
theoretical process for taking advantage of software development as, in part, a social
task and a task in flux and progressive.

In addition, as Herbert Simon said in his essay “Creativity in the Arts and in
the Sciences” (2001):

Phenomena must be observed, recorded, and analyzed objectively;
theories describing and explaining the phenomena must be
constructed; the theories must be matched with the phenomena at
some level of detail and to some degree of accuracy.

By using the grounded theory technique in computer science, the AOPy research
project reflects Simon’s approach to understanding creative tasks, regardless of the
field. Grounded theory could be used more widely in future computer science
research to understand phenomena in new ways.

Overall, the contributions that the AOPy research project makes include the
development of a compatibility framework that can be used to judge usefulness of
aspect-orientation in practice, the assertion that usefulness is greatest when particular
traits of compatibility exist, and the development of qualitative analysis using
grounded theory.

7.3 The Future of Aspect-Orientation

In common practice, grounded theory techniques are used to generate
categories of phenomenon description and patterns of relationships between
categories. In this AOPy project, the techniques were used to develop a theory
describing the specific relationships between object-oriented programming and aspect-oriented programming. The resulting theory is embodied in the compatibility framework. The compatibility framework suggests a variety of areas in which useful information has been gleaned and can be applied in practical ways: aspect-oriented analysis and design, approaches to pedagogy, and guidelines for further tool developments. In other words, this dissertation's assertion of a compatibility framework provides a guide for 1) future research and analysis of the usefulness aspect-orientation in a variety of problem-solving situations, 2) for software development teams incorporating aspect-orientation into their practices, 3) for classroom education and on-the-job training in aspect-orientation, and 4) for areas in which the field might pursue new tool developments.

Winograd and Flores (1986), in *Understanding Computers and Cognition*, had this to say about the nature of tool usage: “This transparency of interaction […] is not achieved by having a car communicate like a person, but by providing the right coupling between the driver and action in the relevant domain[.]” Compatibility, like transparency of interaction, offers the seamless use of aspect-orientation. When aspect-oriented ideas, terminology, and techniques are employed with ease, as if intuitive or as if a good fit, the interaction of aspect-oriented programming and object-oriented programming appears to be transparent. Future work in aspect-orientation can benefit from attention to the right coupling between developer and task that is defined here by the AOPy project.
In particular, the AOPy research project documented in this dissertation suggests several ways that software developers can proceed in the future of aspect-oriented programming.

The compatibility framework itself can be used to evaluate the software development process and the developers' practices. If a project manager recognizes the appropriate use of aspect-oriented ideas, terminology, and techniques in the team's development process, then the manager can see where aspect-orientation and object-orientation are working together. Moreover, when scattered concerns are known to exist, that manager can use the compatibility framework described in this dissertation to understand why potential compatibility has been thwarted.

In addition, non-compatibility—an appropriate or potential but thwarted usefulness—must be addressed if aspect-orientation is to flourish in practice. The AOPy project suggests that the anthropomorphism of object-orientation, for instance, can thwart compatibility. Additional metaphors can be developed for aspect-orientation that leverage those metaphors that are already prevalent in the object-oriented development process. In other words, aspect-orientation might extend the metaphors already used by object-oriented developers.

In addition, aspect-orientation must not be put into direct competition with object-orientation. An important conclusion of this dissertation's research is that, when subjects in the AOPy experiment articulated a direct comparison between the two alternatives, the dominant methodology of object-orientation was likely to
prevail. Also, when subjects articulated an object-centric approach, they were unlikely to recognize scattered or crosscutting concerns, let alone exhibit ideas, terms, or techniques that indicated aspect-orientation was useful. Presumably, as developers become more used to considering primary and secondary concerns—considering both what objects encapsulate neatly or very effectively as well as what is scattered disparately throughout classes—aspect-orientation will become more compatible and, therefore, more useful in practice. New metaphors that take aspect-orientation out of competition with object-orientation and that downplay object-centricism might be developed to assist developers in recognizing secondary or crosscutting concerns and in recognizing that aspects can encapsulate these concerns usefully.

Overall, the AOPy project recognizes that potential usefulness alone does not ensure the success or growth of aspect-orientation. Just because an aspect can be used to encapsulate a concern does not ensure that a developer will use an aspect to encapsulate that concern, particularly because the concern can already, though poorly, be encapsulated by an object. Software developers, then, need to recognize the validity of separation of concerns and potential usefulness of aspect-orientation appropriately and then be able to actualize that potential usefulness through aspect-oriented ideas, terms, and techniques. This recognition and actualization process of aspect-orientation must be built in conjunction with, but not in competition with or subsumed by, object-orientation.
When early parts of the AOPy research project were presented at conferences, the most common question was “Why wasn’t this work done in AspectJ?” This query reflects the pragmatic nature of the aspect-oriented software development community. While Python worked well for this research project, the question also suggests a future direction for research into the relationships between aspect- and object-orientation by using AspectJ. Additional research could solidify and expand the compatibility framework defined here in the AOPy project by applying the approaches documented in this dissertation to other areas.

Finally, the successful development and use of grounded theory techniques in the AOPy project suggests that other qualitative data analysis techniques might be applied to the gathered data in this field. If one wanted to account especially for the linguistic nature of software development—instead of for the social nature as was done in the AOPy project—techniques inspired by textual analysis such as semiotics and hermeneutics, for example, could be beneficial in understanding the relationship between aspect- and object-orientation. Or, for instance, software development has not yet been adequately studied in terms of the rhetorical contexts that developers establish and work within. Research methods that more fully account for the larger contexts in which software is developed and for software development as an ongoing process might be used in future investigations. In addition, though the AOPy project replicated software development teams carefully, future research might make more
extensive use of fieldwork in data collection than academic research in the field currently does.

