Concern-Driven Software Evolution

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Eugen C. Nistor

Dissertation Committee:
Professor André van der Hoek, Chair
Professor Cristina V. Lopes
Professor Susan Elliott Sim

2009
The dissertation of Eugen C. Nistor

is approved and is acceptable in quality and form for

publication on microfilm and in digital formats:

__________________________

__________________________

__________________________

__________________________

Committee Chair

University of California, Irvine

2009
DEDICATION

To my wife, Diana.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>VII</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>IX</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>X</td>
</tr>
<tr>
<td>CURRICULUM VITAE</td>
<td>XI</td>
</tr>
<tr>
<td>ABSTRACT OF THE DISSERTATION</td>
<td>XIII</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. BACKGROUND AND MOTIVATION</td>
<td>9</td>
</tr>
<tr>
<td>2.1. Definitions</td>
<td>9</td>
</tr>
<tr>
<td>2.2. Motivating Example</td>
<td>17</td>
</tr>
<tr>
<td>2.2.1. Selection of Concerns</td>
<td>19</td>
</tr>
<tr>
<td>2.2.2. Identification of concerns in code</td>
<td>22</td>
</tr>
<tr>
<td>2.2.3. Limitations</td>
<td>26</td>
</tr>
<tr>
<td>2.3. Concerns in Software Development</td>
<td>27</td>
</tr>
<tr>
<td>2.3.1. Properties of Concerns</td>
<td>27</td>
</tr>
<tr>
<td>2.3.2. Separation of Concerns and Modularization</td>
<td>32</td>
</tr>
<tr>
<td>2.3.3. Concerns in Software Evolution</td>
<td>38</td>
</tr>
<tr>
<td>2.3.4. Concern Identification</td>
<td>43</td>
</tr>
<tr>
<td>2.3.5. Concerns in Architecture</td>
<td>45</td>
</tr>
<tr>
<td>2.3.6. Architecture and Implementation</td>
<td>48</td>
</tr>
<tr>
<td>2.4. Ideal Support</td>
<td>50</td>
</tr>
<tr>
<td>3. RESEARCH QUESTIONS</td>
<td>52</td>
</tr>
<tr>
<td>4. APPROACH</td>
<td>54</td>
</tr>
<tr>
<td>Principles</td>
<td>55</td>
</tr>
<tr>
<td>4.1. Features</td>
<td>58</td>
</tr>
<tr>
<td>4.1.1. Concern Model</td>
<td>59</td>
</tr>
</tbody>
</table>
4.1.2. Concern Identification 66
4.1.3. Concern Visualization and Analysis 68
4.1.4. Evolution Support 73

5. ARCHEVOL 78

5.1. Eclipse 81

5.2. Internal Architecture 83
  5.2.1. Foundation 83
  5.2.2. Infrastructure 85
  5.2.3. Core 86

5.3. Features 87
  5.3.1. Development Context 87
  5.3.2. Architectural Development Support 89
  5.3.3. Concern Model 94
  5.3.4. Concern Visualization 101
  5.3.5. Evolution Support 108

5.4. Implementation Challenges 112

6. EVALUATION 116

6.1. Evaluation Goals 116

6.2. jEdit User Study 119
  6.2.1. Experiment setup 120
  6.2.2. Experiment procedure 124
  6.2.3. Data Analysis 131
  6.2.4. Threats to validity 152
  6.2.5. Conclusions 153

6.3. ArchStudio User Study 154
  6.3.1. Background 157
  6.3.2. Setup 158
  6.3.3. Preliminary session 159
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Examples of code fragments from jEdit.</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>Average and standard deviation for the size of source code fragments compared to the number of fragments for concerns.</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>Overlap of concerns over lines of code.</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>Modularity choices.</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>Distribution of source code fragments.</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>Ratio of average length and average difference to the nearest module boundary for source code fragments.</td>
<td>37</td>
</tr>
<tr>
<td>7</td>
<td>Evolution of the buffer concern in jEdit.</td>
<td>41</td>
</tr>
<tr>
<td>8</td>
<td>Evolution of number of classes for the buffer concern compared to the entire system.</td>
<td>42</td>
</tr>
<tr>
<td>9</td>
<td>Approach overview.</td>
<td>55</td>
</tr>
<tr>
<td>10</td>
<td>Approach features.</td>
<td>59</td>
</tr>
<tr>
<td>11</td>
<td>Example of links to software fragments.</td>
<td>65</td>
</tr>
<tr>
<td>12</td>
<td>Code visualization of scattering and tangling.</td>
<td>70</td>
</tr>
<tr>
<td>13</td>
<td>Metrics of concerns in the architecture.</td>
<td>73</td>
</tr>
<tr>
<td>14</td>
<td>Overview of the architecture and source code evolution support.</td>
<td>75</td>
</tr>
<tr>
<td>15</td>
<td>Visualization of evolution of concerns.</td>
<td>77</td>
</tr>
<tr>
<td>16</td>
<td>ArchEvol.</td>
<td>79</td>
</tr>
<tr>
<td>17</td>
<td>Regular development view of Eclipse.</td>
<td>83</td>
</tr>
<tr>
<td>18</td>
<td>ArchEvol plug-in architecture.</td>
<td>84</td>
</tr>
<tr>
<td>19</td>
<td>Eclipse project extensions to xADL2.0.</td>
<td>92</td>
</tr>
<tr>
<td>20</td>
<td>The architectural views and editor in ArchEvol.</td>
<td>93</td>
</tr>
<tr>
<td>21</td>
<td>xADL schema extensions for concerns.</td>
<td>95</td>
</tr>
<tr>
<td>22</td>
<td>The internal architecture of concern event architecture in ArchEvol.</td>
<td>96</td>
</tr>
<tr>
<td>23</td>
<td>Concern views in ArchEvol.</td>
<td>100</td>
</tr>
</tbody>
</table>
Figure 24. Visualization of concerns in code with the Concern Overview.

Figure 25. Visualization of concerns in the architecture.

Figure 26. Visualization of concerns in the History View.

Figure 27. Versioning extension to xADL2.0.

Figure 28. ArchEvol specific repository layout. Dotted lines are links maintained by ArchEvol.

Figure 29. Options dialog in jEdit.

Figure 30. Task 1 analysis results.

Figure 31. Task 2 analysis results.

Figure 32. Task 3 analysis results.

Figure 33. Task completion scores.

Figure 34. Navigation time versus performance.

Figure 35. Navigation and inspection time versus performance.

Figure 36. Navigation and Concern Model use time versus score.

Figure 37. Navigation and Concern Model time versus score for tasks 2 and 3.

Figure 38. Concern model maintenance results.

Figure 39. ArchStudio architecture.

Figure 40. User interface concern implementation in ArchStudio.

Figure 41. Example of a unified diff file.

Figure 42. Percentage of source code fragments manually corrected during the study.

Figure 43. Evolution of number of components and links in ArgoUML.

Figure 44. AgoUML's architecture.

Figure 45. Evolution history visualization for the diagram concerns (top) and the state machine concern (bottom).

Figure 46. Evolution history view showing diagram and state machine concerns (top) and state machine and association concerns (bottom).

Figure 47. Seesoft-like visualization in AspectJ.
LIST OF TABLES

Table 1. Concerns identified in jEdit. 21
Table 2. Source code organization in jEdit. 23
Table 3. Categories of concerns in jEdit by total size in lines of code. 29
Table 4. Comparison of concern identification methods. 68
Table 5. Concerns identified in ArchStudio and their corresponding related components. 160
Table 6. Results for concern identification in ArchStudio. 168
Table 7. Results for evaluation of concerns in ArchStudio. 169
Table 8. Identification of multiple concerns over components in ArchStudio. 170
Table 9. ArgoUML revision history. 176
Table 10. Number of fragments associated to the concerns at each milestone revision. 181
ACKNOWLEDGMENTS

This dissertation, like any other, is only the final step of a long process. I came to believe that this is a process of self-discovery, through years of ups and downs, failure and success, misery and exhilaration. I am forever grateful to all those who helped me, during these years, to discover my self.

First and foremost, I would like to thank my advisor, André van der Hoek for guiding me at every step throughout the Ph.D. You believed in me when I did not, and helped me when I needed it the most. Starting with my first class here at UCI, everything I learned about research has been because of you. Thank you for opening my eyes and teaching me how to take a step back and look at the bigger picture.

To Dr. Harold Ossher, thank you for introducing me to the world of software concerns. The discussions with you, and the work during the internship at IBM, helped shape the initial versions of what now is ArchEvol.

Many thanks to my other members of my committee, Prof. Susan Elliott Sim and Prof. Cristina Lopes, for their support, thoughtful questions and comments.

To the rest of the research group, it has been great having all of you around. Scott Hendrickson and Chris Jensen, thank you for your friendship, your help, and for all those long lunch discussions that kept me sane for these past years. Thanks to Justin Erenkrantz, Sukanya Ratanotayanon, Anita Sarma, Emily Navarro, Alex Baker, Nick Mangano, Chris van der Westhuizen, Ping Chen, Yuzo Kanomata, Eric Dashofy and John Georgas. Thanks to Ravi Chodavarapu for contributions to parts of ArchEvol’s implementation, and thanks to all the students that agreed to participate in my user studies.

I had the rare privilege of working at three of the greatest places in the world. Many thanks to my hosts, Ewen Denney and Bernd Fisher from NASA Ames Research Center, Harold Ossher, Stan Sutton and William Cheng from IBM Research, Robert Gardner, Alex Smolyanov and Mark Scheele from Google. The experience that I gained as a result of our work together is invaluable. Many thanks to the AudioAds IMS group at Google, who tolerated a more stressful version of myself during this last year.

To my parents and my sister, for driving me towards this road. Thank you for being a wonderful family, I could not have asked for anything more. My parents in law, thank you for your support, and for raising a wonderful daughter. Thanks to the Wagner family for your help and for making me dream of this. To my dear friends that I left behind coming here, Claudiu, Christian, Sandu and Catalin, there is such a thing as too much school. Many thanks to all the other friends who supported me during these years. To the Duffields, you should always go for the oats. Mariana and Bogdan, Vali and Doru, Beti and Radu, Dana and Mircea, thank you for being great friends.

Most of all, my deepest gratitude goes towards my wife. Some years ago, we took a leap of faith, and left our families and friends to start a new life centered around this Ph.D. Your courage and determination was remarkable, and your sacrifice was commendable. Thank you for your continuous support and encouragements, and most importantly for your unconditional love. I love you, I am grateful for everything we have done so far and I am looking forward to the rest of our lives together.
CURRICULUM VITAE

Eugen C. Nistor

EDUCATION:

2001-2009  Doctor of Philosophy in Information and Computer Science
            University of California, Irvine
            Area: Software Engineering

2001-2003  Master of Science
            University of California, Irvine
            Area: Software Engineering

1996-2000  Bachelor of Science
            Babes-Bolyai University, Cluj-Napoca, Romania
            Area: Information and Computer Science

EMPLOYMENT:

2007-present  Google Inc.
               Software engineer

2001-2007  University of California, Irvine
            Graduate student researcher
            Teaching assistant

06-09/2006  Google Inc.
            Summer Intern

06-09/2005  T.J. Watson IBM Research Center
            Summer Intern

06-09/2004  MCT/NASA Ames Research Center
            Summer Intern
TEACHING:

Instructor: ICS 52 Introduction to Software Engineering

Teaching assistant: ICS 121 Software Methods and Tools
ICS 125 Project in Software Design
Stats 120A Introduction to Probability and Statistics
ICS 53 Computer Networks
ABSTRACT OF THE DISSERTATION

Concern-Driven Software Evolution

By

Eugen C. Nistor

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2008

André van der Hoek, Chair

Concerns are central to software development. In order to understand and modify complex software systems, with requirements that are intertwined and have complex relationships, we need to be able to reason about different concerns in isolation from the rest. This is the essence of the principle of separation of concerns, used in designing any complex software system.

Today’s development environments do not adequately support managing the design of the system as it evolves. Concerns are invisible during development because environments focus on the programming language artifacts rather than on a conceptual model. However, most concerns are scattered across multiple source code elements, and tangled together with other concerns in each of these elements. This poses a great problem during evolution, as the ability to effectively reason about concerns deteriorates with time. Determining where concerns are implemented, and whether they are relevant in some part of the system, becomes increasingly difficult, if not impossible.

In this dissertation, we propose an approach that addresses this problem. We support a development process in which concerns are placed in a central role, driving all aspects of development. Our approach is embodied in a prototype implementation, ArchEvol, which enhances Eclipse with support for expressing concerns, linking these concerns to source code fragments, maintaining these links during development, and continuously presenting this data through visualizations. ArchEvol is unique in providing these features in an integrated, coherent development
We evaluate ArchEvol in three different case studies, which complement each other by evaluating different aspects of our approach. The first user study compares the ability of developers to effectively maintain a concern model by using ArchEvol or one of two other alternative solutions. A second study asks two expert developers to evaluate the usefulness of concern-related visualizations in a system that they helped develop. A third study imports, offline, the entire change history from an existing open-source project, and traces four concerns throughout the development life of the project. Combined, these three studies provide evidence of the potential of ArchEvol to effectively support a concern-driven process in which concerns become visible at all times.
1. Introduction

Software is inherently complex. One of the root causes of this complexity is the very nature of the problems that software addresses. A software system has to address requirements for functional features and needs to exhibit required qualities. Typically, these requirements are intertwined, having complex relationships, depending on one another, overlapping each other, or sometimes even being in conflict with each other. Even a seemingly simple problem can reveal itself to be more complex than initially thought, sometimes so complex that a single individual cannot comprehend it in its entirety.

During the process of transitioning these complex requirements into a software system that addresses them, developers are faced with various choices. A first step in a typical development scenario is designing the main architecture and assigning responsibilities to its components. Then, implementation languages and mechanisms are chosen, and the desired functionality is implemented in a software program as a set of source code files and associated artifacts. Each of the choices involved in this process has the potential of leading to a different software system, one of the multiple possible solutions that could achieve the same goals.

The life of a software system does not end with the first implementation. Systems evolve, and need to be adapted to incorporate new functionality. These types of changes are similarly difficult, as developers need to identify first where in the system the existing features are implemented, and then decide how to weave in the new functionality. With every change, developers have to take a decision about the way the new features are implemented.

The effects of taking wrong choices during development are typically not immediately seen. A software system can have a very good initial design, but, with time, this design may become brittle as more and more changes are made to it. The decay of the original design results in a structure of
the system that becomes so complex that changing any small part could result in widespread and unforeseen effects. For small and localized sections of the code, refactoring is a commonly used technique for changing the design while preserving functionality. At a higher level, however, refactoring is more likely to be completely avoided, as changing any large part of a system is excruciatingly difficult. Complex systems become, therefore, difficult to develop and, more importantly, difficult to evolve.

The ability to change gracefully is one of the most elusive software qualities, because software systems cannot be directly measured for how easy they are to change. Even if a system has a good design, it is virtually impossible to foresee all the possible ways in which the system can be changed in the future, or which of its features are more likely to change. A better way to address change is to manage its complexity, and make it easier for developers to determine how this complexity is modified over time.

A key insight supporting this dissertation is that today’s development environments lack adequate support for managing the design of a software system as it evolves. If developers could be made aware of the implications of their change over the system, they might be able to take the appropriate actions to correct the drift of the system’s implementation from its early conceptual design, instead of waiting for the time when the undesired consequences of their changes actually become apparent. By being continuously aware of the evolution of their system, developers can better manage the complexity of their system.

Before proposing a solution to this shortcoming in software development support, we need to understand first what causes this disconnect between the conceptual design of a system and its implementation. In order to do so, we need to take a step back and look at the fundamental principles that form the basis of any software development activity.

A fundamental way in which software complexity is typically addressed is to follow the principle of separation of concerns. This principle, introduced by Dijkstra more than 30 years ago [Dijkstra 1976, Dijkstra 1979, Dijkstra 1982], states that developers should be able to reason about each individual
concept of interest, or concern, in isolation from the rest. By focusing on only one concern at a time, a developer can understand better how this concern has to be implemented, and can check whether the implementation adheres to its desired behavior.

Concerns are important because we reason about a system in terms of concerns. The term “concern” itself suggests that they are concepts in which developers are interested. When an existing feature is changed in a system, that feature is a concern relevant at that moment in time. Developers need to be able to quickly identify where that feature is implemented in the system so that they can plan the change accordingly. Similarly, when a new feature is added, developers would need to determine where other features with which this new one may need to interact, are implemented.

Looking at software development practices today, it is striking to observe that, although concerns are critical in developing and evolving a software system, they are effectively invisible when doing so. The implementation of a software system is written using a programming language, which is typically a high-level general language that has no support for modeling concerns. Instead, developers can use existing facilities in programming languages such as element names, comments, and annotations to try to capture the concerns that a piece of software addresses. These methods of documenting concerns have, however, limited success, because of the lack of formal constraints for the use of concerns, because they depend heavily on the developers following very strict naming or commenting rules, and because there is no support for making good use of this concern information in the development environments.

Even more, typical development environments support the programming language instead of supporting the developer. Support for identification and management of concerns is limited, as developers have to rely on basic textual search tools to search the source code for relevant elements that can indicate a concern. Neither the language nor a typical development environment provide adequate support for a representation of a concern, or are able to give reasonable help to developers to determine where concerns are implemented in the system. In the end, the burden of
managing concerns and mapping them to their corresponding code is left as a responsibility of developers themselves.

Modularity has long been proposed as a solution to encapsulate the implementation of one or more concerns, and limit the exposure of these concerns to the rest of the system. However, modularity is still difficult. The domain decomposition of non-trivial problems does not lend itself to cleanly separated modules. Instead, the concerns in these domains have complex relationships that make clean encapsulation difficult or even impossible. Whenever some concerns are being encapsulated in their own module, other concerns will become scattered and tangled over these modules.

This dissertation proposes a different type of approach. The main philosophical difference is the fact that, in our approach, we set the goal of supporting any type of concern without imposing any constraints on their implementation. We want to leave the design and implementation modularity as choices that developers make, and do not try to impose any constraints on them. Instead, we want to be able to raise the awareness of how and where concerns are implemented in the system, in a way that would help developers in making better decisions on how to choose between these alternatives.

The goal of this dissertation is to offer a solution that addresses the basic problems that are related to concerns during development, as we discussed above. These fundamental problems are:

- Concerns are invisible in code, and developers are unaware of the concern implementation in code. As we have already discussed, concerns are not explicitly identified in code, neither by programming language constructs nor by software development environments. The process of determining where a concern is implemented is informal, inherently imprecise, and requires time and effort from developers.

- Current software development environments lack adequate support for managing concerns over time. As a consequence of concerns not being formally defined and mapped to code, the support for maintaining these mappings over time is not appropriate. Developers assume that a concern is going to be encapsulated in a module, but there is no mechanism
to ensure that this is actually true as software changes.

- Developers do not use concerns to effectively manage the evolution of the system. Software systems are not developed in one, single development phase. They evolve, with changes performed over subsequent versions of the same software system over time. Each of these changes can take two different directions: one that would maintain or improve the design quality of the system, or one that would deteriorate the design quality. Although concerns are important when designing the system, they are not considered at the time when these evolutionary changes are taken.

These three problems identify three levels of support required by a complete solution. Being able to manage concerns over time depends on the ability to make concerns visible and explicit during software development. At the same time, being able to take informed decisions during the evolution of the system depends on the ability to maintain precise mappings from concerns to code over time.

The approach proposed by this dissertation is a development method where concerns become central to software development and guide the entire evolution of a software system. The concerns are explicitly modeled in a concern model, and linked to the source code that provides their implementation. Developers use this concern model to explore the system based on concerns. During development, all changes to the source code are monitored in order to keep the concern model up-to-date. The mapping of concerns to source code is then aggregated and presented at the architectural level, in order to keep developers aware of where the concerns are implemented over the system, of whether they are scattered and tangled over the system’s components, and of how the implementation of concerns changes over time as the system evolves.