The AOPy project and this dissertation that documents the research project encourage wider use of qualitative research methodology throughout a project. Wendy Bishop calls this sort of research "the emergent, ethnographic tradition" (1999). She advocates, in referring to the 1985 work of Michael Kamil, Judith Langer, and Timothy Shanahan, the following characteristics of such research:

1. phenomenological and seeks to understand human behavior from participants' frame of reference
2. systematically observes recurring patterns of behavior as people engage in regularly occurring activities
3. uses field settings and develops hypotheses grounded in events and driven by the conceptual framework of the study
4. confirms across a variety of information sources, contexts, times.

Though qualitative research need not negate quantitative approaches and though local circumstances demand adaptation in ethnographic research design, this sort of qualitative approach to future research in computer science can help researchers reframe important questions and also account for the social nature of software development. Future progress in the field, then, may be made in experimental design and methodology as well as in results themselves, as this dissertation demonstrates.
As a whole, the AOPy research project that is documented in this dissertation reaches valuable conclusions about the validity of aspect-orientation to address issues of software complexity and about the usefulness of aspect-orientation in practice. The project offers directions for future work in the exploration of the relationship between aspect-oriented programming and object-oriented programming so that compatibility can be encouraged in practice and non-compatibility can be avoided. And it offers a model for and encouragement of more extensive use of qualitative research methodology in computer science.

7.4 Conclusion

Solid criteria exist for judging qualitative research such as the one documented in this dissertation. The AOPy project is consistent with existing literature on computer science but adds a new understanding of the role of crosscutting concerns in the practice of aspect-oriented programming by object-oriented developers who want to integrate it into their development processes. The research made unexpected discoveries about aspect-orientation as it is used by development teams but did not contradict established knowledge in the field. The AOPy project’s elements—from the literature review to the research methodology to the experiment design to the grounded theoretic analysis—work together as a comprehensive, complementary whole. Therefore, the AOPy project and this dissertation that documents that project have necessary internal integrity. The
conclusions of the AOPy project are relevant to a clearly defined audience within the profession, in that the conclusions explore the usefulness and potential usefulness of aspect-orientation in practice by software developers, as well as suggesting areas of future academic research.
BIBLIOGRAPHY

A


B


Bishop, W., 1999, *Ethnographic Writing Research*, Heinemann Boynton-Cook, Portsmouth, NH.


C


E


F


G

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.


Korienek, G., Wrensch, T., and Dechow, D. 2002, Squeak: A Quick Trip to ObjectLand, Addison-Wesley, Reading, MA.


L


Laddad, R. 2003, AspectJ in Action, Manning Publications, Greenwich, CT.


Lieberherr, K. and Holland, I. 1989, 'Assuring Good Style for Object-Oriented Programs', *IEEE Software*, vol. 6, no. 5.


M


Mili, H., Harrison, H., and Ossher, H. 1996, 'Supporting Subject-Oriented Programming in Smalltalk', Technology of Object-Oriented Languages and Systems (TOOLS '96), Santa Barbara, CA.


Q

R


APPENDICES
Knox College Computer Science Program

Requirements for the major

13 credits as follows:

1) Introductory courses: CS 141, CS 142
2) Intermediate courses: CS 201, CS 205, CS 226, CS 262, and CS 292
3) Mathematics: MATH 175 plus two additional mathematics courses chosen from the following: MATH 151, 160, 180, 210, 214, 216, 217, 300. (MATH 121, MATH 131 and MATH 140 may not be used to fulfill this requirement, but courses from departments other than Mathematics may be considered where a case can be made that they provide the most appropriate background for the student's future work)
4) Advanced study: three additional courses chosen from CS/PHYS 242, CS 303, CS 305, CS 306, CS 308, CS 310, CS 317, CS 320, CS 322, CS 330, CS 340, CS 360, CS 395, CS 399, MATH 311.
5) Capstone experience: After completion of CS 292, students must engage in a capstone experience during their senior year resulting in a written report and an oral presentation. Students may select from
a. completing a College Honors project;
b. completing a one-term senior research seminar (CS 399), which may also count as an elective;
c. completing CS 322 Software Engineering, which may also count as an elective;
d. completing a full-credit independent study or topics course, which may also count as an elective.

**Computer Science Course Descriptions**

**CS 127 Computer Science for the Arts and Sciences**

Introduction to computers and an overview of computer science for students with little or no background in computer science. Topics include history and future of computing, computer hardware, information storage and retrieval, operating systems, networking and the World-Wide-Web, and an introduction to structured problem-solving in a high-level programming language.

**CS 141 Introduction to Computer Science (1)**

An introduction to the fundamental principles of computer science, with emphasis on problem solving techniques, data and procedural abstraction, and use of algorithmic
thinking to understand, decompose and translate problem descriptions into sound, machine-executable solutions. Rudiments of computer functions, data types, control structures and program design considerations, including object-oriented concepts such as modularity, encapsulation and class.

CS 142 Program Design and Methodology (1)
Continued study of principles of computer science, structured programming, object-oriented programming, and algorithmic languages. Introduction to data structures, algorithms and their complexity analysis, advanced problem solving involving recursion and iteration, software engineering concepts, design, implementation, and maintenance of large programs. MNS; Prereq : CS 141 or permission of the instructor.

CS 180 Programming Language and Tools Workshop (1/2)
Students will study programming languages and development environment topics. This course will be offered as needed to support the Computer Science curriculum. Programming languages offered may include, but are not limited to: Lisp, Scheme, Prolog, C, Python, Perl. Tools offered may include Linux/Unix system administration, shell programming, and OpenGL. Prereq : CS 142 or permission of the instructor.
CS 201 Computer Organization and Assembly Language (1)

Computer organization and assembly language programming, machine language, arithmetic and logical operations, indexing and indirect addressing, subroutines, pipelining, memory hierarchy, input/output devices, buses, control units, secondary storage techniques. Prereq: CS 142 or permission of the instructor.

CS 205 Algorithm Design and Analysis (1)

Advanced data structures and analysis of algorithms and their complexity. Trees, graphs, hashing, analysis of sorting algorithms, divide and conquer algorithms, data structures for external search and sort, development of complex abstract data types typically with an object-oriented approach. Prereq: CS 142 and MATH 175 or permission of the instructor.

CS 206 Foundations of Computing (1)

Automata theory (finite machines, regular expressions, context-free languages, an introduction to lexical analyzers and parsers, etc.); computability theory (decidable and undecidable languages, problems that are solvable and not solvable by computers); and complexity theory (time and space complexity of algorithms, NP-completeness, PSPACE-completeness, intractability). Prereq: CS 205 or permission of the instructor.
CS 208 Programming Language Concepts (1)

A critical study of the design issues that underlie modern programming languages. The course includes the study of lexical, syntactic, and semantic analysis and examines the important programming paradigms, including imperative, functional, logic, and object-oriented. Prereq: CS 205 or permission of the instructor.

CS 226 Operating Systems and Networking (1)

Covers the fundamentals of operating systems and the basics of networking and communications. Topics include process and memory management, concurrency, process synchronization and scheduling, network architectures, simple network protocols, and APIs for network operations. Prereq: CS 201.

CS 242 Digital Electronics (1)

Theory and practice of the implementation of digital logic circuits from small scale integrated devices to microprocessors. Three class meetings and one double-period laboratory a week. Prereq: CS 201, PHYS 130 or PHYS 130A; Cross Listing: PHYS 242.

CS 248 Teaching Assistant (1/2 or 1)

Prereq: Permission of instructor.
CS 250 Independent Study (1/2 or 1)

CS 262 Information Management (1)
Uses the idea of information as a unifying theme to investigate a range of issues in
database systems, artificial intelligence, and data communications. Topics covered
include information models and systems, database systems, relational databases,
knowledge representation and reasoning, intellectual property, and privacy and civil
liberties. Prereq : CS 205.

CS 292 Software Development and Professional Practice (1)
Covers elementary topics in software engineering essential to the design and
development of larger software projects. Topics include requirements management,
design, software evolution, testing, and project management. Students typically work
in teams on a medium-sized software project. Issues of social responsibility,
intellectual property, copyright, and assessing the risks in computer systems are
discussed. Prereq : CS 205.

CS 295 Special Topics (1/2)
Courses offered occasionally to students in special areas of Computer Science not
covered in the usual curriculum.
CS 303 Computer Graphics (1)

Mathematical theories, algorithms, software systems, and hardware devices for computer graphics. Translation, rotation, scaling, projection, clipping, segmented display files, hidden line and surface elimination, surface texturing, point plotting display, raster display, random stroke display, input of graphical data. Prereq: CS 205 and MATH 141 (or equivalent).

CS 305 Operating Systems (1)

Advanced management of computer resources such as storage, processors, peripheral devices, and file systems. Storage allocation, virtual memory, scheduling algorithms, synchronization, mutual exclusion, deadlock, concurrent programming, processes, inter-process communication, protection, operating system organization. Prereq: CS 201 and CS 205.

CS 306 Automata Theory and Programming Languages (1)

Automata theory (finite machines, Turing machines, regular expressions, context-free languages); computability theory (decidable and undecidable languages, problems that are solvable and not solvable by computers); complexity theory (time and space complexity of algorithms, NP-completeness, intractability); a critical study of the design issues that underlie modern programming languages including the
study of lexical and syntactic analysis and the important programming paradigms.  
Prereq: CS 205 or permission of the instructor.

CS 308 Networks and Distributed Systems (1)
Covers advanced topics in computer/data networking. Topics include media types,  
network architectures, common networking practices and components, network  
design fundamentals, network management technologies and practices, and an  
introduction to various service and maintenance protocols (IP, DNS, DHCP, WINS,  
etc.). Prereq: CS 226.

CS 310 Compilers and Interpreters (1)
Theory and practice of computer programming language translation. Lexical  
analysis, syntax analysis, finite state automata, parsing methods, error handling, error  
recovery, compiler organization, interpretation, intermediate languages, code  
generation and optimization techniques. Prereq: CS 201 and CS 205; CS 306 is  
recommended.

CS 317 Artificial Intelligence (1)
A survey of topics in the branch of computer science concerned with creating and  
understanding "intelligent" computer systems, including advanced search techniques  
and heuristics, knowledge representation, expert systems, natural language
processing, machine learning, and game playing. Topics will also include the study of the nature of intelligence and the representation of intelligent machines in fiction. Prereq: CS 262 or permission of the instructor.

**CS 320 Database Systems (1)**

Theory and management of database management systems, including database models, design principles, file organizations, data structures and query organization for efficient access, query languages, database-interface applications, normalization and relational concepts such as views, procedural database programming and referential integrity. Prereq: CS 262.