The novelty of our approach is three-fold. First, the representation of the source code fragments chosen for the concern model allows developers to identify concerns in code to any level of detail. This results in a concern model that is more precise and which can present a more accurate representation of how concerns are implemented in code. The second novelty is the use of this concern information to show a picture of the system at a higher level of detail. Our approach uses
the architectural view of the system to show high-level information about concerns, and metrics that
display concerns in implementation. The third novel aspect of our approach is continuously
maintaining this information over time as the system evolves. The architectural view is maintained
throughout development updated with concern information, and developers can become aware of
how the evolution of their system based on concerns. In this way, developers can act upon the
problems emerging in the system’s design before it is too late.

Our vision is implemented in ArchEvol, a development environment that supports modeling of
contents and visualization of concerns in code, that helps developers in maintaining the mappings
of concerns in code, and that supports the developer in becoming aware of the design quality of the
system at hand from the perspective of concerns. ArchEvol makes concerns visible by having the
concern model as an integral part of the development environment, and by providing color-coded
visualizations at both source code and architectural level.

This dissertation is organized as follows. We start, in Chapter 2, by presenting the motivation behind
our approach. We discuss the role of concerns in software development, their general properties,
and the relation between modularization and separation of concerns. In support of our argument, we
performed a case study that looks at the implementation of an existing software system (the jEdit
document editor [jEdit]), and identifies thirty two concerns of varied sources, mapping these
concerns to their source code implementation. This type of study was missing in related research
literature because it takes a broader look at concerns and their implementation. The lessons learned
from this study help us reassert our assumption that concerns have very varied implementations,
and help us in designing a solution that accommodates all types of concerns.

Based on the problems that we identify in current software development practices with relation to
support for concerns, we identify the relevant research questions that this dissertation addresses in
Chapter 3.

In Chapter 4, we present our approach and discuss the choices taken while designing this
approach. Our approach is based on three elements: a concern model that manages concerns and
their mappings to source code implementation, mechanisms for maintaining this concern model over time, and an architectural model that is used to reason about concerns at a higher level of abstraction. All these elements are continuously maintained over time, which results in concerns becoming central concepts that developers can use during the evolution of a software system. Using these elements, rich visualizations of concerns in both the source code and the architecture of a system raise the level of awareness that developers have about concerns during development.

The next chapter, Chapter 5, presents an implementation prototype of ArchEvol. This implementation is based on the Eclipse IDE platform [Eclipse Foundation 2004], as enhanced with support for concern based development.

An evaluation of ArchEvol is presented in Chapter 6. We have evaluated ArchEvol in three different case studies that are complementary to each other. The first one is a user study that looks at how developers use ArchEvol to identify concerns in code and maintain the concern model. In this study, eight subjects were asked to perform basic development tasks using ArchEvol. The subjects were able to perform their tasks better than without ArchEvol and comparatively to a similar tool, FEAT, while maintaining a more accurate concern model. The second study evaluates the second part of ArchEvol, the architectural-level metrics and visualizations. In this study, two expert developers were asked about concerns in a system that they implemented, ArchStudio4. The study proved that even expert developers are not able to accurately describe and estimate scattering and tangling levels of concerns over a system’s architecture, and that the visualizations that ArchEvol uses to show scattering and tangling of concerns are useful. The third study addresses the support for continuous maintenance and evolution of the system’s implementation and concern model. The purpose of this study was to simulate the use of ArchEvol in the development of a software system for a longer time. In order to do so, we looked at the evolution of the ArgoUML development environment, and tracked three concerns and their evolution over thousands of changes. The visualizations that ArchEvol provides prove that concerns are not static during development, and that their implementation changes as software evolves. Together, these three studies provide the evidence for the potential of our approach, and its implementation, to support management of concerns in real-
world projects.

We discuss related work in Chapter 6. Our approach touches a number of different research topics that include different approaches for modeling and coding a software system based on concerns, traceability of concerns to source code, support for concerns in software development environments, and the use and role of architectural models in software development. The contributions of ArchEvol that set it apart from the rest of these approaches are summarized in Chapter 7. We end by discussing future directions that the research presented in this dissertation enables in Chapter 8.
2. Background and Motivation

The previous chapter presented an overview of this dissertation and introduced, at a high level, the problems that this dissertation is addressing. In this chapter, we first present precise definitions of the key concepts underlying this dissertation. Then, we present a detailed analysis of the problems that current software development practices have in supporting concerns, both in initial development and throughout evolution. We end with a discussion of the “ideal” type of support that software development needs in order to address these problems.

2.1. Definitions

Concerns are fundamental to software development. The term “concern” was first mentioned by Dijkstra in the context of his theory of the “scientific thought” more than 30 years ago [Dijkstra 1976, Dijkstra 1979, Dijkstra 1982]. Dijkstra identified the principle of separation of concerns and the principle of the use of abstractions as two indispensable analysis mechanisms that software developers can use in order to address complex problems. The principle of separation of concerns states that one should be able to reason about a concern in isolation from any other concerns. The principle of the use of abstractions states that abstractions need to be used in order to provide simpler views of a software system. These two principles have survived the test of time and have become two of the main drivers of software design and analysis as we perform it today.

Although separation of concerns is a generally accepted principle in software development, the vast majority of the literature that discusses concerns does not present a precise definition of what a concern is. Instead, concerns are described intuitively with specific examples. Dijkstra discusses correctness and efficiency [Dijkstra 1982]. Concurrency and parallelism are discussed in [Lopes 1997]. Other types of concerns stem from implementation-specific mechanisms, such as logging,
debugging, and exception handling [Lippert and Lopes 2000], the implementation of which depends heavily on available programming language support.

Tarr et al. [Tarr et al. 1999] provide a more general description of concerns as “concepts of importance in a domain”. Examples of their concerns include objects of interest, functionality, and properties of the system. Besides the functionality of the system, both non-functional requirements and qualities that the software system might exhibit also are considered concerns in their view.

A different definition of concerns is based on how concerns are used rather than on what they represent. This type of definition implies that concerns arise once developers manifest interest in them. In this situation, a concern is defined by the software elements that are related to it. Sutton et al. describes a concern as “any matter of interest in a software system” [Sutton and Rouvellou 2002], a definition inspired from the, quite abstract dictionary definition of a concern as a “matter for consideration”. Sutton et al. also differentiate between logical concerns that represent conceptual “matters of interest” and physical concerns that represent elements in a software system.

A different definition of concerns is introduced by Robillard as “anything a developer might want to consider as a conceptual unit in a program” [Robillard 2003]. The concern model proposed by Robillard represents concerns as a tuple of fragments, where fragments are “descriptions of relationships between two sets of elements in a program model”.

In this dissertation, we adopt the following definition of concerns:

> A concern is a concept that relates a set of different software module fragments.¹

A concern is a general concept that can denote features of the system, non-functional requirements, or implementation-specific issues. Some such concepts can be very specific (such as a specific algorithm) or can be very abstract (such as reliability), but the one property that qualifies them as concerns is the ability to be linked to software. It is this relationship to software fragments that makes developers concerned about a concept. In this dissertation, we refer to this relationship as

¹ It should be noted that our definition of a software fragment is different from Robillard’s definition, as the remainder of the section will show.
the concerns being “associated” with the fragments, or the fragments “addressing” the concern.

We use the term fragment in this definition to emphasize the fact that the software elements that address a concern do not need to be entire modules. These software fragments can have different forms, depending on the language used to implement the software and its definition of a module. For example, in the context of an implementation written in the Java programming language, a concern could be addressed by an entire Java class, or could be addressed only by specific instructions of the Java class. The same concern could be addressed by a similar implementation in the C language as a set of files, or as parts in a set of files.

A key insight underlying our definition is the fact that software fragments are related to each other through a concern. The concern, then, essentially provides a rationale for the existence of these specific fragments. For example, all the source code fragments that are writing program execution traces to logs are related together in that they collectively participate in implementing the “logging” concern. The reason for why these fragments exist is the fact that the implementation addresses the “logging” concern. Removing the concern from the system will amount to removing all the source code fragments from the system’s implementation.

The ability to relate software fragments by the concern that they address is useful in program comprehension and maintenance. By providing the rationale behind software fragments that might otherwise seem unrelated, concerns help developers in understanding the implementation of a system in a way that is not supported by just the existing source code. When the implementation of a concern needs to change, then all the software fragments that are related by that concern need to be inspected by developers in order to guarantee that the change is complete.

In this dissertation, we extend the use of concerns to more than just program comprehension and maintenance, and move towards viewing concerns as elements of software design. Concerns exist in a system even if the users are not initially interested in changing the software fragments related to them. The size and shape of the various software fragments related to a concern can provide valuable information for judging the design of the system. A concern that represents an important
concept in a system will typically be addressed by a considerable number of possibly quite large software fragments. In contrast, a concern that is not important will be typically implemented by a few fragments of smaller size. In order to better understand a design in terms of concerns, however, these details need to be abstracted at an appropriate level of abstraction.

The principle of abstraction is the second important principle in software engineering mentioned by Dijkstra [Dijkstra 1976, Dijkstra 1979, Dijkstra 1982]. The main design mechanism through which abstraction is achieved is modularization, which is the activity of determining the breakdown of a software system into smaller, more manageable parts called modules. A seminal paper by Parnas links modularization to the principle of separation of concerns [Parnas 1972]. Parnas describes the activity of modularizing a system as a “responsibility assignment”, driven by the identification of important design decisions. System-level design decisions will yield the high level modules and the decomposition of the system, while other design decisions are encapsulated inside different modules. Each module will hide its implementation details from the rest of the system, with which it will interact through well-defined interfaces. In this way, if the implementation of a module needs to change in the future, the change will be localized to the inside of the module only.

Although a module was initially used to describe a particular type of implementation element, in the context of modular programming, we use a more general definition of what a module is. The definition that we use in this dissertation is:

\[
A \text{ module is an arbitrary element of decomposition of software, determined by a particular formalism used to describe the software.}
\]

To put this definition in context, consider the two levels of abstraction for the modularization of a system we use in this dissertation: source code and architecture. They both describe the same system, but their purpose is different. The implementation is a precise, detailed description of software, the closest representation of software to a machine executable format that a developer produces. The architecture, on the other hand, is an abstract, high-level view of the system.

Both the code and the architecture are modularizations of the same system. Both describe a
breakdown (or even multiple breakdowns) of the same software system, but using different formalisms. In each decomposition, modules are assumed to be distinct, and the totality of the modules cover the entire software system. The main difference lies in the definition of a module that source code and architecture use.

Source code consists, in general, of a set of files that contain text written in a programming language. There are, therefore, two types of modularizations that the source code organization exhibits, one induced by the textual files and a logical one induced by the programming language. In one, modules are folders and files maintained by a file-system; in the other, modules are (for example in an object-oriented language) packages, classes, or methods. The programming language modularization is based on the textual representation, to which it adds constraints for what constitutes a module, constraints enforced by a syntax parser. For example, the implementation of a method block in Java is confined by the block markers “{“ and “}”. In this dissertation, we use the more general file-based representation of the implementation.

A software fragment, in the context of source code implementation, denotes a contiguous part of a source code file. An individual software fragment cannot span multiple files, and has to have at least the length of one text character. However, there are no semantics attached to a software fragment. The software fragment does not have to contain meaningful text, or to only contain full programming elements. Any interpretation of the text is left as the responsibility of tools that use these fragments.

Software architecture is a design representation of a software system at a higher level than the implementation. Perry and Wolf [Perry and Wolf 1992] define architecture as a description of a system that is comprised of its elements, its form, and its rationale. Shaw and Garlan [Shaw and Garlan 1996] define it as a description of a system in terms of computational components and interactions among these components. Both these definitions agree on the canonical elements of an architecture, which are its components, and on the form of the architecture, which is created by connecting components together. The definition that we adopt in this dissertation is:
The architecture of a software system is an abstraction over its implementation that focuses on high-level components and their interconnections.

We loosely agree with preexisting definitions, but we highlight the need for the architecture to reflect the underlying implementation. The purpose of the architecture in our work is thus to “hide” a variety of the implementation details in order to express information about the organization of the software system as a whole.

The architecture of a system is a form of modularization of the system, albeit at a high level of abstraction, where the modules are defined as components. The functionality of the system is modularized by being split apart into large components that each encapsulates a specific behavior or related set of behaviors.

Returning to the topic of modularization and its purpose, we note that, although modularization is driven by the ultimate goal of encapsulating each concern in a single module, in practice this is not attainable. The main cause is that a system cannot be fully decomposed along every piece of functionality without having crosscutting functionality between modules. Some concerns in the system might be fully encapsulated in their own module, but other concerns will inherently crosscut a number of different modules. At the same time, while modules may be intended to address one primary concern, they will inherently address other concerns as well. There is, therefore, a mismatch between the modularization of the system as attained in practice and the logical separation of concerns, problem known in literature as “the problem of dominant decomposition” [Tarr et al. 1999].

Two important phenomena that describe the disconnect between the logical decomposition of a system, based on concerns, and the physical decomposition, imposed by the modules, have been labeled in the literature as scattering and tangling of concerns [Tarr et al. 1999]. Scattering denotes the fragmentation of a concern’s implementation over the modules, while tangling denotes the level of interactions between different concerns in each module.
A concern is scattered if its implementation consists of a number of software fragments that belong to different implementation modules.

A concern is tangled if any part of its constituting software fragments also belong to other concerns.

Concerns can have various degrees of scattering and tangling. The more modules address a given concern, the more scattered it becomes, and the more concerns are addressed by a module, the more tangled these concerns become. A concern will have no scattering and no tangling when all of its related fragments belong to a single module and this module does not address any other concern. This, generally, is impossible to achieve because, as we have shown, concerns will intersect each other and therefore will overlap over the same software fragments.

One could then ask for “good” or “bad” or “minimally acceptable” levels of scattering and tangling. But these naturally vary per system and decomposition approach. Less scattering and tangling is desirable, but there are no generally acceptable absolute values by which to decide whether or not certain levels of scattering and tangling are acceptable for any given system.

One of the goals of software engineering research is to devise methods and tools that lead to the development of better software. One way of evaluating if software is “good” is to test its ability to change gracefully. This is an elusive quality, since it is not always visible at the early stages of development. Rather, this property reveals itself during software evolution.

For a large number of software systems, continuous evolution is a must [Lehman 1980, Lehman and Ramil 2001]. The type of applications that model real-world systems are bound to continuously evolve in order to keep up with changes in requirements. Without continuously taking active measures to manage the increasing complexity of a system, it is known that this evolution becomes increasingly difficult [Parnas 1994]. Our definition of evolution is:

**Software evolution is a continuous process in which the software system is modified to improve the existing implementation of its functionality or is modified to adapt to new functionality requirements.**
In line with other views of evolution, we differentiate between two types of evolutionary changes: the ones that modify an existing implementation of a system without changing its functionality, and the ones that modify the existing functionality of the system. Modifications that preserve the functionality of the system are bug fixes, which correct the functionality to its expected behavior, or refactoring, which improve its design. Adaptation to new requirements includes removal of obsolete functionality as well as addition of new functionality.

During software evolution, the ability to identify code fragments related to the concerns that are being affected by the changes is crucial in ensuring the completeness of the changes. When functionality needs to be preserved, developers need to ensure that each fragment related to the concerns affected by the change is still correct. When functionality is removed, developers have to find and then remove all source code fragments that implement it. At the same time, new functionality is not added separate from the rest of the system, but instead needs to interact with functionality already implemented, for which developers have to find the code fragments.

The ability to support developers in performing evolutionary changes to a system is even more important if we consider that evolution is not just a phase of software development process, but covers instead the entire software development process. Historically, software development processes changed, over time, from the classical models of waterfall and spiral models [Boehm 1988, Royce 1970] where evolution was thought of as only a “maintenance” phase that took place after the system was developed, to new, popular development processes such as extreme programming [Beck 1999] and open source development [Raymond 2001], where the entire software development process is a series of evolutionary cycles, from the start to the end of development. In these new approaches, a software system “grows” incrementally, with new features being constantly added over time. The focus is on developing an initial, simple working system first, which is later evolved with new features, small cycle by small cycle. In this view of software evolution, developers have to be able to perform evolutionary changes, and identify concerns in code, at all times during development.
2.2. Motivating Example

In order to better exemplify how concerns are present in the implementation of a software system, and how the discrepancy between the conceptual separation of concerns and the implemented modularization manifests itself, we performed an analysis case study of an existing software system. Existing work that analyzes the implementation of concerns in an existing system has focused on using specific concerns, usually one or at best a few, which are then used as examples for a broader theory of how concerns are implemented. Examples include logging and exception handling [Filho et al. 2006], distributed programming [Lopes 1997], and other such general concerns. Case studies that discuss concerns in code are focused on evaluating tools and approaches, and do not represent the broad spectrum along which concerns are implemented in code [Dagenais et al. 2007, de Alwis et al. 2007, Robillard and Murphy 2002, Robillard et al. 2004]. Examples of concerns used in these studies include the syntax highlighting feature in jEdit [Dagenais et al. 2007], saving a document in jHotDraw [Robillard 2003], autosave and setting loading in jEdit [de Alwis et al. 2007], and again the autosave feature in jEdit [Robillard and Murphy 2002, Robillard et al. 2004]. There is no current study, to the best of our knowledge, that takes a comprehensive look at concerns in an existing system.

The study described in this section takes a broader approach to concern identification and implementation in an existing system. Two of the goals of the study are to look at a sizeable number of different concerns, rather than picking one or two specific examples, and to comprehensively link these concerns to source code fragments that are related to them. In this way, the study provides useful data about the nature of the implementation of concerns, as well as about the form, size, and number of the source code fragments that are related to each concern. The data resulting from this study allows us to take a more thorough look at the problems that software development faces today with respect to being able to encapsulate concerns in modules, and to draw conclusions about the levels of scattering and tangling of concerns in an existing system.

The project that the study used was the open-source document editor jEdit [jEdit]. The jEdit project is frequently used as the subject for case studies of software development in general [Girba et al.
2005, Kirk et al. 2007, Liu et al. 2007, Mariani and Pezzè 2007, Rysselberghe et al. 2006, Zimmerman et al. 2004]. It also has been the subject of a few studies about concerns, but only a small number of concerns has been evaluated in these studies [de Alwis et al. 2007, Robillard and Murphy 2002, Robillard et al. 2004]. Dagenais et al. looked at 16 concerns in four systems, but the identification process was limited to mapping only fields and methods related to each concern [Dagenais et al. 2007, de Alwis et al. 2007, Robillard and Murphy 2002, Robillard et al. 2004].

Besides having been used in other user studies, jEdit has other characteristics that make it suitable for our study: it has a medium-sized implementation, it is a mature application with features that cover both basic and advanced text editing, and, due to it being an open-source project, grew incrementally over time as more features were needed.

The study was performed in two major steps. In the first step, a number of concerns were identified, based on existing documentation from the jedit.org website. The second step consisted of manually inspecting the source code and assigning source code fragments from the text of the source code to the concerns that were related to them. Since the potential number of concerns in an application such as jEdit can be very large, identifying every conceivable concern and linking it to the source code is not a reasonable goal due to the time and effort such an effort would require. Instead, we focused our analysis on a smaller, but still significant subset of concerns that allows us to draw meaningful conclusions without sacrificing generality.

Three guiding principles were used to ensure some form of validity in the selection of concerns and the resulting generality of conclusions. These principles are:

- **Diversity**: use a diversified list of concerns, which includes different types of concerns at different conceptual levels;

- **Precision and completeness**: in determining which source code fragments are associated with each of these concerns, only include the most specific fragments and ensure that all such fragments are found; and

- **Code coverage**: Have, for each line of code, at least one concern that provides the rationale
These three principles help us manage the potentially very large scope of our study. By ensuring that we had a diverse list of concerns, we restricted the set of concerns to a manageable, but still representative set. The precision and completeness of the concern identification exercise is crucial in drawing meaningful conclusions about how concerns are implemented. The code coverage principle is helpful in verifying that both concern selection and identification in code are complete. In other words, it provided a “stopping criterion” for our study.

### 2.2.1. Selection of Concerns

The primary source for selecting concerns was the existing online documentation of jEdit\(^1\). The available documentation for the jEdit project consists primarily of a summary of features\(^2\) and of a detailed user guide\(^3\). These were the sources for an initial set of concerns.

To this initial set of concerns were added concerns that emerged during the study. While identifying concerns in code, there were cases when, for a given source code fragment, none of the concerns in the list were related to it. In this case, a new concern was selected from the names of the programming elements in the code (such as keywords in package or class names) and added to the list. The code coverage principle ensured that the addition of new concerns had a “stopping criterion”, as we ended adding new concerns when every line of code was accounted for by at least one concern. Whenever a new concern was added during the study, the entire code needed to be reexamined in order to find out all source code fragments for this concern, as required by the completeness principle.

A total of 32 concerns resulted, as shown in Table 1. Two important domain concepts that are related to the domain of jEdit are buffer\(^4\) and view. According to the user guide, the buffer is the internal representation of an open file, and the view is an editor window, which is a part of a larger

---

1 Information available on jEdit.org and sourceforge.net/projects/jedit as of December 2007.
2 http://www.jedit.org/index.php?page=features
4 The concern names are italicized throughout the description of jEdit’s features.
User Interface (UI) concern. Other visual elements are menus, which are used to invoke commands that manipulate the text in the buffer, an indicator strip named gutter that is located next to the view and shows properties for each line in the buffer, and the status bar which is located at the bottom of the view and shows general properties about the buffer. One of the properties of the buffer is its character encoding. Other properties are syntax-based highlighting of the text, and automatic folding, based either on indent or on markers, which are pointers to locations in text. Markers have their own separate view, associated commands for adding and removing them, and are saved to, and loaded from, the file system.

jEdit includes a number of features for text editing, of which common ones are copy and paste, and unlimited undo and redo. Search and replace is based on regular expressions. More specific editor customizations are specified as an edit mode. An edit mode contains syntax highlight rules, auto indent behavior, and other customizations that are associated with a file based on the file name extension used, such as auto-expansion of abbreviations into longer text. Editor actions can be recorded as macros and performed later.

Related to jEdit’s interaction with the file system, saving and loading are two major features. The location of the files on disk is found through a file system browser. Also related to file interaction, two special features, autosave and backup, are used to recover from lost data in the case of a crash. The autosave feature periodically saves a copy of the buffer to disk, while the backup feature stores incremental versions of each saved buffer.

One of the main features of jEdit is its extensibility through plug-ins. The plug-ins infrastructure in the code makes possible this extensibility. Three important properties of the buffer are shown in the status bar: the edit mode, the folding mode and the character encoding. Many of jEdit’s properties can be customized through options. The implementation of the options feature includes: (1) a UI component containing the panels that show the values of the options and let the user modify them, (2) storage of the values to a local file, and (3) the use of these values throughout the implementation. The features of jEdit are explained to the users through a help system.
Table 1. Concerns identified in jEdit.

<table>
<thead>
<tr>
<th>Concern</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>internal representation of a document</td>
<td>documentation</td>
</tr>
<tr>
<td>view</td>
<td>the user interface representation of a document</td>
<td>documentation</td>
</tr>
<tr>
<td>UI</td>
<td>user interface, all elements with a visual representation</td>
<td>code</td>
</tr>
<tr>
<td>menu</td>
<td>a window menu showing a list of actions</td>
<td>documentation</td>
</tr>
<tr>
<td>gutter</td>
<td>a visual element that shows properties for lines in the buffer</td>
<td>documentation</td>
</tr>
<tr>
<td>status bar</td>
<td>a visual element that shows general properties of the buffer</td>
<td>documentation</td>
</tr>
<tr>
<td>encoding</td>
<td>the character encoding used in a document</td>
<td>documentation</td>
</tr>
<tr>
<td>highlight</td>
<td>a visual indication of specific text fragments with colors</td>
<td>documentation</td>
</tr>
<tr>
<td>indent</td>
<td>ability to indent text based on syntax</td>
<td>documentation</td>
</tr>
<tr>
<td>folding</td>
<td>capability of collapsing indented blocks of text in a single line</td>
<td>documentation</td>
</tr>
<tr>
<td>markers</td>
<td>anchors that the users can place in the text of the document</td>
<td>documentation</td>
</tr>
<tr>
<td>save</td>
<td>the functionality of saving the content of a buffer to a file</td>
<td>documentation</td>
</tr>
<tr>
<td>load</td>
<td>the functionality of loading the content of a buffer form a file</td>
<td>documentation</td>
</tr>
<tr>
<td>file system</td>
<td>interaction with the underlying file system</td>
<td>code</td>
</tr>
<tr>
<td>copy/paste</td>
<td>functionality of moving text between the buffer and the clipboard</td>
<td>documentation</td>
</tr>
<tr>
<td>undo/redo</td>
<td>ability to remove or add back the last change to the buffer</td>
<td>code</td>
</tr>
<tr>
<td>regexp</td>
<td>the use of regular expressions in searches</td>
<td>code</td>
</tr>
<tr>
<td>edit mode</td>
<td>a specific set of shortcuts used by jEdit to edit certain files</td>
<td>documentation</td>
</tr>
<tr>
<td>macros</td>
<td>ability to record and play a succession of commands</td>
<td>documentation</td>
</tr>
<tr>
<td>search</td>
<td>ability to search the text of the buffer</td>
<td>documentation</td>
</tr>
<tr>
<td>abbreviations</td>
<td>ability to replace words in the text with their shorter forms</td>
<td>code</td>
</tr>
<tr>
<td>browser</td>
<td>a user interface element that lets the user navigate the file system</td>
<td>documentation</td>
</tr>
<tr>
<td>autosave</td>
<td>automatic save of the buffer at fixed time intervals</td>
<td>documentation</td>
</tr>
<tr>
<td>backup</td>
<td>save of multiple older copies of the buffer</td>
<td>documentation</td>
</tr>
<tr>
<td>plug-ins</td>
<td>an extension mechanism that jEdit uses to integrate new features</td>
<td>documentation</td>
</tr>
<tr>
<td>options</td>
<td>parameters that jEdit allows the users to modify</td>
<td>documentation</td>
</tr>
<tr>
<td>help</td>
<td>ability to provide the user with assistance and information</td>
<td>documentation</td>
</tr>
<tr>
<td>logging</td>
<td>the use of Java logging mechanisms</td>
<td>code</td>
</tr>
<tr>
<td>exception handling</td>
<td>the use of Java exception handling mechanisms</td>
<td>code</td>
</tr>
<tr>
<td>multithreading</td>
<td>the use of multithreading Java mechanisms</td>
<td>code</td>
</tr>
<tr>
<td>UI helper</td>
<td>functionality related to UI but not having a visual representation</td>
<td>code</td>
</tr>
<tr>
<td>application</td>
<td>functionality related to instantiating JEdit</td>
<td>code</td>
</tr>
</tbody>
</table>
To this list of concerns, which are specific to jEdit, were added a number of concerns that are
generic and applicable to any software system. Two important concerns are logging and exception
handling, which are used extensively in related literature as examples of concerns [Baniassad and
Two other general concerns added were multithreading, which is related to the use of programming
threads to parallelize certain operations, and application, which is related to the overall instantiation
of the system and coordination of its main components.

The concerns selected follow the diversification principle, with concerns that belong to different
categories based on their type, granularity, generality, and source. Some of the concerns, such as
buffer, markup, and abbreviations are domain objects that are responsible for storing data; other
contents, such as save, undo, highlight and messaging, are functional features that manipulate and
organize data objects. Concerns such as UI are general, high-level concerns that are likely to impact
large segments of the source code. By contrast, gutter, menu, and status bar are all finer grained
constraints that subclass the more general UI concern. Search, formatting, and undo are common
concerns that any text editor application should address. Buffer, browser, and marker are specific to
jEdit’s design and are the result of choices made during its development.

2.2.2. Identification of concerns in code

The second step of the study was to associate existing source code fragments with the concerns
that provide the rationale for their existence. The source code for jEdit version 2.4 consists of 305
classes organized in 18 packages\(^1\). Table 2 shows a breakdown of the source code files and the
total number of lines of code for each of the packages in jEdit, calculated with the “wc” Unix
command line utility\(^2\). jEdit’s source code project contains more source code files, but only the code
in the “org.gjt.sp.jedit” project and its children contain the implementation of jEdit; the other

\(^1\) The source code was checked out from https://jedit.svn.sourceforge.net/svnroot/jedit/jEdit/trunk at
revision 10440.

\(^2\) The command used was: find *.java -exec cat -- {} \; | wc -l
packages are general utilities that are used by jEdit. As can be seen from Table 2, more than half the implementation of the entire project is found in only three packages: jedit, jedit.gui and jedit.textarea. By contrast, eight out of the 18 packages have less than 2500 lines of code.

Fragments belonging to concerns were manually identified in the code, which was a tedious and not always straightforward process. Each of the files from jEdit’s implementation had to be carefully inspected at least once, in order to determine which fragments are related to which components. In a few cases, new concerns were identified during this process, as for some fragments of code none of the existing concerns fit as providing a rationale for their existence. When this was the case, files that were already processed had to be re-inspected in order to re-examine them with respect to new concerns. These concerns are labeled with “code” in column 3 of Table 1.

During the inspection of the text of the source code files, keywords from names of packages,
classes, methods or variables were used to suggest which concerns should be associated with a text fragment. For example, the names of the files Buffer.java and BufferHistory.java suggest that the entire file content is related to the buffer concern. Similarly, method names also suggest keywords that can be used. For instance, the save method in Buffer.java is related to the save concern.

However, our source code analysis aimed to be both precise and complete, as described in the previous section. Therefore, the concern identification process was not limited to only classes and methods, but looked at finer granularity of individual source code statements, narrowing down the minimal textual fragment that can be associated with each concern.

When examining individual statements, a more nuanced decision process was used in determining which parts of the code were included as belonging to a concern. The general principle followed was to limit the textual fragment to the smallest size that is still related to the concern, but which was still forming contiguous text intervals. Each software statement is evaluated individually, but when two or more fragments were physically next to each other in the text, then a large fragment that encompasses them was assigned in a single piece.

As with method names, the names of the variables would suggest, in general, the concern to which these variables belong. For a variable, its declaration statement and all of its uses throughout the code are the fragments that are linked to its associated concern. For instance, the parameters in the declaration of a method would be individually marked as a separate fragment. In the top code fragment in Figure 1, the “recover Autosave” method is related to the autosave concern, and as a result the entire code of the method is assigned to that concern, but the “final View view” parameter is related to the view concern, and only the parameter declaration and the uses of the “view” parameter in the method’s code are assigned to the view concern.

For simple statements, the variables used would suggest the concerns that the entire statements should be associated with. For example, the second code snippet in Figure 1 shows a comment and three statements, all associated with the markers concern. For control statements, the variables in the control structure would indicate the concern to which the entire group of statements in the
statement’s block would be assigned. However, the content of the statements still have to be inspected. For example, the third snippet in Figure 1 shows an “if-else” statement whose main condition is a variable relating to the *folding* concern. In fact, both the block under the “if” clause and the block under the “else” clause are related to the *folding* concern, and as a result the entire fragment was associated with the *folding* concern. The fourth snippet in Figure 1 shows, by contrast, an “if-else” statement where the “if” block is related to the *regular expressions* concern, but the “else” block is not.

This entire process required a careful inspection of the code before deciding which fragments to associate with a concern. For the most part, the decision of whether to assign a source code fragment to a concern was based on the names of the classes, methods and variables. As a result, keywords in these names would suggest one or more concerns to which the source code fragment was assigned. However, this was not a process that could be easily automated because of a number of irregularities in the patterns of how these keywords would be present in the implementation, and because of various cases of collision, where the same keyword would be used in two different contexts to suggest two different concerns.

For example, the keyword “marker” is associated with both text markers, which we associate with the *markers* concern, and with token markers, which are syntax elements associated with the *syntax* concern. Similarly, searching for the keyword “mode” would lead to results that are associated not just with the *edit mode* concern or with the *folding* concern, but also with the word “model”, and so on. The “filter” keyword is used to indicate a file filter in jEdit’s file browser, but it is also used in names of classes such as REFilterReader and REFilterInputStream to indicate filtered streams, or in ClassNodeFilter to indicate patterns in text, both of which are unrelated to the *browser* concern. The keyword “try” used in exception handling blocks was also used in variable names and comments, or as part of the “Entry” name.

One of the factors that eased the concern identification process was the use of markers in jEdit’s source code. Because jEdit was used in its own development, jEdit markers are used pervasively in
its code to group together blocks of source code fragments that are logically related. For example, we found instances where markers would enclose a number of methods between “//{{{ Edit modes, syntax highlighting” and “//}}”, suggesting that these methods are related to the edit mode and highlighting concerns.

2.2.3. Limitations

jEdit is an open-source project, and as such its development was driven by individual developers’ visions rather than by formal requirements. Therefore, it is expected that most of its features (and as a result some of its main concerns) were added over time, as they were needed. We believe that this scenario is typical for a number of software development projects that evolve their features rather than having them known completely in advance. For these types of projects, we believe jEdit serves
as a representative sample.

A limitation coming from the way this case study was performed is the fact that the concerns are the ones that we determined to be representative of the system. This subjective selection of concerns might result in a different set of concerns than the ones that developers might consider important. We addressed this issue by selecting concerns that are indicated in the maintained, existing documentation, as well as existing dominant keywords in the implementation. While these concerns tend to be about higher-level concepts, they are also much more likely to be among the concerns in which developers are interested.

2.3. Concerns in Software Development

The definitions of concerns and their scattering and tangling that were discussed in Section 2.1, together with the data provided by our jEdit case study, set the stage for a discussion of concerns in software development. Below, we present our analysis of the ways in which current software development practices lack comprehensive support for concern identification and management.

We first present a set of properties that characterize concerns, and which distinguish concerns from other concepts in software development. Next, we discuss how current software development practices try to achieve separation of concerns through modularization, and explain the causes as to why this approach falls short. We then discuss the effects that the lack of identification of concerns has upon the evolution of a software system. Then, we introduce the concept of software architecture, which will be discussed here from a conventional perspective of software development, but which will become a key element in our approach, later in this dissertation. These discussions are supported with examples from the jEdit case study.

2.3.1. Properties of Concerns

A critical property of concerns is their generality, as a concern can denote different things. In our jEdit study we observed concerns that represent functional features of a system, that range from
functionality that is general to any system (multithreading, logging, application), to general user interface features (view, status bar, menu) to features that any text editor has (save, load, view), to features that are specific to jEdit (abbreviations, highlight, syntax). Other types of concerns, such as system qualities (non-functional requirements) could also potentially be concerns, although they would be more difficult to trace to source code elements. This general nature of concerns makes designing the modularization of the system based on these concerns difficult as it becomes challenging to simultaneously optimize a broad variety of different types of concerns, some of which might not be even know until later.

One of the consequences of the generality of concerns is their level of granularity as measured in the implementation. The source code implementation that is related to a concern can be as large as one or more packages and span one or more entire files. Alternatively, it can be as small as a few lines of code inside a specific file. In general, there is no implementation element that would generally fit such a broad range of needs. We cannot force every concern to be implemented in one file, for instance. Instead, then, a developer must select from the available choices what elements may be best used in encapsulating each different concern.

A quantitative analysis by the total number of lines of code that were associated with each concern in jEdit is presented in Table 3. The value for the lines of code corresponding to each concern varies from a few hundreds lines of code to almost half the total lines of code in the implementation of the entire jEdit project. Intuitively, the ordering of concerns implied by clustering LOC values, as shown in Table 3, implies an ordering of the prevalence of concerns in the implementation. Some of these results can be intuitively derived from the generality of the concerns: we expect that the implementation of autosave and backup to be smaller than that of load and save, and even smaller than the implementation of the UI concern.

However, while important, little useful can be generically said about a large concern, and large concerns need to be broken down into smaller ones to be able to effectively reason about them. Reasoning about the smaller concerns that are part of the UI concern, such as highlight, menu,
browser, and determining a similar ordering intuitively, before the concerns are actually identified in code is difficult if not impossible. The same is true when comparing different types of concerns, such as general concerns (logging and exception handling), data concerns (buffer) and feature concerns (folding). Thus, the data provided by the identification of concerns in code is essential in accurately determining the prevalence of the implementation of individual concerns in the overall system implementation.

A more detailed analysis of the source code fragments that were associated with each concern is shown in Figure 2. The average number of lines of code per fragment, and the standard deviation, are plotted with continuous lines. The values for these two measures are shown on the y-axis on the left of the graph. Superimposed over these values is the number of source code fragments for each concern, drawn with a dotted line, and whose values use the right-side y-axis.

A first observation that can be drawn from this graph is the following:

The shape, size and form of the source code fragments for the majority of concerns are extremely varied.

This graph shows a high variation in the size of source code fragments by concern. The average
length starts from a few lines of code for exception handling and logging to about 200 contiguous lines of code for the UI concern (if we exclude the application concern, which is an outlier compared to the rest of the values). The standard deviation follows a similar trend, but reaches 450 lines of code for the UI concern. It can also be observed that there is no direct correlation between the number of fragments, shown with the dashed line, and the size of the fragments, shown with the two full lines. Notable examples are exception handling and logging, which have a large number of very small fragments, buffer, which has a large number of fragments of medium size, and UI, which has a small number of very large contiguous fragments.

With respect to previous work, we can observe that, although exception handling and logging are used as two of the most frequent examples of concerns, they are not representative for all types of concerns that might be identified in code. Their implementation is typically a large number of fragments that have only a few lines of code, which is not representative of the concerns we found in the jEdit study.

A second important property of concerns, stemming from their generality, is the fact that their
implementations inevitably overlap. One of the reasons for overlapping is the hierarchical nature of concerns: the implementation of a concern will automatically overlap all of its sub-concerns’ implementation fragments. For example, all the implementation of the view concern also belongs to the UI concern, because the view in jEdit is a user interface element.

Another type of overlapping is related to the inherent complex structure of concerns. The same source code fragment can belong to two or more concerns. The implementation of the functionality that saves the buffer to a file in jEdit belongs to both the buffer and the save concerns, and at the same time, these two concerns are not in a hierarchical relationship (the buffer concern has more implementation than just saving the buffer, and the save concern deals with saving other data as well, such as options and recovery information.) Another example is the autosave functionality which is related to, but not contained by, the save functionality. The autosave concern is a separate concern from the save concern, but both are intricately linked together in the implementation.

Figure 3 shows an analysis of the overlapping of concerns in the jEdit case study, on a line of code and on a file basis. The first data series, shown in the left columns on each pair, represents the
number of lines of code that are linked to one or more concerns. The second data series, shown in the columns on the right of each pair (with the corresponding values shown on the scale on the right side), shows the number of files that have one or more concerns. The x-axis represents the number of concerns that overlap.

We can see that only about half the total lines of code assigned to concerns are actually linked to a single concern. The other half has two, three, or even four concerns that overlap. At the file level, only a small percentage of the total files (about a quarter) contain a single concern. The other files contain more than one concern, with a significant number of files having more than four concerns associated in them.

These numbers result all from only 32 concerns, which represent a conservative number of concerns. In a real system, where the number of concerns can potentially reach a much larger value, the overlapping of concerns can only increase. We can therefore draw the following observations about the overlapping of concerns:

\[
\text{A significant percentage of concerns overlap over the same source code files and lines of code.}
\]

### 2.3.2. Separation of Concerns and Modularization

A seminal paper revealing the mismatch between modularization and separation of concerns is the paper describing Multi-Dimensional Separation of Concerns (MDSOC) [Tarr et al. 1999]. The problem, known as the “tyranny of dominant decomposition”, appears when the structure of concerns, which are conceptually intertwined, is forced into a “flat” modularization structure. As a result, the decomposition imposed by the modularity makes some concerns more relevant than others by encapsulating their implementations in modules. While some concerns are then indeed isolated, other concerns will crosscut one or more modules in the system.