**CS 322 Software Engineering (1)**

Building large-scale computing systems uses requirements analysis, project planning, extensive documentation, cooperative teamwork, and design techniques to decompose a system into independent units. The course covers all the phases of large-scale system development. Different development models are examined including the waterfall model, the spiral mode, rapid prototyping, and extreme programming. Students typically work together in teams to build a term-long project, gaining practical experience with developing larger systems. Prereq: CS 292.
CS 330 Cryptography and Computer Security (1)

With the increasing ubiquity of computers and computer networks, issues of privacy and security are becoming increasingly important for computing professionals. This course introduces students to a number of related areas in computer security. Topics covered include classical cryptography, public-key cryptography, block and stream ciphers, file system security, network security, Internet and web-based security, and design principles behind cryptographic systems. In addition, the course examines social, political, legal, and ethical issues related to security systems. Prereq: CS 205 and CS 226.

CS 340 Human-Computer Interaction (1)

As computing becomes more pervasive, there is a growing need to understand the point where humans and machines connect. This course is a survey of topics that arise from examination of this connection. Topics include user interface design, usability analysis, scientific visualization, novel interfaces, and an exploration of what happens when it all goes terribly wrong. Prereq: CS 205.

CS 348 Teaching Assistant (1/2 or 1)

Prereq: Permission of instructor;

CS 350 Independent Study (1/2 or 1)
CS 360 Natural Language Processing (1)

Getting computers to process human language intelligently was one of the earliest goals in computer science, and the task continues today. This course gives a survey of the area, including both 'pure' topics like morphological analysis and parsing, and applications, such as machine translation, question answering, and dialogue systems. There is a strong emphasis on the recent shift toward statistical methods. Prereq : CS 205.

CS 395 Special Topics (1/2 or 1)

Courses offered occasionally to students in special areas of Computer Science not covered in the usual curriculum.

CS 399 Research Seminar in Computer Science (1)

An advanced study of a special topic in computer science not substantially covered in the regular curriculum. Resources are usually drawn from the current computing literature. Emphasis is on student presentations and independent writing and research. Students submit a major paper and give a public lecture. Prereq : senior standing;

CS 400 Advanced Studies (1/2 or 1)

See College Honors Program.
Notes Regarding Knox Computer Science Courses and Students

1) The Knox College introductory computer science curriculum was revised during the time of two of the subjects: 4 and 5. At this time, the language taught in CS 141: Introduction to Computer Science I was changed from Scheme to Java. One of the subject's, number 4, took a transitional course in Java programming. The language used in CS 142: Introduction to Computer Science II and CS 206: Data Structures is C++. As a result of these language choices, even the subjects with the least amount of exposure to computer science coursework—subjects 6, 7, and 8—were introduced to object-oriented programming in Java and C++.

2) Subject number 2 opted not to participate in the concluding experiment. Subject number 2's background data is presented for the sake of completeness and because the subject numbers were assigned at the beginning of the seminar.
The following is expurgated syllabus information provides the context for the background of subjects in the CS 582 experiment.

**OSU CS 582, Winter 2004**

**Object-Oriented Analysis and Programming**

**Professor Timothy A. Budd**

[http://classes.engr.oregonstate.edu/eecs/winter2004/cs582/](http://classes.engr.oregonstate.edu/eecs/winter2004/cs582/)

**What is this course?**

The catalog description for this course reads as follows:

"An examination of the ideas of Object-Oriented design and Object-Oriented software construction. The lectures present the concepts of object-oriented analysis and programming in a language independent fashion, illustrated with specific examples from a variety of programming languages. Students are expected to complete and report on a nontrivial project developed in an object-oriented language of their choice."

As indicated in the description, the course is intended to be a broad introduction to the field of object oriented analysis and programming. It is not a course in any
particular programming language, although students are expected to have the ability to learn at least one language on their own during the period of the course.

What Background is Necessary for the Course?

The course is open to any student in the graduate program in computer science at Oregon State University. Undergraduates and students in other departments can take the course with the permission of the instructor. Although the course is listed as a graduate level course (for historical reasons), the material should be accessible to any upper division undergraduate with a background in computer science.

External students are expected to have at least two years programming experience in a conventional programming language, such as C or Pascal. No previous programming experience in Object Oriented languages is assumed.

Students should expect to spend about ten to fifteen hours per week on course material. This includes reading the text, answering the study questions, doing exercises, the midterm exam, and programming the final project.

The Textbook

This course is based around my book, *An Introduction to Object-Oriented Programming, 3E* (published by Addison-Wesley Longman, 2002). You should be
able to find a copy of this book at any good technical bookstore (such as Powells in Portland), or you can order it on-line. In the past the on-line prices were often less than the OSU bookstore price, but now they seem very similar.

A variety of information for the textbook, include an errata sheet, is available on-line.

In addition to the textbook, students will likely want to purchase a reference manual for whatever language they are using for their project. The OSU bookstore has quite a range of titles, as do many other technical bookstores. If asked (in the discussion group) I can suggest a few titles.

Calendar of Topics

The following calendar lists the intended dates for discussing specific sections of the text, as well as hyperlinks to the associated lectures and audio files. (Audio files not yet available for the 3rd edition, sorry). Participants are asked to try to stay as close as possible to this schedule.

A table of Study Questions To Date is available.
Week 1

* Introductions, Getting Started, Learning how to use the Web.
* Chapter 1 - Thinking Object Oriented
* Chapter 2 - Abstraction

Week 2

* Classes, Methods and Messages
* Chapter 3 - Object-Oriented Design
* Cybervid Exercise (text, pdf, html) Picture of kiosk.
* Chapter 4 - Classes and Methods
* Chapter 5 - Messages, Instances and Initialization

Week 3

* Inheritance and Substitution
* Chapter 6 - A Case Study: Eight Queens
* Chapter 7 - Case Study: A Billiards Game
* Chapter 8 - Inheritance
* Chapter 9 - Case Study: Solitaire

Week 4

* More on inheritance
Chapter 10 - Subclasses and Subtypes

Chapter 11 - Static and Dynamic Behavior

Chapter 12 - Implications of Substitution

Chapter 13 - Multiple Inheritance

Week 5

Polymorphism

Chapter 14 - Polymorphism and Software Reuse

Chapter 15 - Overloading

Chapter 16 - Overriding

Chapter 17 - The Polymorphic Variable

Chapter 18 - Generics

MIDTERM EXAM

Week 6

Applications of Polymorphism

Chapter 19 - Container Classes

Chapter 20 - The STL

Chapter 21 - Frameworks

Project Description and Examinations Due by November 5
Week 7

* Software Engineering Topics
* Chapter 22 - The AWT and Swing
* Chapter 23 - Object Interconnections
* Chapter 24 - Design Patterns

Week 8

* Advanced Topics
* Chapter 25 - Reflection and Introspection
* Chapter 26 - Distributed Objects

Week 9

* Chapter 27 - Implementation

Week 10

* Project Demonstrations, Evaluation, Wrap up Discussions
* All course related material due by March 12.
1. Templating is the process of defining a block of text that contains embedded variables, code and other markup. This text block is then automatically processed to yield another text block, in which the variables and code have been evaluated and the results have been substituted into the text. Most dynamic web sites are generated with the help of templating mechanisms.

Below is the template engine/processor for Snakelet's. Imagine that you have just added this module to the system. Unfortunately, your system is unable to generate html. Describe how you could use the techniques of Aspect-Orientation to find the problem in your system. Feel free to run Snakelets and Pythius in exploring your solution. Sketch your solution with a pencil and paper mock-up.

```python
import sys, re
import cStringIO
import cgi
```
class TemplateProcessor:
    def __init__(self, response):
        self.response = response  # the snakelet Response object, for headers/cookies
        self.matcher = re.compile(r""
                                  (.*?)          # any characters before the tag (non-greedy)
                                  (<%=?)         # <%= or <%==
                                  (.*?)          # the characters inside the tag (non-greedy)
                                  %>             # closing tag %>
                                 "", re.VERBOSE | re.DOTALL)

    def processToStream(self, file, environ={}):
        template = file.read()
        begin, end = 0, 0
        self.s_out = cStringIO.StringIO()

        s_out_orig = sys.stdout
        s_err_orig = sys.stderr
sys.stdout=self.s_out
#sys.stderr=s_out
self.must_abort=0

s = self.matcher.match(template)
while s and not self.must_abort:
sys.stdout.write(s.group(1))  # the preceding HTML
begin,end = s.span()
tag = s.group(2)
script = s.group(3)+'
'  
try:
    if tag=='<%':
        exec (script, environ)
    elif tag=='<%=':

sys.stdout.write(str(eval(script, environ)))

except:
    import traceback
    (etype, evalue, etb) = sys.exc_info()
    descr =
    traceback.format_exception(etype,evalue,etb)
    print
"<hr><strong><pre">+cgi.escape(''.join(descr),1)+"</pre></strong><hr
>"
s = self.matcher.match(template, end)
if not self.must_abort:
    sys.stdout.write(template[end:])  # the trailing HTML
sys.stdout = s_out_orig
sys.stderr = s_err_orig
self.s_out.flush()
self.s_out.seek(0)
return (self.s_out, self.response)

def write(self, msg):
    self.s_out.write(str(msg))

def abort(self, msg=' '):
    print '<hr><strong>Page aborted</strong>', msg, '</strong>'
    self.must_abort = 1

def __getattr__(self, name):
    # pass any other methods to the response object
    return getattr(self.response, name)

if __name__ == '__main__':
processor = TemplateProcessor()
result = processor.processToStream(open(sys.argv[1]))
.getvalue()
print 'Done.'
print 'result=', len(result)
open("x.html", 'wb').write(result)

2. Below is the source code for the Snakelet's server class (it can be found in
~/.Snakelets-0.3/snakeserver/server.py). Using your knowledge of the Aspect-
Orientation paradigm: 1. attempt to identify a concern that could be separated out
into an aspect; and 2. sketch a design for such an aspect.

#
# The threading HTTP server.
#
class ThreadingServer(SocketServer.ThreadingMixIn,
BaseHTTPServer.HTTPServer):

def __init__(self, address, handler):
    self.servername=address[0]
    BaseHTTPServer.HTTPServer.__init__(self, address, handler)
def server_activate(self):
    # initialize the server
    self.webApps = {}
    self.rootWebApp = None  # webapp that handles '/'
    BaseHTTPServer.HTTPServer.server_activate(self) # activate socket listener
    self._hostname_cache = {}

def getHostName(self, ip):
    # get the FQ hostname that belongs to the specified IP address
    if self._hostname_cache.has_key(ip):
        return self._hostname_cache[ip]
    else:
        x=socket.getfqdn(ip)
        self._hostname_cache[ip]=x
        return x

def readWebApps(self):
    # scan web applications
    for web in os.listdir("webapps"): 
if os.path.isdir(os.path.join("webapps",web)):
    self.readWebApp(web)

try:
    self.rootWebApp=self.webApps['/']
    print 'ROOT webapp: ',self.rootWebApp.abspath
except KeyError:
    print "No ROOT webapp for url '/' has been defined!"
    raise SystemExit(1)

print len(self.webApps), "webapps registered."