A key observation underneath this dissertation is that modularity is different from separation of concerns. While separation of concerns is conceptual and therefore stable at a given moment in time, the modules are an implementation artifact and therefore variable. Modularity is a choice made
during software design. Choosing a different modularization will have different effects on the quality of the system, one of which is the ability to be easily changed in the future.

Take, for example, the three examples of possible modularizations for a hypothetical software system, shown in Figure 4. At one extreme, shown in the figure on the left side, a system could be implemented as a single, very large module. The other extreme, shown on the right side, would be a modularization where each concern is separated and encapsulated in its own module. For example, in an object-oriented programming implementation, we can consider source code classes as being the modules. The extreme on the left means that the entire system is implemented in only one class, while the example on the right means that the system is implemented using classes with very little code, potentially containing one method with one line of code in them.

Of course, these two hypothetical extremes would yield modularizations that are clearly not desirable, if not even impossible to attain. Except for extremely simple applications, the choice of writing the entire source code of a software system inside a single class is not a choice that would realistically be considered. Writing each concern in its own individual class would, on the other hand, lead to an implementation that would have a very large number of classes. If a concern would be implemented in only a few lines of code, then following this principle would require those lines of code to be implemented in their own class. And, as we have seen in the previous section, completely separating two or more overlapping concerns that share the same code might even not...
be possible, in which case the best separation of concerns will be achieved if the overlapping parts will be extracted and written in their own classes. This would lead to an extremely large number of files, each with very limited functionality. While such an implementation might be possible, in practice it will not be realistically achieved, because of the extremely large number of classes and relationships between them that have to be comprehended and managed by developers.

However, in between these two hypothetical extremes lie a number of possible, more reasonable alternatives. In Figure 4, one such modularization is shown in the middle. The key observation is that a selected modularization of a software system is just one instance from a space of different possible, viable design alternatives that can be considered during development. In all these hypothetical modularization alternatives, the same concerns would be implemented, and the system will still implement the same requirements. The difference between these possible alternatives is the way in which the concerns are assigned to each of the modules and, consequently, the scattering and tangling that necessarily results.

Since all of the modularization choices that developers may choose for a system exist in between the two hypothetical extremes of Figure 4, scattering and tangling of concerns over the modularization cannot be completely eliminated. Due to the intricate, crosscutting relationships between concerns, minimizing scattering and tangling for one concern might mean that other concerns to which this concern relates become more scattered and tangled.

Although minimizing scattering and tangling is one of the goals underlying a good design, the degree of scattering and tangling of a concern might change over time, and this change might take place either intentionally or unintentionally. Some concerns will naturally crosscut a number of other concerns, and therefore their degree of scattering and tangling is likely to change when any of these other concerns is changed. Other concerns might start being cleanly encapsulated, but might become crosscutting because developers spend little effort on reevaluating the scattering and tangling of concerns, each time a change is added to the system.

We can use examples from the implementation of the jEdit project to illustrate these properties. One
such example is the implementation of the buffer concern, which takes a number of different forms, and which can be traced from the coarser granularity of packages and classes to the finer level of lines of code. A quick look at the implementation reveals 39 classes that are related to the buffer concern, and these classes are spread over 11 packages. In some packages, such as the org.gjt.sp.jedit.buffer.* and org.gjt.sp.jedit.bufferio.* packages, all the classes are related to the buffer concern. In other packages, only a few classes are related to it.

Most of the implementation of the core functionality related to the buffer is implemented in two main classes, org.gjt.sp.jedit.Buffer and org.gjt.sp.jedit.buffer.JEditBuffer. Taking a closer look at the implementation of the org.gjt.sp.jedit.Buffer class, the implementation of this class is closely intertwined with the implementation of other concerns. From the 76 methods in the Buffer class, 37 are accessor methods. The rest are related to one or more features related to the buffer. For example, the load(...) and save(...) methods are related to the serialization of the buffer. At the same time, the serialization concern is not implemented in only this method. Almost 30 other classes have classes or methods related to saving or loading, not only of the buffer, but also of backup copies, abbreviations, options, user properties, and other type of data managed by jEdit.

If we look in more detail at the implementation of the load(...) and save(...) methods, we observe that even these methods contain code that is intermingled with other concerns. The implementation of the buffer concern is tangled with the autosave, logging, file system, multithreading and markers concerns in a way that is difficult to separate, at least not without disrupting the logical flow of the algorithm.

In order to determine whether the implementation scenario illustrated by the buffer concern is indeed representative for the other concerns in jEdit, we analyzed the data collected in the study for the correlation between the size and shape of the source code fragments for a concern and the existing source code modularization induced by classes and methods. In order to do so, we categorized the source code fragments for each concern into three types:

- Classes: the source code fragment covers an entire class’ source code;
− Methods: the source code fragment is not the entire class, but covers one or more complete methods; and

− Lines: the fragment is smaller than a class, and covers individual lines of text.

The graph in Figure 5 shows the relative percentages of these three types of fragments. For each concern, a vertical bar shows how the total number of fragments is split between entire classes (the lower section of the bar), methods (the middle section) and lines (the upper section). The last column shows the cumulative data for all of the concerns that we have looked at. When interpreting the data in this figure, it is important to realize that all columns were normalized; the actual values for each of these columns are shown with numbers inside each segment of the columns. The sum of all these numbers for a column is identical to the dotted line graphed in Figure 3.

A key observation stemming from this analysis of fragments is that:

*The majority of the source code fragments do not align themselves with the existing decomposition elements (classes and methods).*

Although we can observe some variation in the data, and for a small number of concerns the entire implementation can be limited to classes (UI, UI helper, multithreading, messaging, application,
printing), the other types of concerns have fragments that are mostly implemented as groups of individual lines of code. Therefore, if we were to limit ourselves to only trace a concern in code to entire methods and entire classes, then we would not be able to precisely trace 75% of source code fragments, which is the percentage of the top segment of the last column in the graph (3985 out of a total of 4837 fragments).

This loss of precision translates into more time spent by developer analyzing the methods or classes traced and finding out which exact fragments inside these methods or classes are indeed related to the concern. Therefore, the difference in size between the fragments and the size of their enclosing modules is very important in determining whether fragments can be successfully approximated by their enclosing modules. We calculated this difference for each source code fragment that was categorized as “lines of code” (in Figure 5). The resulting proportions between the average length of the fragments and the average length of the difference between the length of the fragment and the length of the enclosing method are shown in Figure 6. We can see that for some concerns, trying to approximate the fragments by their enclosing methods is unreasonable, while for some other concerns it would possibly be reasonable. However, the overall precision for such an approximation...
would only be close to 50%.

The observation that we can conclude from this data is:

The modules that encapsulate software fragments related to a concern are not a precise approximation for the implementation of the concern.

2.3.3. Concerns in Software Evolution

In the previous sections, we have seen the difficulties that software development faces in mapping concerns to the modularization of a system. As a result, concerns become scattered and tangled over a system’s implementation. This scattering and tangling of concerns induces two major problems related to software evolution: it makes the implementation of changes difficult and, at the same time, it makes the evaluation of the overall design of the system difficult. A well-designed system will have less scattering and tangling, which translates into less effort needed to implement changes to the system that affect these concerns.

The first problem is related to the implementation of a concern in code. Changes to an existing system are conceived with respect to the concerns that the system implements. For example, in a situation where a feature needs to be changed in order to accommodate new requirements for that feature, developers have to determine where the feature is implemented in the code. The more scattered and tangled the feature’s implementation is, the more difficult finding these source code fragments will be. New features to the system will not exist in isolation, but will instead need to interact with other features that are already implemented. In order to evaluate the best implementation of this addition to the system, developers will have to find and analyze the source code fragments that implement the related features first.

Since concerns are not mapped to their related source code fragments explicitly, developers have to rely either on their experience and knowledge about the system or on their ability to quickly understand the code. Neither of these methods is inherently accurate or efficient. Documentation separate from the implementation can be incomplete or inaccurate, and even if developers find such documentation, they will still need to inspect the system’s implementation in order to find out
particular details or to ensure that the system adheres to this documentation. Existing hints in the keywords used in code can help, but the results of the search still have to be carefully analyzed and a judgment call has to be made on whether or not the result is indeed related to the concern. For example, in searching the code related to a “logging” concern, developers might look for matches to strings such as “logging”, “logger” or “log”. Searching for a pattern like “log*” might return all these three options, but might also return matches like “logarithm” or “login”, which are not related to the concern and therefore not relevant. Additionally, searching for these textual hints is a time-consuming process, and if the result of this search is not saved, other developers might potentially perform the same search process later, which leads to more lost time. Our experience in finding the implementation of concerns in jEdit, described in Section 2.2, confirms this scenario. We used keywords to help find the fragments, but keyword collision in almost every case forced us to manually inspect and validate each individual software fragment.

The second problem of scattering and tangling during evolution relates to the ability of developers to evaluate, or foresee, the effects of their changes before they start implementing them. If developers know where different concerns are implemented in the system, and what their levels of scattering and tangling are, then they would be able to better understand the effort required to implement a change, and better foresee the potential effects of their change. For example, a change to a concern that is only implemented in a few lines of code will not be difficult to implement. If a concern is scattered in a large part of the overall system, then changing this concern would require more effort. And, if the implementation of this concern is tangled with the implementation of other concerns, then the complexity of the change increases because all the other concerns need to be considered too.

For example, in the case of jEdit, the buffer concept is of critical importance to the entire application. However, its implementation is not highlighted in the modularization in an obvious manner: it is not a high-level component, it is not implemented in one package only, and it is not encapsulated in only a few source code files. Given the importance of the buffer concept for jEdit, it is safe to assume that its importance translates into a sizeable part of the implementation. The results of the jEdit study confirmed this by showing that the implementation of the buffer concern has 16000 lines, scattered
across over 100 classes.

Developers therefore need to be aware, when changing the code, of all the consequences (intended and unintended) of their changes with respect to scattering and tangling of concerns. This is not easy without specialized tools that make developers aware of where concerns are implemented in the system and of their scattering and tangling. Without any tools, it is difficult to identify where even one concern is implemented in the system. The problem becomes much more complex if a larger number of concerns that are relevant for the system need to be evaluated and weighted with respect to one another.

Compounding to the problems that scattering and tangling of concerns impose on implementing changes to the system and on maintaining a good design is the fact that concerns evolve themselves as the software evolves. If developers are not aware where a concern is implemented, then, with each change, scattering and tangling of concerns can become worse.

One would like to think that software systems are carefully evolved, and that the decisions on where to place new functionality are taken based on revising the initial design. However, in practice, this is not always possible. In practice, developers have a difficult time during evolution to adhere to the principle of separation of concerns of the system.

When developers are not aware of how concerns evolve, and they do not plan their changes with respect to concerns, the overall design of the system deteriorates with time. If a change needs to be made to the system, and that change affects a number of different concerns, the lack of a clear picture where each of these concerns are implemented in the system generally leads to “naive” changes to the implementation where the code tends to be added to the most prominent of those concerns. For example, when adding the buffer save concern implementation, developers would be tempted to add a “save” method to the already existing implementation of the buffer concern. Over time, the implementation of this concern will become bloated with a number of other tangled concerns, and reach a point where it becomes difficult to add new changes to it. This situation will usually prompt a re-design of the system.
We can see some evidence of this happening in jEdit’s case. Figure 7 shows the evolution of two source code elements related to the buffer concern: a class named Buffer.java, found in the main package of the jEdit source code base (org.gjt.sp) and a package named org.gjt.sp.buffer that also contains code related to the buffer concern. The size of the Buffer.java class, in lines of code, is marked with triangles. The total size of all the classes in the org.gjt.sp.buffer package is marked with circles. The number of classes in the org.gjt.sp.buffer package is denoted with a dashed line, and the values correspond to the right-side axis.

It can be observed that the evolution of the buffer concern, and the Buffer.java class, had a period of radical changes before revision 6000, after which their development stabilized and the concerns remained fairly unchanged. At revision 4000, the Buffer.java class was already very large, containing far more lines of code than the other five classes found in the org.gjt.sp.buffer package. A particularly interesting point in the evolution of the Buffer.java file can be observed in between revisions 5000 and 6000. Code that was in the Buffer.java class was moved to other, separate classes, still related to the buffer concern, but now in the org.gjt.sp.buffer package.

We also observe that the evolution of the org.gjt.sp.buffer package followed a different pattern.
While in between revisions 4000 and 6000 both the number of classes and the number of lines of code increased sharply, they both declined in between revisions 7000 and 8000. For the rest of the time, the org.gjt.sp.buffer package remained about the same size in lines of code, but two more classes were added that raised the number of classes from 14 to 16.

Another important observation that we can draw from this example is that the evolution of a concern is not necessarily correlated with the evolution of the system as a whole. Figure 8 shows the evolution of the number of classes related to the buffer concern (the same information shown in the dashed line in Figure 7) over the total number of classes in JEdit. The number of classes for the buffer concern is shown with a dashed line, with values shown on the left-side axis, while the total number of classes is shown with a continuous line and values on the right-side axis.

While the total source code size of the entire system increased gradually over time, the evolution of the Buffer class and of the org.gjt.sp.buffer package showed sharp changes. These changes in the implementation of the buffer concern also mean changes in the scattering and tangling levels of this concern. If we consider the number of classes that implement the concern as a measure of its
scattering, then the variations in the number of classes of org.gjt.sp.buffer package (plus one class in the Buffer.java file) indicate variations, over time, of its scattering from 11 classes at revision 4000 to 19 classes at revision 6000 to 16 classes at revision 11000. If we consider the classes in the org.gjt.sp.buffer package less tangled with other concerns (which is one of the reasons they were moved to a dedicated package), then the code at revision 5000 is more tangled with other concerns then the code at revision 6000.

2.3.4. Concern Identification

Besides the data collected from the study, an important source for observations was the exercise of performing the study itself. One of the insights that became clearer while performing the study is that:

*Concern identification after the fact is an inherently ambiguous process.*

There are different factors that can make concern identification in code more or less accurate when it takes place well after the code has been written and possibly by others than those who wrote it. The ambiguous nature of the process of identifying concerns in code is due to the fact that, during this process, one must try to assign a semantic meaning (the concern) to a group of syntactic elements (the code). Inherent to the process are subjective decisions of the human marking the source code with respect to what should be included in a fragment, and to the understanding of the human of what a concern really means.

The first of these two problems, the decision as to which code should be included in a source code fragment, is influenced by how the human interprets the code, and how easy it is to determine the rationale for the code. We can assume that an expert developer will need less time than one who sees the code for the first time. This expert knowledge about the code is an intangible and unquantifiable factor that affects the quality of the concern identification process. The developers who actually wrote the code are probably the best candidates for concern identification. However, even they can forget the rationale for the code over time.

In addition, the specification of a concern is ambiguous in itself. The description of a concern is
informal, and therefore the human has to decide how this informal description translates into source code fragments. In the process of linking certain source code fragments to a concern and leaving others not linked, the developer is creating a more precise definition of what the concern means.

A second insight from the experience of conducting this study is that:

*The meaning of a concern becomes clearer as more source code fragments are identified and related to it.*

As the developer is faced with the decision of whether or not to link certain fragments to a concern, the original definition of the concern becomes more and more clearly shaped. In our experience, although a concern is thought to be clearly described, or “common knowledge”, the inspection of the source code base revealed situations that were not thought about a priori.

For example, one of the concerns that was identified at the beginning was the *user interface (UI)* concern. User interface is a general, high-level concept that is part of jEdit’s domain. A reasonable description of the UI concern can be “everything that has a visual representation”. This includes user interface widgets, such as buttons, menus, etc., as well as other features that manifest themselves visually, such as text markers, highlighting, cursor indicators, and others.

However, following the definition of concerns that we gave in Section 2.1, and which we used as a guide for the identification of concerns in the jEdit code base, the concern has to be the *rationale* for the existence of the source code fragments that are linked to it. One of the situations that became apparent during the study, but was not initially considered, was that a considerable number of source code fragments were not directly user interface elements, but the only rationale for their existence was the user interface concern. For example, event listeners are not user interface widgets, but they are closely tied to the user interface concern. Data models that connect the rest of the application to the user interface elements, classes that need to implement specific interfaces from the user interface libraries and glue code for different user interface elements are all examples of source code fragments which are not related directly to *UI*, but which are an integral part of the UI concern.
This example is meant to illustrate the kinds of decisions that had to be made during the concern identification phase and that directly influenced how the concern model was constructed and populated in the end. In the situation presented above, we had to decide whether to include the newly found fragments in the concern model as linked to the user interface concern (and therefore to expand the original intent of a concern to include auxiliary elements that are related to the user interface but are not visual elements) or to create a new concern whose description would include these user interface helper implementation fragments. Each of these two decisions will result in a different concern model, with a different organization. (In the case of our study, the second option was used, and a separate concern UI helper was linked to these source code fragments.)

Clearly, the fact that we were unable to foresee all the situations that need a reconfiguration of the concern model is to one extent a consequence of the unfamiliarity with the existing system’s implementation. However, this is an inherent problem, since most developers contribute only a small part of a larger system, and therefore will not know the details of the implementation of other parts in the system. As a generalization, it is reasonable to expect that the ambiguous nature of concerns is a problem for a scenario where multiple developers share and maintain the same set of concerns.

2.3.5. Concerns in Architecture

Software architecture is a part of our approach to addressing the problems related to concerns and system evolution as described in the previous sections. We discuss software architecture in this section in order to understand its purpose in software development, its relation to concerns, software development, and evolution, and the problems that the use of software architecture still faces.

The primary purpose of software architecture is to provide a high-level abstraction of a software system. As such, the architecture is typically created before the implementation, as the first step towards transforming requirements into a system design. Designing the architecture of a system as such involves the identification of the main elements in the system and their interactions.

The decisions about which elements are the main elements that need to be outlined in the
architecture typically follows the principle of separation of concerns. Which concerns become components is dictated by identifying the important pieces of functionality, either from the analysis performed by the developer, or as a result of following architectural styles that are proven appropriate for different types of applications [Shaw and Garlan 1996]. For instance, the “client-server” architectural style specifies an architecture where multiple clients access shared services exposed by a server, and it is a proven architectural solution for cases where scalability is an issue. The “peer-to-peer” architectural style also addresses scalability, but in a setting where data can be distributed across the components. The “layered” architectural style is commonly used to separate functionality in a system, where components that implement data stores, logic processing and user interface would be placed on the same layer.

The architecture is thought to be a blueprint that is further refined during lower-level design and the system’s implementation. The architecture, as reflected by the components and the structure of the system, will therefore become encoded in the implementation. In a typical development process, this encoding is implicit. A written documentation outlining the architecture and the rationale behind its structure might exist, but in the majority of cases the overall structure and behavior of the system is just a shared understanding among developers. For instance, a web-based software system is implicitly client-server, with the client executed in a web browser, and the server executed in a web server such as the Apache HTTP Server [The Apache Software Foundation 2004].

Software architecture research proposes the use of architectural models to express the architecture of the system explicitly [Garlan and Shaw 1993, Medvidovic and Taylor 2000, Perry and Wolf 1992]. These models are described using specialized Architecture Description Languages (ADLs), with examples including Rapide [Luckham and Vera 1995], Darwin [Magee and Kramer 1996], Wright [Allen and Garlan 1997], ACME [Garlan et al. 1997] and xADL2.0 [Dashofy et al. 2001]. Although these languages exhibit some variability with respect to the features that they support, they commonly support the description of a system in terms of its components, connectors, interfaces, and overall configuration [Medvidovic and Taylor 2000]. The difference between components and connectors is semantic: components are “the loci of computation” while connectors are “the loci of
communication” [Shaw et al. 1995], meaning that components are responsible for producing and consuming data while connectors are responsible for transporting the data between components.