def readWebApp(self, web, mustReload=0):
    # add the webapps dir to the module search path.
    # (only if we're not reloading)
    abspath=os.path.abspath(os.path.join("webapps",web))
    if not mustReload:
        sys.path.append(abspath)
    else:
        # reloading. first, remove the old webapp
        name=os.path.split(web)[1]
        url='/'+name+'/'
        web=name
        self.unloadWebApp(url)

        try:
            exec "import webapps."+web+" as WA"
if not hasattr(WA,"configItems"):
    WA.configItems={}
if not hasattr(WA,"dirListAllower"):
    WA.dirListAllower=None
url='/'+web+'/'
if hasattr(WA,"ROOT_WEBAPP") and WA.ROOT_WEBAPP:
    url='/'
try:
    wa=webapp.WebApp(abspath, WA.name, url, WA.docroot, WA.snakelets, WA.configItems, WA.dirListAllower, self)
except AttributeError,x:
    # XXX weird?? occurs when reloading server?
    print "!!! problem during webapp load:",x
    print "!!! webapp="",url
    import traceback
    traceback.print_exc()
else:
    if self.webApps.has_key(url):
        print "Duplicate webapp url: ",url,
        (webapp="",web,"")
        raise SystemExit(1)
    self.webApps[url]=wa
    print "WEBAPP",web,
    (",len(wa.snakelets),"snakelets )"
    print " name =",WA.name
print " url =",url

except ImportError,x:
    # not a correct webapp
    print "!!! No correct config found for webapp",web
    print " ",x

def getWebApp(self, url):
    # find the webapp that handles this url
    for web in self.webApps:
        if url.startswith(web):
            webapp=self.webApps[web]
            if webapp.enabled:
                return webapp
            else:
                break
    # fallback to ROOT webapp
    return self.rootWebApp

def enableWebApp(self, url, enabled):
    # enable/disable the webapp for this url prefix
    self.webApps[url].enabled=enabled

def reloadWebApp(self, url):
    # reload the webapp for this url prefix
    web=self.webApps[url].abspath
self.readWebApp(web, 1)

def unloadAllWebApps(self):
    for web in self.webApps.keys():
        self.unloadWebApp(web)

def unloadWebApp(self, web):
    name = os.path.split(self.webApps[web].abspath)[1]
    modulename = "webapps." + name
    for n in sys.modules.keys():
        if n.startswith(modulename) and type(sys.modules[n]) is types.ModuleType:
            del sys.modules[n]
    del self.webApps[web]

def reloadAllWebApps(self):
    self.unloadAllWebApps()
    self.readWebApps()

def getWebRoot(self):
    return self.rootWebApp.abspath

def setServerName(self, servername):
    self.servername = servername

def shutdown(self):
    self.mustShutdown = 1
def restart(self):
    self.mustShutdown = 1
    self.mustRestart = 1

def serve_forever(self):
    import select
    self.mustShutdown = 0
    self.mustRestart = 0
    while not self.mustShutdown:
        ins, outs, excs = select.select([self], [], [self], 5)
        if self in ins:
            self.handle_request()
            self.reapSessions()
        print "Shutting down gracefully."

def reapSessions(self):
    # close and delete all web sessions that have timed out.
    for webapp in self.webApps.values():
        webapp.scanSessionTimeouts()

    # Start everything!
def main(HTTPD_PORT=80, servername='localhost', bindname=None):
    if bindname is None:
        bindname = servername
    print 'Creating server on', bindname, ' (servername= ' + servername + ')
    httpd = ThreadingServer( (bindname, HTTPD_PORT),
                          MyRequestHandler)
    httpd.setServerName(servername)
    # read and initialize the webapps with their snakelets
    httpd.readWebApps()

    print 'WEBROOT=', httpd.getWebRoot()
    print "Serving HTTP on port", HTTPD_PORT
    while 1:
        httpd.serve_forever()
        if not httpd.mustRestart:
            break
        print "RESTARTING SERVER"
        httpd.reloadAllWebApps()

if __name__=='__main__':
    main(80)
APPENDIX D: CS 582 Experiment Workshop Syllabus

CS 582: Advanced Separation of Concerns Workshop

Instructor
Douglas R. Dechow
Email: dechow@cs.orst.edu
Web Page: http://www.cs.orst.edu/~dechow/cs582-aop

Workshop Time and Location
Class Times:
* March 1: 7-9 pm
* March 2: 7-9 pm
* March 3: 7-9 pm
* March 8: 7-9 pm
* March 9: 7-9 pm
* March 10: 7-9 pm
* March 11, 12, or 13: TBA

Location: Dearborn 401/Dearborn 115
Workshop Description

This workshop provides computer science students with an overview of current thinking in advanced separation of concerns. Students are introduced to relevant concepts in object-oriented and aspect-oriented programming and to the programming language Python. Most current work in aspect-oriented programming has been done in system programming languages, notably Java. With the rise of web application services, lightweight, dynamic languages (scripting languages) have seen significant increase in use.

The workshop will start with an (independent) introduction to the Python programming language. The idea of separation of concerns and the techniques of aspect-oriented programming will be introduced and related to appropriate application domains.

The workshop culminates with each student solving a given separation-of-concerns problem and then talking with the instructor about the process of solving the problem. This type of experiment is known as a 'Thinking-Aloud' study.

Additionally, students are required to prepare a final paper in accordance with the description provided in the CS582 course syllabus.
Finally, you are expected to have completed chapters 1-4 of http://diveintopython.org/ by the start of the workshop: Monday, 1 March 2004.

Contents of this web page:

* Syllabus change log
* Learning Objectives
* Tutorials, Books, and Other Materials
* Prerequisites
* Schedule
* Attendance Policy
* Interesting links, etc.