A key aspect of an architectural model is its associated graphical representation. This representation is typically a boxes-and-arrows type diagram showing the components and their interconnections. Components are typically shown as boxes, their public interfaces through which they consume services from other components or publish data to other components are shown as smaller boxes on the border of the components, and the connections between components are lines that connect interfaces of different components. This basic architectural view supports the goal of an architecture providing an easy to understand, clear view of the high-level organization and properties of a system.

The decomposition into components and connectors is one particular architectural view, and there are other types of architectural views that have been proposed. Kruchten proposes the 4+1 view model of software architecture, with four separate views describing the system based on its logical, development, process, and physical decompositions, tied together by a fifth view which consists of usage scenarios [Kruchten 1995]. Soni, Nord and Hofmeister [Soni et al. 1995] similarly propose four types of architectural descriptions: the conceptual, module interconnection, execution, and source code architectures. Clements et al. [Clements et al. 2002] use a module view to show the static decomposition, a component-and-connector view to describe the runtime behavior, and an allocation view to show the physical allocation to environmental elements, such as hardware, implementation infrastructure, and human development teams.

While having an architectural model of a software system can serve multiple purposes, and the different views can be used to analyze the model from different points of view, one of these views has a most direct impact on software development. Particularly, the source code view of the architecture represents the primary decomposition of a system in the same sense as the modularization method proposed by Parnas. The system’s functional features, or “loci of communication”, are not only identified separately in the architecture, but are also encapsulations of
the underlying source code that implements the feature.

The components in an architecture encapsulate some concerns, which are considered important from a system-wide perspective. While these concerns are clearly shown as components in the architectural design model, other concerns, even if they are implemented somewhere in code, are not shown at all. For example, if we look back at the case study presented in Section 2.2, a number of concerns from that list will not show in an architectural design model for the jEdit system. Logging and exception handling are implementation concerns that are not “loci of computation”. The buffer and edit mode concerns represent data objects that are also not suitable to be components in the architecture. UI can be a component, but because of its size, which is almost half the implementation of jEdit, will probably be scattered in multiple components. At the same time, gutter and commands are sub-concerns of the UI concern, but are probably too small to be shown in the architecture and therefore are not going to be visible in an architectural model.

There are two different fundamental views on what the role of software architecture should have in software development. One envisions software architecture in a prescriptive role, where the architecture is used as a design model, created before the implementation of the system, which is a blueprint of how the system should be implemented. The other view envisions a descriptive role for architecture, as in showing a high-level representation of how the system is actually implemented.

2.3.6. Architecture and Implementation

Underlying both these views of the architecture is an implicit expectation that the architectural model represents faithfully the properties of the underlying implementation. In a contrary case, where the architectural model shows some properties, but the actual implementation of the system does not satisfy these properties, most analyses performed on the architectural model would be useless or at best misleading [Medvidovic and Taylor 2000, Moriconi et al. 1995, Shaw and Garlan 1996]

In order to be able to verify the adherence of the implementation to the architecture, one has to be able to first determine a mapping between the elements of the architecture and the elements of the implementation. One solution to this problem is to embed the architectural description into the
implementation. ArchJava [Aldrich et al. 2002] is an architectural language that extends the Java programming language with new constructs. The code and the architectural language are written together and a compiler is used to enforce consistency properties between the architecture and implementation. However, while this type of approach provides such a mapping between the architecture and code implicitly, it is also limiting the architectural description to only concepts that are relevant in the implementation, and does not include constructs for other types of architectural properties. Determining a general type of mapping between an architectural description as separate from the implementation is still an open research problem.

Such a mapping also needs to be maintained throughout the evolution of the system. During evolution, the prescriptive architecture and the descriptive architecture of the same system might become disparate. The “ideal” architecture and the “as implemented” architecture can become, with time, significantly different from one another. A compelling example of this situation is given by Bowman, Holt et al. in a case study regarding the architecture of the Linux kernel project [Bowman et al. 1999]. The study showed that the architecture recovered from the implementation had far more connections that the idealized version that developers thought it had, amounting to almost a complete graph. While at the beginning of development developers shared an understanding of the architecture of the system, with time the as-implemented architecture starts to drift away from it.

For the architectural model to be useful during evolution, the mappings of the components to the implementation have to be maintained gradually over time, with every single smaller change to the implementation. Otherwise, as research in architectural recovery methods shows, the process of determining these mappings from an existing implementation is critical knowledge that requires the guidance of a human, expert developer [Biggerstaff 1989, Harris et al. 1995, Murphy et al. 2001, Riva et al. 2001]. By maintaining the architecture in small steps, this effort can be offset, and the architecture can become both a prescriptive and a descriptive model of the system. This architectural model can then be used to show system-level properties of concerns, such as where concern are implemented, how much of the system’s implementation is related to certain concerns, and how concerns are scattered and tangled in the system’s architecture.
2.4. Ideal Support

In this chapter, we have discussed the underlying problem related to modularization of concerns in implementation, and the problems that arise during software evolution. We have also presented software architecture and its importance related to concerns, implementation, and evolution. Following these discussions, we propose here a set of requirements that an approach needs to satisfy in order to support a concern-oriented development process.

**Requirement 1. Concerns have to be visible during software development.**

The fundamental issue that arises from the discussion in the previous sections is the fact that, despite the need to reason about a software system in terms of concerns, concerns are not made an explicit part of, or given an explicit representation in, software development. Information pertinent to concerns is not captured during development, and is therefore lost. Even though developers typically have initially designed a system based on the principle of separation of concerns, once the implementation of the system is in place, there is no clear way of determining which concerns have been addressed and in which form.

**Requirement 2. Concerns have to be mapped to their implementation.**

Once concerns become visible in implementation, they need to be mapped to the actual implementation. We have seen, in the previous sections, that concerns are fragmented over multiple implementation elements. While some concerns might be encapsulated in classes or methods, other concerns are implemented as small code fragments, sometimes even individual lines of code, dispersed over a number of classes or methods.

Without a clear mapping of concerns to the implementation fragments related to them it is easy to lose track of where concerns are implemented. The mappings, if created, can be used to determine, in the event that a concern’s implementation needs to be changed, where in the source code developers should focus their attention while implementing the change. Reversely, the developers should also be able to see, for a certain source code fragment, which concerns, if any, are directly related to it.
Requirement 3. Concerns and their links to related source code fragments need to be precisely maintained as the software evolves.

As software evolves, and changes are added incrementally, it becomes virtually impossible to determine which concerns have been implemented and where these concerns are implemented in the system. Software evolution is in itself complex, because implementation elements are continuously added, deleted, and modified. With concerns being mapped to an implementation, this complexity is bound to increase in scale, because of the natural scattering and tangling of concerns over these elements. A solution to this problem has to include a mechanism that will satisfy the third requirement.

Requirement 4. Developers have to be made aware of scattering and tangling of concerns.

Scattering and tangling of concerns directly influence the quality of the system and its ability to change in the future. Every change to an existing system can potentially increase or decrease the level of scattering and tangling of its concerns. Without being able to assess the levels of scattering of tangling and scattering of concerns, developers are more likely to plan suboptimal changes, that deteriorate the quality of the design of the system over time. Visualizations and metrics related to the scattering and tangling of concerns have to become an integral part of the development environment.

Requirement 5. Concerns need to be abstracted onto architecture.

Another observation based on current development practices is that current development practices lack a high-level representation of concerns. Such a high-level view of concerns is needed in order to abstract away the complexity of the details of concern to implementation mappings. The information presented in these mappings can be used to determine the local importance of concerns (such as, for example determining which concerns affect a given line of code), but this level of detail is overwhelming in determining the system-level importance of concerns. The architectural-level abstractions of the scattering and tangling of concerns are properties of the system that can be used to evaluate its design quality.
3. Research Questions

In the previous chapter, we introduced the concept of concerns and presented the problems that software development faces today with respect to using concerns in software evolution. We also presented a set of requirements of what we considered to be the ideal support that an approach will have to provide. In this section, we discuss the main research questions that underline our work towards creating such an approach.

The pertinent research questions are:

1. **Can concerns be made visible in code?**

   We have seen that in a regular development setting, concerns are not visible, which leads to a number of problems in software development. The purpose of this research question is to determine whether an approach that makes concerns visible during development can be built, effectively testing the feasibility of such an approach.

2. **What level of support is needed in order to precisely maintain the concerns and their mappings to code during evolution?**

   Concerns need to be maintained throughout the coding exercise. In the previous section, we discussed the complexity of this problem. If the level of support provided by an approach is not sufficient to precisely maintain concerns and their mappings to code, then the concern model can slowly become out of touch with the code, and therefore lose its relevance.

3. **What are the effects of using such an environment?**

   The introduction of concerns in software development by making them visible and providing methods to maintain them during development represents a new kind of capability that is not currently available. Provided that such an environment can be built, and the concerns can be
effectively maintained, analyzing a system based on this information is still an important paradigm shift from the usual analysis based on modules. We have proposed scattering and tangling of concerns as two indicators of the evolution of a design’s quality, but because no environment exists that is able to maintain concerns to code during evolution, no such analysis has been performed to date. It is an important research objective to determine the potential benefits of providing this new, concern-based information to developers, and what the resulting effects of using concerns to guide software evolution are.

We believe that concerns can be made visible during development by integrating support for concerns within the software development environment. During programming, the developer is encouraged to identify concerns by being informed of where and how concerns are implemented in the system. Furthermore, the environment provides support for maintaining the mappings of concerns to fragments as both evolve. With such an environment, then, it becomes possible to maintain an accurate, up-to-date concern model that can then be examined as to whether a desired design structure – from the perspective of concerns – is still present and will continue to be present with future changes.

The main hypothesis of this dissertation is:

Hypothesis. A development environment in which concerns are continuously and incrementally identified by users and visualized by the development environment will result in a well-maintained concern model that can be used to analyze the design quality of a system based on concerns.
4. Approach

The approach proposed in this dissertation can be described as concern-based evolution, a novel development paradigm where concerns drive software evolution by being continuously used and maintained throughout the life of a software system. The approach is supported in the form of ArchEvol, a development environment where: (1) concerns are explicit first-class entities, (2) concerns are maintained while the code is being written, and (3) levels of scattering and tangling are continuously shown so that developers using ArchEvol become aware of how the system evolves with respect to concerns and can take appropriate action if this evolution is not as desired.

Central to our approach is the continuous use of concern information. Throughout a system’s evolution, concerns are continuously identified and maintained so that they guide the changes to the system. Low-level concern information is kept in sync with the code, while high-level properties such as scattering and tangling are made amenable for system analysis by showing them at the architectural level.

An overview of our vision for concern-based evolution is presented in Figure 9. The figure shows the evolution of a software system as a list of versions, with each transition from a version to another representing an evolutionary step that enhances the system in order to accommodate new requirements. While typical software projects rely on code as the only representation of a system, in our approach architecture is introduced as a second, equally indispensable representation of the system. An architectural model of the system is explicitly described and maintained in a close relationship with the code throughout the life of the software system as it is being developed. While in the code the concerns are described at a fine level of detail, the architecture becomes the place where high-level information about the concerns summarizes the state of the system at a given moment in time.
Principles

In designing our approach to address the research questions presented in the previous chapter, we were guided by a number of principles. These principles are based on insights gained from both taking a closer look at the original intent of the principle of separation of concerns and from our evaluation of the type of support needed in current software development practices.

Principle 1. Modularity is left as a choice to the developer, and concerns are identified and maintained over the fragments of the modules that are chosen.

The original intent of the principle of separation of concerns, as envisioned by Dijkstra, is to be able to reason about concerns in isolation from others. This does not necessarily mean that we need to encapsulate every single concern in its own module. As we have shown in Chapter 2, this is generally not even possible, because of the inherent scattering and tangling of concerns. The direction taken by our approach is to maintain concerns separate from the implementation modules and to link them to the module fragments for which they provide the rationale behind their existence.

If these links can be maintained accurately during development, then localization of concerns in implementation is as effective as in the case of encapsulation. In addition, using explicit concerns has the advantage that the two critical properties of concerns described in Section 2.3.1, namely

Figure 9. Approach overview.
their generality and overlapping nature, can be accommodated without imposing unnecessary constraints on the implementation structure of the system. The complex nature of concerns can be managed separately from, but alongside with, the complex structure of the implementation.

**Principle 2. Support the user in capturing concern information.**

This principle underlines our belief that, in determining if a concern is related to source code, the developer writing the code is the primary source of information about concerns. When writing the code, the developer should know to which concerns the code is related, since the concerns provide the rationale for the existence of the code and for the choices made while writing it.

This type of information is typically not captured at the moment when developers are writing the code, with only hints of the relevant concerns being left in comments, element names, or documentation. When other developers inspect the code in search of a concern, or when specialized tools use heuristics to determine where concerns are implemented, they are in fact attempting to reconstruct the original intent of the developer. The results of this reconstruction are approximate and can be inaccurate. Therefore, the environment should support capturing the original intent of the user at the moment it is entered, so changes to the implementation can be precisely related to the concerns.

**Principle 3. Support the user in understanding the system and in making informed changes based on concerns.**

While the goal of a modularization is to reduce the scattering and tangling of certain concerns, other concerns will naturally become scattered and tangled. Deciding what is an acceptable level of scattering and tangling for a concern or set of concerns is, however, a subjective matter. There is no fixed value that will be considered universally acceptable, and therefore the user should be able to decide, based on various references (e.g., application domain, organizational rules, internal preferences, experience, etc.), what acceptable levels of scattering and tangling are for a particular concern or set of concerns. In turn, the environment has to support the user in taking informed decisions about the design of the system by providing information about where concerns are
implemented, if concerns are aligned to the modularization of the system, and how relevant concerns are for the implementation of each module in the system.

*Principle 4. Maintain and monitor concerns continuously.*

This principle states that concerns, and their implementations, have to be monitored continuously over the life of the system. As we have already discussed, scattering and tangling are dynamic phenomena. The levels of scattering and tangling for all concerns in a system vary over time, because with every little change that a developer makes to the existing source code, new implementation fragments that are related to one or more concerns are added, modified, or removed. In order to be able to precisely maintain concerns, the developer has to be encouraged to actively participate in their maintenance, and be involved in linking concerns to code fragments with every change.

Continuous, incremental maintenance of concerns is key to being able to maintain a large body of information about where in the system various concerns are addressed. The identification of concerns and the maintenance of concern implementation information should ideally start from the beginning of a system’s lifetime, such that the effort of updating concern information is in concordance with every single (generally small) change to the implementation of the system.

*Principle 5. Use architecture as a design model that reflects on code.*

The architecture is a high-level representation of the design of the system. As we have discussed in Section 2.3.5, architecture has traditionally been used in two ways: either as a prescriptive description of the intended design, made before the implementation, or as a descriptive picture of the status of system’s design, generated after the implementation is done. We attempt to unify these two views by having the architectural model as a permanent representation of the system, accompanying the source code at all times. The architecture starts as a prescriptive description, and helps creating the implementation. However, once the development of the implementation starts, the architecture becomes the main view that reflects the properties of the system as implemented. In our case, the information shown in the architecture is about concerns, and their levels of scattering
and tangling. The architecture becomes, in this way, the primary design view of the system.


As we have already seen, concerns should be made visible during implementation. Our approach maintains concerns explicitly and links them to source code fragments, but also provides a number of visualizations of concerns. These visualizations show where concerns are implemented in the code and how their scattering and tangling manifest themselves at the architectural level, both with the purpose of making the developers aware, at all times, of how concerns are implemented in the system.

Additionally, by maintaining the concerns continuously over time, our approach offers a visualization of not only the snapshot of concerns at a given point in time, but also the evolution of concerns as the system itself evolves. It is this historical data that we believe is essential in gauging acceptable and unacceptable levels of scattering and tangling in various modules of a system. Developers can observe the evolution of concerns, and assess whether it was stable or marked by periods of variations up or down.

4.1. Features

The principles described in the previous section guided ArchEvol’s design and the identification of its main features. As shown in Figure 10, at the center of ArchEvol’s design lies the concern model, which documents the concerns that are relevant for the system. The concern model is the point where the two main functionalities of ArchEvol meet. The first main feature is responsible for offering support for concern identification, the process through which concerns are linked to the source code. The second main functionality is responsible for using the data stored in the concern model and visualizing it in both the code and the architecture of the system. The underlying infrastructure of ArchEvol ensures that all this information, including the source code, the concern model, and the architecture, is properly evolved so that the links are maintained accurately over time.
4.1.1. Concern Model

The purpose of the concern model is to retain and organize the concerns that are relevant for a software system. By maintaining these concerns explicitly in a concern model, ArchEvol addresses the first requirement from Section 2.4, which states that concerns have to be made visible in development. During development, the users will refer to the concern model to find concerns and their related source code fragments.

In designing the concern model of ArchEvol, a number of choices were made regarding the representation of concerns, the representation of the software fragments, and the mechanisms for maintaining the links between the concerns and software fragments. Different choices for each of these decisions would have resulted in different outcomes with respect to the performance, usability, and overall usefulness of ArchEvol, especially since our entire approach relies on the information stored in the concern model. We therefore carefully weighted each of the available choices with respect to the desired properties that the concern model needs to have.

Two desirable properties of the concern model are its precision and completeness. The concern model is precise if a concern does not link to fragments of software other than the ones strictly related to it. The model is complete if, for each concern, all the software fragments that are conceptually related to it are linked in the model. These two properties are desirable because they impact all the other features of ArchEvol. An inaccurate or incomplete model would make the
analysis of the system irrelevant and misleading, and will not be useful in providing reliable information about the status of the system.

The concern model also potentially needs to accommodate a large number of concerns. As a consequence of the generality of concerns, the size of the concern model can vary based on what concerns are determined important by the user for the purpose of analyzing the system. The more concerns are being modeled, the more information is available to support future changes. However, the verbosity of the concern model comes at a cost, as it requires more maintenance effort to maintain the concern-to-code links.

**Concern Representation**

A concern in ArchEvol is represented by its name and description. Given the general nature of concerns, as we described in Section 2.3.1, the concern model has to be generic, in that it cannot associate any semantic meaning to concerns. Developers can use the description field to provide more information about the scope and purpose of the concern and to avoid confusion in the case that the name is not sufficiently clear.

In order to accommodate concerns with different levels of granularity, the concern model in ArchEvol is organized in a hierarchical structure, where larger concerns can be decomposed into sub-concerns. As a result, if the software fragments that are related to a concern are too coarse-grained, sub-concerns can be used to link source code fragments that are smaller in nature. For example, in our initial example of jEdit from Chapter 2, we identified a serialization concern, responsible for saving buffers to the file system and loading the content of the buffers from the file system. However, since serialization is a general concern, it has a large number of source code fragments associated with it. In this case, the concern can be further decomposed into smaller-level concerns, such as load, save, autosave and backup, which all represent smaller-grained functionality related to serialization. These sub-concerns have fewer fragments associated with, which is more manageable for a user exploring the concern model.

A generic concern model has been proposed in the Concern Manipulation Environment (CME)
[Harrison et al. 2005], which is an Eclipse plug-in that manages a concern model. This concern model is organized as a graph, where a concern has both a number of different children and belongs to a number of different parents. The relationships between concerns are modeled as separate entities in the concern model. The concern model in CME was designed to be very general, and its implementation contained builders that can assign Java elements to a concern. CME supports both explicit sets of elements to be linked to a concern, or can evaluate queries over the programming elements to determine these sets.

Unlike CME, ArchEvol does not have the purpose of providing a generic framework for classification of concerns. A taxonomy of concerns, in a particular domain or as used in a particular development method, is out of the scope of ArchEvol. Rather, the decision of which concerns to add to the model is left as a choice to the users, as to what they find important. The set of concerns will differ based on the domain of the problems that the specific software project addresses, will be based on the assessment of the users as to what will likely be changed in future, and can be enriched later, as the users become more knowledgeable about the system.

In ArchEvol, the users can add additional concerns to the concern model at any point in time during development, or can re-organize the hierarchy of the concerns if they determine this to be necessary. The concern model views include support for selecting a number of different source code fragments and assigning them to a new parent concern. It is also expected that, as software development progresses, the users might find some concerns to be too high-level and needing to be split into smaller sub-concerns, or that they need to group a number of individual concerns as children of a higher-level concern. The concern model can thus easily be reorganized at a later time.

Once we decided how to model and store concerns, we needed to determine how to represent the links from concerns to the implementation. From the definition of concerns in Section 2.1, a concern conceptually links to a number of different source code fragments. The decisions that have to be made are about: (1) how to represent a set of related source code fragments and (2) how to represent each of the fragments individually.
Fragment Set Representation

For the set of source code fragments, two alternative representations can be used: extensional or intensional [Harrison et al. 2005, Robillard and Murphy 2002]. In an extensional representation, the set of fragments is linked explicitly, element by element, to a concern. In an intensional representation, a predicate is used to express this set, and evaluating this predicate will ideally result in the same set of fragments as in the explicit form.

Related work includes examples of both intensional and extensional concern models. CME supports both intensional and extensional representation of concerns. FEAT [Robillard and Murphy 2002] is an Eclipse tool that provides a concern model in which concerns (represented as Concern Graphs), are a set of programming elements tied together through a query. ConcernMapper [Robillard and Weigand-Warr 2005] supports a simpler, extensional representation of concerns, that allows developers to manually link Java elements (classes and methods) to concerns.

Both of these alternatives have advantages and drawbacks. The extensional set is explicit, so it is easier to maintain when the set of fragments is relatively small. At the same time, it can become difficult to inspect, verify, and maintain if the set of fragments reaches a large size. This lack of scalability is one of its main problems, particularly during evolution. In the intensional form, the expression of a query can be dramatically more compact and clearer than its corresponding extensional expression, provided that the fragments that are linked to a concern are written in a more regular fashion that supports a clean regular expression. However, the more irregular the fragments and their locations in the code, the more verbose and nuanced the expression of the query will need to be.

There are two major considerations that led us to adopt the explicit representation of concerns. First, as we have observed in the jEdit study presented in Section 2.2, the concerns and their fragments are highly varied in their form and size. Our jEdit study revealed that many fragments are not full methods or full classes, and therefore cannot be referenced by name. The only way to reference these fragments is by instead referencing their enclosing methods or classes. This would mean, for
example, that if two lines of code in a larger method belong to a concern, then the entire method would need to be part of the predicate. We decided, for ArchEvol’s purpose, that this type of approximation is not an acceptable solution, as it would go directly against our goal of having a precise concern model. As we have shown in one of the observations in Section 2.3.2, the implicit representation of concerns would lead to a potentially significant loss of precision. The explicit representation of concerns, on the other hand, does not impose any restrictions on the format of the individual source code fragments, and, by accommodating fragments of different sizes and forms, can be more precise.

The second consideration against the intensional representation of concerns is that it requires much more attention from the part of the developer in order to maintain all the predicates correctly. A large concern model can potentially have a considerable number of predicates in order to reference all source code fragments for all concerns. In such a situation, the developers have to be aware how each change to the code influences all these predicates. For example, when developers add a new method called “logging”, it would fall under the incidence of an existing predicate that groups all methods that start with “log” (for example, the predicate expression “log*()”). This is the intended behavior of the predicate. But if the method is “logarithm”, then the predicate would incorrectly include it as being a “log” method. In this case, developers would have to revisit and rewrite either the new method, or the expression of the predicate. The more complex and numerous these predicates are, the more difficult rewriting them can become. The extensional representation is, on the other hand, easy to maintain because it only requires the developer to decide whether an individual text fragment is going to be added or not to a concern, and not revisit the rest of the concern model and evaluate whether it is still accurate or not every time they modify it.

As a result of these considerations, ArchEvol uses an explicit representation of concerns. While it is simpler and more verbose than the intensional representation, the extensional representation used by ArchEvol allows its users to create a more precise and complete representation of the source code fragments over time.
Individual Fragment Representation

The second decision related to the mappings from concerns to the implementation is the choice of the representation of the source code fragments themselves. The definition of concerns from Section 2.1 does not specify precisely how source code fragments should be represented. The only guideline, as the name "fragment" suggests, is that this definition imposes that fragments should not be limited to entire source code elements.

The two choices that are available for the source code fragments are related to the two complementary representations of the source code: an Abstract Syntax Tree (AST) representation given by the programming language structure, and a textual representation given by the organization of the source code files. Using the AST representation would mean that the source code fragments would be limited, in the case of Java programs, for example, to methods, classes, or statements.

The second option, and the one that ArchEvol uses, is to represent implementation fragments as links to the text of the source code. In ArchEvol, a software fragment is an interval of the form \([file, start, end]\), where \(file\) is a text file indicator and \(start\) and \(end\) are character indices in the file. For example, in a concern model representation for the backup concern in jEdit, as shown in Figure 11, there are two links to source code fragments in the \(SaveBackupOptionPane.java\) file, and one to a fragment containing configuration parameters in the \(jedit.props\) file. By referencing text fragments, there is no semantic attachment to what the fragment might mean. In this way, the concern model is not tied to a particular implementation language's semantics.

From the point of view of precision, both the textual representation and the AST representation are equally precise in the sense that, for every text snippet it is possible to determine which AST elements are touched by the fragment, and for every AST element there is a corresponding text interval that contains it. One difference is that the textual snippet might contain separators or white space characters that do not have a meaning. However, these are easy to ignore by a developer when inspecting such a snippet.
Concern Model
- Serialization
- Load
  - ...
- Save
  - ...
- Backup
  - [SaveBackupOptionPane.java, 11221, 14111]
  - [SaveBackupOptionPane.java, 15222, 16442]
  - [jedit.props, 154, 189]

SaveBackupOptionPane.java

jedit.props

Figure 11. Example of links to software fragments.
The difference between the two representations is in their flexibility. As the analysis of the jEdit case study from Section 2.3.2 shows, most of the source code fragments are significantly smaller than methods and classes, and are code snippets of arbitrary size or form. In order to preserve the desired level of precision, representing these fragments using AST elements would result in a very verbose fragment set. For example, in Figure 11, the highlighted text on the left contains statements for the initialization of seven variables, from which five are member variables and two are local, five calls to an addComponent() method, nine calls to getProperty() or getBooleanProperty() methods, and the use of ten configuration parameters. All these lines of source code are related to the backup concern in that they either initialize a variable that holds information about backup, create user interface widgets that show information about backup, and call methods that will link these widgets to the main panel. In the textual representation, everything can be captured together in a single text fragment reference. An AST representation would need to include at minimum the 16 individual statements.

4.1.2. Concern Identification

Concern identification is the activity of augmenting the concern model with links to source code fragments. While the concern model is responsible for organizing and storing the concerns and their links to the related source code fragments, concern identification is the process through which this data is collected.

A number of methods for linking concerns to code have already been proposed in the literature. Some approaches mine source code for aspect candidates through the use of automatic or semi-automatic methods of identifying crosscutting concerns [Breu et al. 2006, Marin et al. 2004, Shepherd et al. 2007]. However, the success of these methods depends on a number of factors: (1) how well naming conventions are used, (2) the clarity of the comments in the code, (3) the relevance of the keywords, and (4) the way the software is organized. Other methods, such as FEAT [Robillard et al. 2004], record the exploration process of a developer who wants to find out where a concern is implemented in an existing system. This method relies on the skills of the developer who performs
the investigation, and who might not be able to find out all of the places where a concern is implemented.

The method proposed here is different in that it tries to get the information about where concerns are implemented from the best source possible: the developer actually writing the implementation. As we discussed in the previous section, supporting the developer in collecting information about concerns is a guiding principle in ArchEvol’s design. The reason is that, when writing the source code for a particular change, the user should know which concerns are addressed by this change and how this change is going to affect those concerns: is the new code adding a new concern, removing parts of an old concern, modifying an existing concern’s implementation, or some combination of all of these?

Whenever developers are asked to maintain documentation in addition to the source code, which is the case of maintaining a concern model with links to source code, it is important that the methods used be as easy and unobtrusive as possible. Otherwise, developers might perceive maintaining this documentation as an additional effort and might forgo the steps required to do so.

The solution adopted by ArchEvol is to support two methods of concern identification: context recording and manual selection. Both methods use different paradigms that are appropriate for different situations, and as a result are complementary in nature.

**Context recording** assumes that the user declares, at the beginning of a development task, the concerns that the change is going to address. Similar to some change-driven configuration management systems that associate text changes to a tag, in ArchEvol, users can designate one or more concerns as the active concerns in the current development context, and all modifications to the current editor will be automatically linked to these concerns.

**Manual selection** is a feature of ArchEvol that can be used to change the concerns manually in the current text editor used by developers for coding. While context recording is suitable for changes that are continuous in larger segments of time, manual selection is useful for quick edits of the concern model. While writing code in the current editor, the user can access a pop-up menu option
that associates the selected text fragment with a particular concern.

These two methods are different from the point of view of their impact upon the development process, and each has its own strengths and weaknesses. Context recording assumes that the user will set and change the concerns in the current context every time the task is changed. Manual selection will leave the choice of updating the concern model to the user, but this might mean either that the development activity is interrupted to do so, or that the model is updated at the end of a development activity, which is more prone to the user forgetting to mark some fragments appropriately.

Table 4 presents an analysis of the different characteristics of the two identification methods based on different situations in which they will be used. Differentiating by the size of the source code fragments to be used, context recording is appropriate for large pieces of functionality, such as entire methods, while manual selection is more useful for smaller size fragments. When a task is going to change a number of concerns concurrently, the context recording method might require the user to frequently change the set of concerns forming the current context, which makes it less useful in this kind of scenario. At the same time, when the tasks are short, using manual selection is preferable to constantly changing contexts for each task.

### 4.1.3. Concern Visualization and Analysis

The primary reason for which concerns are modeled in ArchEvol is to help the user in making more informed changes to the system, in a way that respects the separation of concerns principle. ArchEvol provides a number of visualizations that have the purpose of making concerns visible
during development. Through these visualizations, the users are made aware of where concerns are implemented in the system and how the balance of concerns changes when the system’s implementation changes. By continuously informing the users about concerns, the system will be less likely to become out of date with respect to its original separation of concerns, or at the least do so in a desirable way.

As we presented in the previous section, the process through which the concern model is populated with links to implementation fragments is called concern identification. While concern identification provides data for the concern model, concern visualization is a consumer for this data. During development, the user will switch between these two activities. By continuously viewing where concerns are implemented, the user can analyze the system based on concerns, which in turn influences their future changes.

In ArchEvol, concern awareness is provided at both the code level and at the architectural level. The code and the architecture are used in a side-by-side view, each offering a view of the system at a different level of abstraction. While the visualizations in the code are changing with every small code modification, the information in the architecture will emerge slowly, over time. A single new line of code added to the implementation will have little effect over the entire system. However, after tens or hundreds of such modifications to the code, the distribution of concerns over the architecture can change dramatically.

**Code visualization**

The purpose of the source code level visualizations is to inform the user about the concerns that are implemented in code. Developers primarily use source code editors for two types of activities: (1) they inspect the source code in a file to understand more about the logic about the existing code and plan their changes accordingly, and (2) they modify the text of the files in order to implement the changes. Visualization of concerns as implemented in code is useful in both these activities.

For code inspection purposes, the goal of the visualization is to show scattered and tangled concerns over the entire file. As a single file can potentially have a considerable number of concerns,
and since concerns can overlap each other, showing this information over the code (by using highlighting or color-coding) will become overwhelming and confusing for the user. The solution proposed by ArchEvol is to move the concern information on the side of the text, in a visualization that is focused solely on concerns, but is still very close to the code. This visualization gives an overview of the concerns, and conveys to the user scattering and tangling of these concerns in a succinct and concise manner.

As shown in Figure 12, the concern information is concentrated on the side of the text, and shows, for each line of code, which concerns are implemented by that line of code. The three concerns that have fragments in the code snipped shown in the figure are labeled as “c1”, “c2” and “c3”. It can easily be observed that the concern “c1” is the predominant concern in this code snippet, and covers every line of code. This concern is not scattered, as it has one contiguous fragment, but is tangled at certain points with the concerns “c2” and “c3”. The concern “c2”, by contrast, is of a much smaller size, it is scattered because it has two fragments, and it is tangled with only one concern, “c1”. The concern “c3” also has a small fragment, and is tangled with only “c1”, but it is...
not scattered.

By showing which concerns are implemented at the very point where the developer starts writing a change, we help the developer in determining whether the change will increase or decrease the scattering and tangling of concerns at a lines-of-code level. Our hypothesis for the source code visualizations is that the user will not have to scan the code in minute detail anymore, because the places where a concern is implemented is already marked and thus will attract the attention of the user to the relevant places in code. A side effect of these visualizations is that the user will be more inclined to place the new code for a concern next to the already marked fragments for that concern, minimizing in this way the scattering and tangling of concerns.

**Architecture Visualization**

Higher-level information about how concerns are implemented in a system is displayed at the architectural level. The motivation for this type of visualization is that users should be able to see, at the architectural level, the scattering and tangling of concerns over the components in the system. The information in the concern model is useful, but without a way of abstracting its data and showing it at a higher level of abstraction, the sheer size of the concern model and the number of possible fragments linked to each of these concerns will make it impractical to draw any system-level conclusions about where concerns are implemented in the system. ArchEvol aggregates the data from the concern model and shows it over the architectural elements in the architecture.

ArchEvol uses two architectural-level concern metrics: Component Relevance for a Concern (CRC) and Concern Extent in Components (CEC).

- The Component Relevance for a Concern (CRC) is measured, for a given component and a given concern, as the ratio of the implementation files that have fragments related to the concern and belong to the implementation of the component, over the total number of files that have fragments linked to the concern from all the components in the system.

- The Concern Extent in Components (CEC) is measured, for a given concern and a given component, as the ratio of all implementation files that have fragments linked to the concern
over the total number of files in the implementation of the component.

These two metrics are complementary and symmetrical, in that one measures the relevance of a concern in a certain component, while the other measures the relevance of a component in the overall implementation of a concern. If a concern is addressed by the implementation of a component, then the values of both CRC and CEC will be greater than zero. If a concern is not addressed at all in a component, then the values for both metrics will be equal to zero.

Scattering and tangling of concerns can be observed by analyzing the values of these two metrics, and comparing them to zero. A concern’s implementation will be scattered over all the components where the values for its CRC and CEC are greater than zero. At the same time, all the concerns that have values greater than zero for CRC and CEC in a component are considered tangled in that component’s implementation.

However, these two metrics offer more detailed information about the level of scattering and tangling. Particularly CRC answers the question “What percentage of the implementation of a concern is contained in a particular component?” A concern that has most of its implementation in one component, and only a small fraction is present in other components, is less scattered than a component whose CRC are equal across all these components, even though both concerns are scattered over the same components. In contrast, CEC answers the question “How important is a concern to the implementation of a component?” A concern that is present in almost every implementation file in a component, and therefore has a very high CEC value for that component, likely defines the primary purpose of that component. However, if other concerns are also present, this concern is going to be more tangled with these concerns than a concern that has a low CEC value.

For example, in Figure 13, a hypothetical architecture with three components, A, B and C, and two concerns, X and Y, is presented. We see that concern X is scattered across all three components in the system. However, looking at the values of CRC reveals that this scattering is not equal over the three components, since component A contains 25% of the concern’s implementation, component
B contains 45% and component C contains the rest (30%). At the same time, it can be observed that concern Y is only scattered over components A and C, since its values for CRC and CEC in component B are equal to zero.

With respect to the tangling of concerns, we can observe that both components A and C have the two concerns tangled in their implementation, while component B only addresses concern X. At the same time, in component A’s implementation, both concern X and concern Y are equally important, since they both have a value for CEC equal to 50%. However, in component C, concern Y is more important since it is located in almost all of the component’s files, having a value for CEC equal to 80%.

4.1.4. Evolution Support

An important part of ArchEvol’s approach is the support offered for maintaining the source code, the architecture and the concern model throughout the evolution of the software system. In order to do so, we have to first define a mapping between the architecture of the system and its source code implementation, a mapping that is needed in order to calculate the values for the concern-based metrics discussed above. Second, this mapping needs to be maintained throughout the evolution of
the system, so that an old version of the architecture will point to the correct versions of the code, at
the corresponding moments in time.

The mapping solution adopted by ArchEvol is to not only maintain the architecture separate from the
implementation of the system, but also to split the implementation for the entire system into smaller
subsets of source code files that correspond to the implementation of individual components. The
mapping between the architecture and the implementation of the entire system will then be
translated into a set of links between each component in the architecture and its own
implementation files.

The maintenance of consistent versions of different software artifacts has traditionally been the focus
of Software Configuration Management (SCM) research. As the architecture of a system is itself a
model of the overall configuration of a system, maintaining the architecture of the system consistent
with the underlying implementation is a goal similar to that of SCM systems [Conradi and
Westfechtel 1999]. However, as SCM systems are not aware of the semantics of the software
artifacts that they manage, they cannot take specific actions needed to semantically link
components to their implementation. These links must therefore be managed outside the SCM
system, which is the problem that ArchEvol solves.

ArchEvol manages a three-way relationship, as shown in Figure 14, between the versions of the
components, the versions of the architecture, and the mappings between architecture and these
projects. The architectural description contains links to the implementation of the components. The
source code for these components is checked-in a SCM repository. The architecture is also
checked-in a SCM repository, and also keeps references to the repositories of the components.

ArchEvol can be used to create, in an atomic operation, new versions of all component
implementations and of the architecture at the same time. These versions will be linked to each
other so that, if the user wants to go back and inspect an older version of the architecture, the exact
implementation of the components in the architecture, as it existed at the time when the version was
created, is available for inspection too.
This versioning infrastructure is also used by ArchEvol to store the concern model along with the architecture of the system. In this way, the user can see not only how concerns are scattered and tangled over the system as the system is being developed, but also how the scattering and tangling have evolved as the system itself evolved. In this way, even if new concerns need to be identified later in the life of the system, the previous versions of both the architecture and the implementation are always available for a comprehensive analysis.

**Concern Evolution Visualization**

Capturing scattering and tangling information about concerns during evolution produces a three-dimensional “cube” of data. The dimensions of this cube are the components in the system, the versions of the system, and the concerns in each component and version. For each combination of the three, there are specific values for scattering and tangling that are recorded by ArchEvol.

In general, there are different scenarios by which the scattering and tangling of concerns can evolve over time. Developers that have this data available can use it to analyze two important aspects of the evolution:

1. What was the evolution of a concern in a particular component? For example, developers

Figure 14. Overview of the architecture and source code evolution support.
may be interested in the component that had the most variations of scattering and tangling over time for a given concern. High variations of scattering and tangling over time can indicate an unstable design in the component, and could potentially be related to an increased number of bugs for that component. Reversely, developers could be interested in components that did not change in scattering and tangling, which could indicate that the component was not changed much.

2. What was the evolution of a concern over the components in the system? For example, developers might want to see if the components that have the highest levels of scattering and tangling in the current version of the architecture always had similar high levels of scattering and tangling. If this is the case, it might mean that those components need more attention from developers in relation to their design quality.

The visualization that we propose is exemplified in Figure 15, which shows the evolution of two concerns, “c1” and “c2”, in a system with two components, that has an evolution history of four versions. The evolution of the concerns over the components is shown on each row, while the evolution of concerns over the entire system at a particular version is shown on each column. The values for concern metrics form the third dimension, because there can be multiple concerns implemented in every component and version combination.