Learning Objectives

On completion of the workshop, students will demonstrate:

* an understanding of Python
* a familiarity with separation of concerns in the context of software development
* familiarity with the need for advanced separation of concerns techniques
* an understanding of some of these techniques, such as mix-ins, dynamic proxies, and aspect-oriented programming (AOP)
an understanding of the use of these techniques in the creation of problem solutions within a variety of problem domains

Tutorials, Books, and Other Materials

Tutorials

* [http://diveintopython.org/](http://diveintopython.org/)
* [http://www.pentangle.net/python/handbook/](http://www.pentangle.net/python/handbook/)

Books

Officially, we will not be using a textbook for this workshop. There are plenty of online materials. Nonetheless, the books listed below contain solid introductions to Python if you want a reference book.

* *Learning Python, 2ed.* by Mark Lutz, David Ascher, O'Reilly & Associates 1999
* *Python Programming on Win32* by Mark Hammond, Andy Robinson, O'Reilly & Associates 2000
* *Core Python Programming* by Wesley J. Chun, Prentice Hall PTR 2000
Other Materials

Other software systems such the Snakelet's web application server may be used in conjunction with the workshop.

A copy can be downloaded at Snakelets

Prerequisites

Concurrent enrollment in CS582 or familiarity with an object-oriented programming language, preferably Java.

Class Schedule

The following table gives a general outline for the workshop.

<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>Topics Assignments &amp; Homework</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Feb 16</td>
<td>Read Ch 1. Installing Python</td>
</tr>
<tr>
<td></td>
<td>Feb 17</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Feb 18</td>
<td>Read Ch 2. Getting To Know Python</td>
</tr>
<tr>
<td></td>
<td>Feb 19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Feb 20</td>
<td>Some problems to think about</td>
</tr>
<tr>
<td>2</td>
<td>Feb 23</td>
<td>Read Ch 3. The Power Of Introspection</td>
</tr>
<tr>
<td></td>
<td>Feb 24</td>
<td></td>
</tr>
</tbody>
</table>
Feb 25 Read Ch 4. An Object-Oriented Framework

Feb 26

Feb 27 Some problems to think about

3 Mar 1 Introduction Advanced Separation of Concerns & Aspect-Oriented Programming

Class slides: aop-intro.ppt  Skills & experiences survey

Code examples: AccountClass, AttrExample1, AttrExample2

Linux Journal: article

Mar 2 ASOC Techniques: Dynamic Proxies & Mix-Ins

Class slides: mixins.ppt

Code examples: QuickAccount, Lister, testmixin, diamond, ProxyInterfaceOf, FooBar, classadopt

Mar 3 More Advanced Separation of Concerns

Weekend homework

Code examples: AccountClass, AOPy, AOPyTraceAspect, TestAOPy, Webparser

Mar 4

Mar 5

4 Mar 8 Aspect-Oriented Programming

Mar 9 Aspect-Oriented Programming

Mar 10 Aspect-Oriented Programming
Attendance Policy

All participating students must be in attendance on Monday, March 1, 2004. Because this workshop is designed as a small, often collaborative investigation and because of the schedule, it is crucial that all students attend every class.

Complete attendance is also necessary for participation in the Final Exam/'Thinking-Aloud' Experiment. If you must miss a class for hospitalization, death in family, or a sanctioned Oregon State University event in which you are required to participate, you must notify me immediately to see if accommodations can be made for an equivalent make-up session.
1. Described below is a Python module containing a variety of classes. You can test the module from the Python command line by entering the code shown in "__main__". You can also execute the module as a script: $> python swt.py.

Analyze the software by describing all of the concerns that are present in the module. Are any of the concerns crosscutting (tangled, orthogonal to the primary concerns, etc.)? If so, based on the techniques that were covered in the workshop, describe alternative solutions for separating the concerns.

Note: Do not worry about producing executable software. A pencil and paper description of your solution is sufficient.
**"SWT: Simulated Windowing Toolkit**

This module is a simulation for implementing a trivial non-gui-based windowing framework.

When creating new widgets, you can think of an SWT object's initialization process as analogous to the registration phase that you see in other windowing toolkits.

In the Java AWT you might see something similar to the following:

```java
public static void main(String[] args) {
    Frame frame = new Frame('My New Frame');
    Panel panel = new Panel(frame);
    Button button = new Button('Click me!', panel);
    frame.getContentPane().add(panel);
    panel.add(button);
    frame.pack();
    frame.setVisible(true);
}
```
Using Python's built-in Tk toolkit you might see something like this from the interpreter prompt:

```python
>>> import Tkinter
>>> root = Tkinter.Tk()
>>> LabelText = 'My New Root Window'
>>> LabelWidget = Tkinter.Label( root, text = LabelText )
>>> LabelWidget.pack( )
>>> ButtonWidget = Tkinter.Button( root )
>>> ButtonWidget['text'] = 'Click me!'
>>> ButtonWidget.pack( side = Tkinter.LEFT )
```

In SWT, most of this work is simulated through initialization. You can see an example of this by looking at the code in the test script section.

Below are brief descriptions of the main classes that make-up the SWT.

**FRAME**

The class SWT_Frame is a top-level window—i.e., a window that is not contained inside any other window.
* __init__ --
* handleRequest -- overridden method; handled here
* add -- add an SWT component/widget to this frame
* pack -- tells the layout manager to handle the geometric positioning of a component
* setVisible -- shows or hides the component depending on whether b is true or false

PANEL
The class SWT_Panel represents a rectangular area of the screen contained in other PANELS or in a FRAME. A PANEL's function is to contain other widgets (like a button).