By showing the concern metrics values in this format, the user can identify: (1) which concerns are implemented in which concerns and at what version, (2) the evolution of a concern metric over time, and (3) how this evolution compares with the evolution of other concerns. In Figure 15, we use some generic percentages to illustrate these scenarios. In the figure, it can be observed that the concern “c1” was implemented mainly in component 1, but, starting with the version 4, it was implemented in component 2 too. The evolution of the concern “c1” was monotone, with very similar values for all versions, with values for the metric visualized at 85% in version 1 and 90% in version 4. By contrast, the evolution of the concern “c2” in component 2 is characterized by considerable variation: a low 15% in version 1, then a sudden jump to 75% in version 2, then again low at 20% in version 3 and
The purpose of maintaining the evolution of the architecture, component implementations, and levels of scattering and tangling of concerns over time is to present the developer with new data that would not have been otherwise accessible or which would have been very difficult to obtain. Continuously maintaining them in ArchEvol makes it possible to offer this information to developers at all times, with the intention that if developers are aware of the design qualities of their system, and especially how they have evolved, they will take action to actively maintain it.

![Figure 15. Visualization of evolution of concerns.](image)
5. ArchEvol

A proof-of-concept prototype implementation of the approach described in the previous chapter has been implemented as the ArchEvol development environment. Based on the Eclipse development platform, ArchEvol supports concern-based development by making concerns explicit, by facilitating the maintenance of concerns throughout development, and by providing visualizations meant to increase a developer’s awareness about the overall design quality of the system being developed and about the evolution of this design quality over time.

An overall screenshot of ArchEvol can be seen in Figure 16. As explained in the approach description, ArchEvol promotes a dual-view of the software system: the software system being developed is represented as both source code and architecture. In the screenshot, this can be seen through the two main editors in the center: the source code editor on the left, and the architectural editor on the right. Although they are used in different ways, and serve different purposes, these two editors are meant to be visible together at all times. The source code editor is used frequently, while changes to the system are being implemented. In contrast, the architectural editor is modified directly less frequently, but offers a view of the status of the design of the system in a way that is always readily available to developers.

This dual-editor view of software development is suitable for a development setting that includes two monitors. In such a setup, one monitor can be actively used to implement changes in source code, while the other monitor can be dedicated to making developers aware of the status of the system’s design. In this way, developers will be able to refer to a continuously updating representation of the design of the system at the same time that they change the code.
Figure 16. ArchEvol.
These two views of the software system are bound together by the concerns that govern both the implementation and the design of the system. The overall set of concerns that are used in the system’s development is shown in two available concern model views. In one of the views, targeted at code exploration, the concerns are organized in a concern tree model, just as the files in the implementation are organized into packages and projects, and the architecture is organized into components. In the other view, whose purpose is to control the visualization of concerns, the concerns are shown in a more compact form. This view is used to associate a color to each concern and to select which concerns should be shown in the visualization. In the screenshot from Figure 16, the concern model views are shown on the right side, with the Concern Visualization View on top of the Concern Exploration View.

The concerns are visualized in both the source code editor and the architectural diagram editor. In source code, concerns are visualized by: (1) highlighting the source code fragments that are related to them in the editor, and (2) providing a separate view of the overlapping concerns next to the editor. At the architectural level, high-level aggregate metrics about the implementation of concerns are shown, also using color-coding. In this way, the developer is continuously aware of not only where concerns are implemented in a single source code file, but also where concerns are implemented at a system-wide level. By using the same colors for concerns across all visualizations, consistency is achieved and the interpretation of the visualizations facilitated.

The ArchEvol development environment is implemented as a set of extensions to the Eclipse platform [Eclipse Foundation 2004]. Eclipse is not only an open-source development environment, but also a platform that allows for development of specialized development tools, and for their integration into the overall environment. By providing the basic underlying functionality needed by a development environment, Eclipse allowed us to focus our tools’ development on adding specific functionality related to concern-based development.

The rest of the chapter describes the details of ArchEvol’s implementation, explaining the challenges faced during ArchEvol’s development and the choices made that address these challenges. Since
Eclipse is the underlying platform upon which ArchEvol was built, and a critical component of ArchEvol, we start by providing an overview of the main concepts in Eclipse. We then describe ArchEvol’s architectural organization, followed by detailed descriptions of each of its main functional elements. We end with a discussion of the main challenges and threats to ArchEvol’s transition from a prototype to a usable environment on a large scale, and present a list of future enhancements.

5.1. Eclipse

Eclipse is an open-source development environment platform. Eclipse’s development is overseen and maintained by the Eclipse Foundation, an open source community initially supported by IBM, but which now includes a large number of organizations. Its open source nature, its rich support for coding, and emphasis on extensibility are some of the reasons for Eclipse becoming one of the most popular development environments today [Geer 2005, Goth 2005].

The main feature that makes Eclipse different from other development environments is its emphasis on extensibility. Eclipse was designed to be extensible through special components called plug-ins, each plug-in being an encapsulated, deployable contribution to the Eclipse environment. The functionality inside a plug-in is described in its manifest file (written in XML), which specifies both how the plug-in uses existing functionality from other plug-ins, and how the plug-in allows other, future plug-ins to use its functionality. This extensible architecture transforms Eclipse from being just a development environment to being a development environment platform, to which specialized development tools can easily be integrated.

The core of the Eclipse project consists of the infrastructure necessary for creating a basic development environment, including support for accessing and organizing file system resources, basic user interface elements such as widgets and menus, and simple text editors. This functionality can be further extended by other plug-ins, and the Eclipse Foundation offers a number of such plug-ins as open-source projects as well. As Eclipse’s implementation is written in the Java language, one of the major extensions to the Eclipse platform is the Java Development Tools (JDT) project, which
includes comprehensive support for Java programming.

The Eclipse development environment is centered around two key concepts: the *workspace* and the *workbench*. The workspace represents the collection of resources available to Eclipse. In Eclipse, resources are organized into individual *projects*, which represent arbitrary units of development. Each project is a general container for source code implementation files. Java projects are extensions to a regular project that add individual compilation and runtime settings. A project can contain operating system directories, which are interpreted as packages in Java, and implementation files, which are continuously compiled after each change.

The workbench contains the set of Eclipse visual tools loaded in the environment. Eclipse uses two main types of visual elements: editors, which are used to actively change the content of a resource, and views, which show properties of resources and other useful information. A typical screenshot of Eclipse is shown in Figure 17. The default layout contains editors occupying the center of the environment (see (1) in the figure), and different views that show information about the code around them (see (2.a) and (2.b)). A set of views and their associated layout can be grouped together in a *perspective*. Users can further change and customize these perspectives to fit their needs, by adding views to the set or removing views from it.

The platform that Eclipse offers forms a key foundation for the development of ArchEvol. There are three main benefits of choosing Eclipse as a starting point. First, the Eclipse platform provides the basis functionality needed to develop an environment, allowing ArchEvol’s development to focus on only specialized functionality. Second, it already includes comprehensive Java development support, which allows us to focus on the architectural and concern visualization aspects of the environment. A third benefit, due to the extensible implementation of its editors, allowed ArchEvol to tightly integrate its functionality within the source code editor. With this existing support, ArchEvol can integrate source code development with architectural development and introduce concern awareness throughout a system’s development lifecycle.
5.2. Internal Architecture

ArchEvol was implemented as a set of six Eclipse plug-ins. Conceptually, these plug-ins can be grouped in two distinct groups, one offering supporting functionality that is reusable beyond ArchEvol, and one implementing ArchEvol specific features. Figure 18 shows an overview of these plug-ins and their dependencies. As the Eclipse platform provides the foundation for the implementation of all these plug-ins, it is not shown in this diagram. Instead, we show, on the bottom layer, a set of four significant Eclipse plug-ins used by ArchEvol in its development. The middle layer shows the ArchEvol plug-ins that implement the supporting features. The top layer contains the three plug-ins where the main logic of ArchEvol is implemented.

5.2.1. Foundation

The development of ArchEvol builds upon a set of open-source projects that provide development environment features beyond the basic functionality included in the Eclipse platform alone. ArchEvol
depends on these projects to create a comprehensive development environment. These projects are, in order of importance: Java Development Tools (JDT) [Eclipse Foundation], Subclipse [CollabNet 2004], Graphical Modeling Framework (GMF) [Eclipse Foundation] and Eclipse Modeling Framework (EMF) [Eclipse Foundation].

The most significant of the projects maintained by the Eclipse Foundation, besides the Eclipse platform, is the Java Development Tools (JDT) project. JDT is responsible for extending Eclipse with support for development of applications written in the Java language. As Eclipse is written entirely in Java, the development of JDT was instrumental in the development of Eclipse itself.

Among the most relevant features included in JDT are a textual editor for Java source code files, support for the organization of Java source code into Java projects, support for extensions of regular projects with specific functionality for Java, and a number of views and wizards that are helpful during development. Some of these wizards allow developers to navigate from locations in the source code where a method is used to the location where it is declared, and reversely, search for all places where a method (or variable) is used. Others allow developers to easily refactor the source code. Yet others allow developers to explore the inheritance hierarchies between Java classes.

A second project that ArchEvol depends on is Subclipse, which is maintained and developed by the
open-source Tigris.org community, and hosted by CollabNet [CollabNet]. The Subclipse project has the goal of integrating the services provided by Subversion, a version management system, with Eclipse. Being an open-source project results in Subclipse also providing a Java API that other Eclipse plug-ins can use to manipulate Subversion repositories programmatically.

The Graphical Editing Framework (GEF) is an Eclipse Foundation plug-in that has the purpose of making development of graphical editors easier. GEF is more than just a graphical library (in this case being the Eclipse draw2d graphical library), because it provides support for implementing graphical editors that follow a Model-View-Controller organization. The use of GMF was significant in ArchEvol’s architectural diagram editor implementation.

The Eclipse Modeling Framework (EMF) project was instrumental in providing an object-oriented interface to the xADL2.0 architectural description language. ArchEvol uses xADL2.0 description for storing all its data, and EMF facilitated the serialization and deserialization of this data in a way that is easily accessible from a Java program.

5.2.2. Infrastructure

The middle layer contains three subprojects that create the supporting infrastructure in Eclipse. These projects were created for ArchEvol, but during the design of ArchEvol they emerged as self-contained, general projects. Their generality makes them suitable for reuse in other possible software development projects that extend Eclipse.

The Eclipse Plug-in for Architecture Development (epad) plug-in enhances the functionality of Eclipse with support for architectural development. It does so by extending regular Eclipse projects to become specialized architectural or component projects. An architectural project is responsible for storing the architectural description of a system, while the implementation for each component in the architecture is implemented in its own, separate component project. These categorizations are necessary in order for Eclipse to support architectural development by helping developers manage a number of different implementation projects that are all related to the same overarching architectural description.
The Eclipse Context (econtext) plug-in was created as a general extension to the Eclipse environment where the developer can record, and refer to during development, descriptions of the larger context of development. By context we understand here a series of properties that describe multiple, consecutive development activities. In Eclipse, by default, any change to the source code that is followed by a save of a file is considered to be a separate, independent activity. However, in general, a series of consecutive changes to the code are all related to each other. For example, developers fixing a bug will need to change multiple files to implement the fix for the bug. During this time, all the individual changes to the code are related to the same bug description. The econtext plug-in provides an Eclipse view where the contextual properties of the tasks can be shown, and methods for changing these properties either programmatically or manually.

The Model-Based Graphical Editor Framework (mbgef) project takes the GEF project further in that it automates graphical editor interaction and serialization to an EMF model. The mbgef project was created as the general, reusable part of the architectural editor implementation in ArchEvol. Therefore, it instantiates the Model from the MVC support from GEF to a xADL2.0 model, it instantiates the View to a boxes-and-arrows diagram, and automates the Controller to synchronize properties between the Model and the View.

5.2.3. Core

The core plug-ins of ArchEvol are directly focused on supporting concern-based development. These plug-ins are responsible for integrating the services of the other, underlying plug-ins and on implementing new functionality that implements the vision for a concern-centric environment as it was laid out in our approach description from Section 4.

The Architecture and Code Consistency Management (accm) plug-in is responsible for maintaining consistent mappings between an architecture, its code, and the concern model throughout the development lifetime of a software project. In particular, the plug-in uses functionality in Subclipse to create an ArchEvol-specific versioning repository where versions of architecture, code, and concerns are saved. It also uses the epad project to maintain mappings between the three types of
information, continuously while the developer actively changes any of them.

The Concern-Oriented DEvelopment (code) plug-in is responsible for introducing concerns as first-class entities in software development. It creates views through which developers can manage concerns, it extends the Eclipse text editors’ functionality to visualize concerns in code, and it extends the mbgef plug-in with concern visualization in the architectural view.

The ArchEvol (archevol) plug-in is the overarching piece of functionality that integrates all other plug-ins together in order to form a comprehensive software development platform that supports concern-based development. The implementation of the archevol plug-in is responsible for instantiating the general services provided by the other plug-ins and coordinating the data flow of information within the environment.

5.3. Features

The main features of the ArchEvol environment mirror the features described in Section 4. To these features, we add two other supporting features, without which ArchEvol would not have been possible to implement. We present next the design and implementation details for all of the features in ArchEvol, starting with the supporting features.

5.3.1. Development Context

The Development Context feature is, by comparison with the other features from ArchEvol, of lesser importance. However, it is significant because it requires a shift that ArchEvol imposes on the way developers use Eclipse, and points out a shortcoming in the current assumptions that Eclipse has about development, which leads to Eclipse being not entirely suitable for ArchEvol. The development context feature overcomes this shortcoming.

Eclipse has a resource-oriented view of software development. During development, the developer is, in essence, modifying resources from individual projects in the workspace. When the modified resources are saved, the workspace is said to be in a consistent state. From this perspective,
software development can be seen as a series of actions that change and then save a resource. From the point of view of Eclipse, these actions are unrelated to each other.

The shortcoming of Eclipse lies in its inability to define a context of development where multiple changes are conceptually related to each other. There is no suitable place in Eclipse that can be used to describe properties of a larger task that relates a series of activities. This problem became apparent while we were designing ArchEvol and the Development Context feature was created as a solution to it.

The Development Context feature in ArchEvol consists of an Eclipse view, where the properties that define the context of a given development “session” are shown, and an API that can be used to change these values. The main purpose of this feature is to provide a way of making developers aware of the broader development context, and to create a unique place in Eclipse where multiple tools can use this information. The context view can be seen in the screenshot of ArchEvol in Figure 16 in the top-left corner.

ArchEvol uses two such properties for the development context: the current architecture and the current set of concerns. The architecture is shown in this window to remind developers which system are they currently developing. Although we only discuss the use of ArchEvol in the development of one system in this dissertation, it is actually possible in ArchEvol to have multiple projects that are part of different systems in the same Eclipse workspace. The development context is the place where developers can specify which of these systems is important for the task at hand.

The second context parameter that ArchEvol uses is the current set of concerns that the developer is addressing in the current task. While the system architecture context has a longer time span, a task is a smaller grained event, but still greater than a single resource change. In ArchEvol, 

1 The Mylyn project [Mylyn] was developed in parallel with ArchEvol, and includes support for task management. Mylyn lets developers declare their tasks (a list of tasks can be created or imported from issue or bug tracking repositories), with the purpose of monitoring the developer and learning the elements of the environment that form the context of that task. Mylyn is now one of the standard projects in Eclipse.
developers specify which concerns they are addressing in the current task. This information is set manually by the developer, and expected to be changed when the task that the developers work on changes.

5.3.2. Architectural Development Support

As we discussed in our approach, one of the goals of ArchEvol is to enable the use of software architecture as a long-term representation of the design of a system, supporting our vision for side-by-side development of code and architecture. In order to do so, we need to implement a mechanism for linking the architectural description of the system to the source code implementation. This consists of two distinct steps: (1) determining the appropriate organization of the Eclipse Workspace, (2) specifying links in the architectural description to the implementation of components, and (3) supporting the use of these links by the user.

**Eclipse Workspace Structure**

Eclipse inherently supports source code development and lets the user organize source code into multiple projects. The main decision in our case concerned where the architectural description of the system should be stored in ArchEvol and how to determine which resources in the Eclipse Workspace are linked to which components in that architecture.

One option considered was to have both the architecture and the implementation for the entire system in a single Eclipse project. However, this solution would impose unnecessary constraints on the implementation of software systems developed using ArchEvol. It would limit the ability of different developers working on the implementation separately from the architecture, it would prohibit splitting the implementation of the system into multiple projects, and it would complicate the deployment of the source code for each component individually. Furthermore, under such a scenario, individual source code files would have to be individually assigned to components in the architecture. This would require the maintenance of a potentially large number of files, and dealing with situations where new links have to be introduced, existing links have to be removed, or links have to be changed from one component to another.
In ArchEvol, we opted for a different solution: encapsulate the implementation of each component in a system in its own Eclipse project, and store the architectural description is a separate project. The observation that guided the solution adopted by ArchEvol was that, from a practical perspective, there are a number of similarities between the concept of a component and the concept of an Eclipse project. They are both containers for a part of the functionality of the entire system, they can both be of arbitrary size, and they can have dependencies to other elements. Eclipse projects can already be developed separately from other projects in the system, and can be deployed individually.

By choosing an Eclipse project as the container for the implementation of an architectural component, the links between individual files and the architecture will be implicit: if a file is added to the project, it is considered implicitly as being a part of its linked component’s implementation. Removing a file will destroy this link, and moving files between the projects will result in changing the link to a different component.

The implementation of ArchEvol uses Eclipse project “natures” to specialize regular projects into Architectural Projects and Component Projects. Architectural Projects contain the description of an architecture, while Component Projects contain the implementation of a single component in an architecture. While most of the code in ArchEvol does not require the implementation of the components to be in any special language, the Java language is supported as a default option. Therefore, the Component Projects will usually also be Java projects.

**Links between architecture and code**

In ArchEvol, the architecture of a system is described using the xADL2.0 language [Dashofy et al. 2001]. xADL2.0 is an extensible architectural description language, based on XML, which allows various architecture-specific tools to interact with each other by sharing the same type of architectural description. The core definition of the xADL2.0 language provides a basic vocabulary of architectural concepts that other tools can extend through XML Schemas [Costello NA].

The basic architectural constructs supported by xADL2.0 are components, connectors, and interfaces. The difference between components and connectors is semantic, and from the point of
view of ArchEvol there is no discernable difference between a component and connector. Furthermore, connectors may often not have an implementation at all, if components communicate through implicit mechanisms provided by the underlying operating system or virtual machine, such as method calls or remote procedure calls (RPCs) [Mehta et al. 2000]. As a consequence of their main purpose, which is to move the data between the component that produces it and the components that consume it, it can be expected that the same concern implementation or at most a small subset of implementation, is shared by multiple connector instances. Even though in ArchEvol we focus on the links between components and their source code implementation, the same links can be used to for connectors to their implementation.

Interfaces are used by components and connectors to specify the types of services that they provide or consume. In an architecture, component and connectors are linked to each other through these interfaces. As these interfaces do not require a source code implementation container, they can be ignored for the purposes of ArchEvol.

As ArchEvol does not enforce any type of conceptual consistency between the architectural description and the source code other than the correspondence between components and their implementation, the semantics of the differences between a component and a connector and the semantics of an architectural interface are out of the scope of ArchEvol. However, the mappings provided and maintained by ArchEvol could be used by other applications to prescribe and enforce these semantics. For example, such an application could include consistency critics that would check whether a component behaves like a component or like a connector, or that the architectural interfaces have corresponding implementations as interfaces in code.

The set of core schemas for xADL2.0 include an implementation specification for components (and connectors) in the JavaSourceCodeManager element\(^1\). Figure 19 shows the XML schema for this

\(^1\)\(\)xADL2.0 uses the concept of a “component type" to capture a reusable description of a component. In xADL2.0, the implementation of a component resides in a “component type" rather than in a “component" description. However, throughout this dissertation, we do not highlight this syntactic difference in order to avoid confusion.
The extensions added by ArchEvol specify an EclipseSourceCodeManager as a specialization of this element, which stores a name and an id of the Eclipse project that provides the implementation for the component. The identifier is needed because, in Eclipse, project names are volatile in the sense that the same project can be loaded in the workspace with a different name. ArchEvol therefore needs a mechanism that identifies the content of the project, and as a result stores a unique project id in a special file inside each project. At the same time, the name helps differentiate when two projects that have the same id are loaded in the workspace. For instance, two different copies of the same project made at different times would have the same id, but different names.