* __init__ --
* handleRequest -- overridden method; not handled here
* add -- add an SWT component/widget to this frame

BUTTON
The SWT_Button class is a widget that can be 'clicked'. You can simulate this by
sending it a click() message.

* __init__ --
* handleRequest -- overridden method; not handled here
* click -- simulation of mouse depressing a button

HANDLER
You can think of the SWT_Handler class as an interface (or role).
SWT classes
must override the handleRequest( ) method.

* handleRequest -- interface method for handling a request
resulting
   an event being caught

MESSAGE
The SWT_Message class is used to represent any piece of data that is
passed around the windowing system.

* __repr__ -- the data is the message object's representation

```python
class SWT_Message:
    def __init__( self, msg ):
```
self.data = msg
def __repr__(self):
    return self.data

class SWT_Handler:
    def __init__(self):
        pass
def handleRequest(self, swt_Message):
    pass

class SWT_Frame(SWT_Handler):
    def __init__(self, swt_NameString):
        self.successor = None
        self.name = swt_NameString
def handleRequest(self, swt_Message):
    print "Request received by: SWT_Frame ( handled )"
    print "Message is: ", swt_Message
def add(self):
    pass
def pack(self):
    pass
def setVisible(self):
    pass

class SWT_Panel(SWT_Handler):
    def __init__(self, successor):
        self.successor = successor

    def handleRequest(self, swt_Message):
        print "Request received by: SWT_Panel (unhandled: forwarded)"
        self.successor.handleRequest(swt_Message)

    def add(self):
        pass

class SWT_Button(SWT_Handler):
    def __init__(self, successor):
        self.successor = successor

    def handleRequest(self, swt_Message):
        print "Request received by: SWT_Button (unhandled: forwarded)"
        self.successor.handleRequest(swt_Message)

    def click(self, swt_Message):
self.handleRequest(swt_Message)

if __name__ == "__main__":
    myFrame = SWT_Frame(SWT_Handler())
    myPanel = SWT_Panel(myFrame)
    myButton = SWT_Button(myPanel)

    # A more realistic toolkit would also have code similar to this...
    # myFrame.add(myPanel)
    # myPanel.add(myButton)
    # myFrame.pack()
    # myFrame.setVisible(true)

    # A more realistic toolkit would actually present a GUI window, etc.,
    # for the following task...

    myButton.click(swt_Message("Someone please handle this..."))
2. Described below is a Python module containing a variety of classes. You can test the module from the Python command line by entering the code shown in "__main__". You can also execute the module as a script: $> python btt.py.

Analyze the software by describing all of the concerns that are present in the module. Are any of the concerns crosscutting (tangled, orthogonal to the primary concerns, etc.)? If so, based on the techniques that were covered in the workshop, describe alternative solutions for separating the concerns.

Note: Do not worry about producing executable software. A pencil and paper description of your solution is sufficient.
In this example, Node, RegularNode and Leaf build up a binary tree that has int values as leaves. SumTraveller is a NodeTraveller that collects the sum of elements in the leaves (should be 6). TraversalTraveller collects a description of the tree like \[\{(1,2),3\}\]

# Think of Node as an interface

class Node:
    def accept( self, NodeTraveller ):
        pass

class Leaf( Node ):
    def __init__( self, value ):
        self.intValue = value

    def accept( self, nodeTraveller ):
        nodeTraveller.travelToLeaf( self )

    def getValue( self ):
        return self.intValue
class RegularNode( Node ):
    def __init__( self, left, right ):
        self.leftNode = left
        self.rightNode = right

    def accept( self, nodeTraveller ):
        nodeTraveller.travelToRegularNode( self )

    def getLeft( self ):
        return self.leftNode

    def getRight( self ):
        return self.rightNode

# another interface
class NodeTraveller:
    def travelToRegularNode( self, node ):
        pass

    def travelToLeaf( self, node ):
        pass

    def report( self ):
        pass
class SumTraveller( NodeTraveller):
    def __init__( self):
        self.intSum = 0

    def travelToRegularNode( self, node):
        rnode = node
        rnode.leftNode.accept( self)
        rnode.rightNode.accept( self)

    def travelToLeaf( self, node):
        leaf = node
        self.intSum += leaf.getValue()

    def report( self):
        return ">>> SumTraveller collected a sum of " + str(self.intSum)

class TraversalTraveller( NodeTraveller):
    def __init__( self):
        self.result = ""

    def travelToRegularNode( self, node):
        rnode = node
        self.result += "{"
        rnode.getLeft().accept( self)
self.result += ","
        rnode.getRight().accept(self)
        self.result += "}"

def travelToLeaf(self, node):
    leaf = node
    self.result += str(leaf.getValue())

def report(self):
    return ">>> TraversalTraveller traversed the tree to: " + str(self.result)

if __name__ == '__main__':
    print "Building the tree (1): leaves"

    one = Leaf(1)
    two = Leaf(2)
    three = Leaf(3)

    print "Building the tree (1): regular nodes"

    regN = RegularNode(one, two)
    root = RegularNode(regN, three)

    print "The tree now looks like this: "
print "regN" "
print "/ \" "
print "regN 3" "
print "/ \" "
print "1 2" 

print "Traveller 1: SumTraveller, collects the sum of leaf" 
print "values. Result should be 6."

sumTraveller = SumTraveller() 
root.accept(sumTraveller) 
print sumTraveller.report() 

print "Traveller 2: TraversalTraveller, collects a tree" 
print "representation. Result should be \{1,2\},3." 

traversalTraveller = TraversalTraveller( ) 
root.accept(traversalTraveller) 
print traversalTraveller.report( )