The fact that an architecture described in xADL2.0 can be shared by multiple specialized tools allows ArchEvol to integrate easily with other tools that use the same description for architecture. For instance, the ArchStudio [Dashofy et al. 2007] environment consists of a specialized set of tools that can edit, check for consistency, and instantiate architectural descriptions written in the xADL2.0 language. The functionality provided by ArchEvol is complementary to the functionality implemented in ArchStudio.

**Architectural Editors and Views**

ArchEvol provides support for architectural development in the form of an architectural editor and associated views. Each architecture is encapsulated in its own Architecture Project, which is a regular Eclipse project containing one file, named “architecture.xml”, that contains the architectural
description. The graphical editor and the views are used to modify these architecture description files.

Figure 20 shows a screenshot of the architectural editor and views in ArchEvol. The architectural editor, shown in the middle (see (1)), is a boxes-and-arrows graphical editor for a xADL2.0 description. Components and connectors are drawn as large rectangular shapes, with interfaces drawn as smaller rectangles placed on the edges. Any two interfaces can be connected through links. The graphical editor has been implemented using the Eclipse draw2d graphical library and the Eclipse GEF. Basic editor features such as the ability to move components around, to re-attach links to new interfaces, the ability to zoom in, ability to save the diagram to an image, etc., are provided by the GEF framework.

Properties of the graphical elements in the diagram are automatically saved to the architectural description, and are also updated any time the architectural description changes. These properties include the names of components, links between interfaces, positions on the diagram, and, as we
will discuss later in this chapter, concern-based information. This generic framework for architecture graphical editors has been packaged as the Model-Based Graphical Editing Framework (mbgef) sub-project of ArchEvol and can be reused in other environments that use a custom architectural editor.

Three other views help the developer visualize the data in the architectural description that is being edited. An outline of the architectural description is presented by the Architecture Explorer view (see (2)). This view shows a subset of the xADL2.0 elements that are relevant to ArchEvol, namely components and connectors, their types, implementation and repository information, interfaces and links. The Implementation Outline View (see (3) in the figure) shows, for each component in the architecture, the status of its implementation mapping. The status of the versioning information for each component is shown in the Versioning Outline View (see (4) in the figure). We will discuss the versioning extension to xADL2.0 that ArchEvol adds, and the checks that ArchEvol performs below in Section 5.3.5.

The architectural properties that these views show need to be constantly updated as the content of the workspace changes. For example, if a component has an implementation project in the workspace, and the user closes or removes that project, then both the Implementation Outline View and the Versioning Outline View need to be updated to show that the link between the component and the implementation has been severed. In order to do so, ArchEvol actively listens to changes in the Eclipse workspace, through a set of APIs provided by the Eclipse framework, and monitors all architectural descriptions from all architectural projects opened in the workspace. Then, for all components in all these descriptions, it matches their description to existing projects opened in the Eclipse workspace, adds new links once new projects are opened and removes links when projects are closed. ArchEvol maintains, in this way, a “live” set of associations between components and the files that form their implementation.

5.3.3. Concern Model

The concern model is a central component of ArchEvol. The purpose of the concern model is to
maintain the hierarchy of concerns and their sets of source code fragment links. The idea is that users populate the model with new concerns and new source code links, and rely on the model to explore the code base through the concerns. They are supported by the various visualizations, which also build upon the concern model.

ArchEvol stores the concern model in the architectural description of the system being developed. The concern model is similar to the architectural description in that it contains information that is relevant for the entire system, as any concern in the concern model could potentially be implemented in any component in the system. ArchEvol defines an extension to the xADL2.0 description language to support the definition of the concern model.

As shown in Figure 21, a concern is described by a name, a textual description that contains a more detailed explanation for the concern, and a color that will be used for visualization. The concern can have optional sub-concerns, which allow for the hierarchical organization. A list of SourceCodeRef elements represents pointers to the implementation files that are linked by the user to the concerns. The links to the implementation files contain: (1) the project name and id, (2) the file path of the source code file inside the project, and (3) the text interval inside the file, stored as start and end offsets. A copy of the latest snapshot of the text referenced is stored for presenting the user with a
Model maintenance

During software development, concerns are continuously changed. The users can add or modify new concerns, delete existing concerns, and add, modify or delete the source code references that are linked to the concerns. These changes, managed by the concern model, need to be reflected back to the user through visualizations. The visualizations are updated in the Eclipse editors and views that are in use. The changes in these editors and views will result in new changes to the concern model. Further code modifications based on the visualizations begin a new iterative cycle, as shown in Figure 22.

During development, the source code references need to be maintained accurately by ArchEvol. ArchEvol monitors two types of events (as shown in Figure 22), workspace events and editor events, and changes the concern model accordingly when any of these events is triggered. For the first type of events, ArchEvol uses the Eclipse Workspace API to listen to changes to any of the projects that are open. When files are moved and renamed, the concern model is updated to point to the new project and the new file path.

For the editor events, a different type of event listener, using the Workbench API, is used to detect

Figure 22. The internal architecture of concern event architecture in ArchEvol.
text changes inside opened files. A listener is registered with every opened editor, and every text
modification is processed by the concern model to update the start and end positions of all the
source code fragments pertaining to the files being changed.

The algorithm by which the source code fragment positions are updated in ArchEvol takes as input a
text change. Every change to the text is processed as two sequential operations: a deletion followed
by an addition of text. When a text fragment is replaced with a new text, then both the deletion (of
the old text) and the addition (of the new text) have a length greater than zero. Each deletion or
addition is evaluated, in turn, against the existing set of source code fragment links for each
concern, one by one. The algorithm by which these actions are processed is:

1. If the operation did not “touch” any existing source code fragment, then
   1.1. all the source code fragment links that start before the operation offset are left in
       place, while
   1.2. all the links that start after the start offset are moved either forward (for an insertion)
       or backward (for a deletion) by the length of the modified text.

2. If the operation touches existing fragments, then the fragments are extended (for insertion)
   or truncated (for deletion).

3. If, after the modification, two source code fragment links become adjacent, they are merged
together into one link.

4. If, after the modification, there are source code fragment links of size zero, they are removed
   from the model.

When text is added inside an existing fragment already belonging to one concern, ArchEvol can
theoretically take two actions: (1) either to include the new text as a part of the existing fragment, or
(2) to split the fragment in two parts and leave the new text out. There are, however, also two
possible scenarios with regard to the actual intention of the user. The user can insert some text that
belongs to the same concern as the initial fragment, in which case the correct decision for ArchEvol
would be to extend the fragment to include this new text inside, or the user can insert text that
belongs to a different concern, in which case the correct action for ArchEvol would be to split the fragment into two smaller ones and add the inserted text to a new concern. As any of these two scenarios is possible, any of the two actions that ArchEvol can take would work as intended in some situations, and not in other.

ArchEvol supports by default the first option, because it is assumed that developers will try to write code to be less scattered and tangled, and therefore will try to add new code next to fragments that address the same concern. This behavior should be particularly encouraged by the visualizations that ArchEvol provides. However, a configuration parameter is provided to allow the user to change this default behavior to the second option.

Throughout these text modifications, ArchEvol strives to maintain fragments as compactly as possible. This is helped by the fact that fragments are expanded if text is inserted inside their boundaries, are removed if they are empty, and are joined with other fragments (belonging to the same concern) if they are adjacent.

**Concern Model Views**

Figure 23 shows a screenshot of the concern-related views of ArchEvol. The concern model is shown in two distinct views, each one emphasizing the use of the concern model for a different purpose.

The view on the far right is the Concern Tree View (see (1) in the figure), which organizes concerns hierarchically. In this view, the data is structured as a tree-view, in a similar fashion to two other commonly used views in Eclipse, (e.g., the Resource Navigator and the Package Explorer), which makes it look familiar to Eclipse users. Each concern has as its children either sub-concerns, or source code references that are grouped by project and with each project per file. The source code references show the beginning of the text fragment that they link to, in order to help the users differentiate them, and are constantly updated as the text changes in the editors. This type of view is appropriate for the exploration of the code through concerns. Users can easily explore all the places where a concern is implemented by expanding its concern sub-tree. Clicking on a source code
reference will bring the related file editor in focus, and highlight the referenced text interval.

While the tree view of the concern model is good for the exploration of code, it is not suitable for quick visualization of concerns. The large number of source code references to which a concern might link would typically make the tree exceed the visible area of the view, which leads to having the other concerns in the concern model hidden from the user’s view.

For overview purposes, we need a view that can show concerns and their colors only. ArchEvol uses color-coding to visualize concerns in both code and architecture. Therefore, a map between the color used in the various visualizations and the actual concern that the color represents is needed so that the user can determine which color belongs to which concern. The second concern view, the Concern Visualization View (see (2) in Figure 23), simply lists the concerns and their associated colors. References to a selected concern are shown as a list in the bottom part of the view that updates when the selected concern changes.

Adding New References

The user can choose either context recording or manual selection to associate source code fragments to concerns from the concern model. Context recording uses the Development Context View, discussed in Section 5.3.1, to let the developer choose the list of the concerns that are addressed by a given task. Any additions to the text of the editors, if there are any context concerns, will be automatically associated with these concerns. In this way, the user will only have to change the current context concerns when a new task begins. This behavior is useful when, for example, the developer enters a number of distinct source code fragments, in different files, for the same concern. In contrast, if a developer adds fragments for a number of smaller concerns, the automatic recording would require that the user change the list of context concerns too often. As an alternative to automatic recording, ArchEvol provides the option of manual selection.

The manual selection option uses context menus that allow the user to: (1) associate a selected text fragment with a concern, in which case the concern model will add the selected text as a source code fragment under that concern, or (2) to disassociate the selected text fragment from a concern.
The option to disassociate a text fragment is valid only if the concern already has one or more source code fragments that overlap the selected text. In this case, the selected text is removed from all fragment links. Manual selection is complementary to automatic recording in that it is more useful when small fragments that address different concerns are added at the same time, but it requires an extra effort from the developer (selecting the menu from the editor’s popup menu list) if multiple fragments for the same concern are added.

Concern fragment links are also modified after each time the text of the current editor in Eclipse changes. If the text falls inside or next to an existing fragment link, this fragment is expanded to include the new text. This is not a method of adding new references, since only existing fragments can be modified or deleted. Deletion of fragments occurs because, after any modification to the concern model, the list of source code fragment links for all of the concerns is compacted by merging adjacent links and removing links to empty fragments (of zero length).

Figure 23. Concern views in ArchEvol.
5.3.4. Concern Visualization

ArchEvol provides a series of visualizations that help the developer in maintaining concerns and in analyzing the system based on concerns. These visualizations have the purpose of making the developer aware of the concerns as implemented in the system, which is an important factor when deciding the design and implementation choices for a change to the system.

As we have discussed, our visualization mechanisms use color-coding to distinguish between different concerns. The color for each concern is manually set in the Concern Model View. This color is used to show concern information in two places: the source code text editor and the architectural diagram.

Concern Visualization in Code

The primary purpose for the visualizations in code is to make developers aware of where the code related to one or more concerns is implemented. One of the options considered while designing ArchEvol was to highlight the source code based on concerns. While ArchEvol implements this functionality, it has some limitations.

First, it is not suitable for showing overlapping concerns, where the same line of code can belong to two or more different concerns. When a line is highlighted, then only one of these concerns will be visible. A possible alternative of showing the overlapping fragments with a different color is even more confusing, as this color will not be shown in the concern-to-colors map.

Second, showing all concerns highlighted in a file might become overwhelming for the user, since in practice we expect that all the text in a source code file to be associated to one concern or another.

Third, color coding adds a new visual element to an already loaded source code editor, where source code syntax is color-coded, compilation errors and warnings are shown as annotations, and debugging breakpoints are commonly used. Adding multiple concern annotations with different colors would likely conflict with some of these colors, and introduce confusion.

A fourth reason against code highlighting is an Eclipse technical limitation. As of the 3.2 version of Eclipse, the existing facilities for editors lack support for multiple dynamic annotation types.
The visualization that we chose overcomes all of these problems. Particularly, we implemented the Concern Overview, shown in Figure 23 to the left of the editor (see (3) in the figure), and in more detail in Figure 24. The Concern Overview shows, as the name implies, an overview of the organization of concerns in code at the granularity of a line of text.

The visualization in the Concern Overview view is organized in lines, with each line showing which concerns are implemented in the corresponding line of code in the editor. For a line of text in the editor, if a concern contains a source code reference that intersects this line, then a small colored block is placed in the Concern Overview. The color of the block is the same as the color for the concerns, chosen in the Concern Visualization View. The lines in the view have the same height as the lines of text in the code editor. When the user scrolls through the editor, the view is automatically updated to show the corresponding information for each visible line.

This Concern Overview makes scattering and tangling of concerns easy to visualize at the level of detail of a single file. A concern that is scattered through the code will have multiple, separate “columns” formed of one or more colored cells. On the level of a single line of code, the presence of multiple cells on the same “row” in the Concern Overview indicates that some concerns are tangled with each other in a single line of code. Even more, the user can be aware of these properties of concerns while in the process of writing the code, because the visualization is unobtrusive and always available next to the editor.

By using the Concern Overview visualization to show tangling and scattering of concerns, we can use highlighting of the text in the editor for a different purpose, which is to help the user find the fragments related to one or more concerns of interest. The user can select one or more concerns in the Concern Model View, and only the fragments from those concerns will be highlighted in the code. The text is highlighted with a neutral color that can also be selected by the user.

The Concern Overview visualization tolerates all of the previously mentioned requirements that text annotations have difficulty addressing. It shows overlapping of concerns by the presence of two different blocks next to each other on the same line. All the concerns in code can be shown since
this view is not superimposed on the text of the source code, and since it is a separate view, its colors do not coincide with other colors used for other purposes. Being a custom view also gets rid of the Eclipse limitation regarding multiple annotation types.

For example, in Figure 24, a fragment from the `load()` method in the `Buffer.java` file from the jEdit project is shown. Two concerns, the `buffer` and the `load` concerns are associated with every line of code in this method. The visualization shows that the `UI` concern overlaps these other two concerns in line 160 and the autosave concern overlaps the buffer and load concerns in two places: on line 174 and on lines 181 and beyond.

Since it cannot show the exact indices in the line of text where concern references start and end, this visualization cannot show precisely which characters in the line of code are associated with each concern. Therefore, while there might be a number of concerns shown in the view, they might not necessarily overlap in the text, even if they are on the same line. Therefore, users should be
careful when interpreting the data in the Concern Overview. In this situation, the user can choose to select one concern to be highlighted in the text, in order to see the exact boundaries of the code fragment. In Figure 24, the autosave concern is highlighted, and the visualization shows that the concern covers the entire text of the lines on which it is present.

**High-level Concern Metrics**

ArchEvol continuously calculates concern metrics to be shown at the architectural level in the architectural diagram. The architectural model, enhanced with the visualizations for high-level concern metrics is always present, throughout development. This peripheral data that is available at all times helps developers in making informed decisions about the future changes to the system.

The two metrics that are implemented are the metrics described in Section 4.1.3, but other metrics can be easily inserted into ArchEvol with the condition to extend a given API. The metrics that ArchEvol uses have two different inputs: one is the Eclipse workspace, and one is the concern model from ArchEvol. From the Eclipse workspace, these metrics need a count of the number of source code classes per component. From the concern model, the metrics need a count of the number of source code files with which each concern is associated, both as a total and on a per-concern basis.

The major implementation issue in the implementation of these measures is responsiveness. As the values for these measures need to be visible at all times, calculating them every time the concern model changes (which can potentially be every time a keystroke is pressed in an editor) is unfeasible. In order to cache these values when possible and update only when strictly necessary, ArchEvol uses a notification mechanism included in the Eclipse workspace to detect when files or their contents are changed. After the concern model is updated as per these changes, only the metrics strictly affected are re-calculated and the visualizations updated. in this way, ArchEvol ensures that the metric calculators refresh as little information as possible, which increases their responsiveness.

**Concern Visualization in Architecture**

ArchEvol uses the architectural diagram to show the concern metrics that it calculates. The
architectural diagram is created and maintained by the Architectural Development Support sub-project from ArchEvol, which was described in Section 5.3.2. The user selects which concerns to visualize through the Concern Visualization View, which was described at the beginning of this section. The same view is used to select which metric will be shown in the architectural diagram.

Figure 25 shows the visualization of concerns over the architecture in ArchEvol. The two main elements in this figure are the architectural diagram (see (1)) and the Concern Visualization View (see (2)). The presence of a concern in a component’s implementation is shown in the architectural diagram by a small box inside the component, using the same color as the concern’s color from the Visualization View. If the concern has no implementation in the component (and therefore its metric value is equal to zero), the component is shown as semi-transparent, which helps emphasizing the relevant components in the view.

The concern metrics have values represented as percentages. The small box for each concern is filled with the same value as the concern metric percentage, using the color of the concern. In this way, the user can gauge these values just by looking at the visualization.
Looking at Figure 25, we observe that three concerns are selected (*buffer, makers* and *encoding*). The "syntax" component has none of these concerns, and therefore is semi-transparent. Two components, “bufferio” and “options” address all three concerns, while the “buffer” component addresses only the *buffer* and *encoding* concerns, and the “print” component addresses only the *buffer* concern.

Since the view shows the Concern Relevance metric, we can also observe that the buffer concern is present in every file in three of the components, but it is associated with only a small percentage of files in the “options” component. We also observe that about half the classes in the “bufferio” component are associated with the *markers* concern, and that only a small percentage of classes in the “options” component are associated with that concern.

From the same visualization, it can be observed that the *buffer* concern is scattered around four components, and in three of them is scattered in almost all their enclosed classes (because of the high value of Concern Relevance metric). By comparison, the *encoding* concern is only scattered over two components, and in both of these components is not present in a majority of classes.

Overall, we conclude that these three concerns are tangled with each other, especially in the implementation of the bufferio and options components. However, because in the bufferio the values for the Concern Relevance metric are high, the concerns are more tangled than in the component “options”, where each concern covers only a small number of classes, and there is a possibility that they do not overlap.

The architectural visualization is rich in information about the concerns, their scattering and tangling, and yet shows this information in a compact form. The user will be able to use this visualization to obtain an informed view of the system by just inspecting the architectural view, and selecting the relevant concerns.

**Visualization of Evolution of Concerns**

The ability of ArchEvol to manage versions of the source code, architecture and concerns over time provides an opportunity to document the evolution of concerns as the system itself evolves.
Developers have the ability to check out older versions of the system for analysis and analyze them, because the source code, architecture, and concern model are saved by ArchEvol at each version. However, in order to understand the evolution of concerns, they need a dedicated view.

We designed the visualization shown in Figure 26 to allow the user to gather a quick overview of the evolution of concerns. The left side shows the architectural elements, while the columns represent the versions of the system stored in the ArchEvol repository. Each cell, therefore, contains the data for the specific implementation of a specific version of a specific component.

An important issue in creating a useful history visualization is scalability. The data that can be shown in each of these cells is calculated for each metric and for each concern in the concern model, and therefore, showing all of this data at the same time is impossible. In ArchEvol, as we already mentioned, the user selects which concerns to view in the History View through the Concern Visualization View, where both the visible concerns and the metric shown can be selected.

To ease the visualization, ArchEvol automatically sorts the list of components in the History View. The data in the view can be sorted by two criteria:

1. Components are ordered by the sum of the metric values for the current version of the system (the last one in the history list). This helps the user in observing the components that are more relevant with respect to the concerns selected, as they would appear first in the list.

2. Components are ordered by the highest variation in the metric values across versions. This helps in determining which components are unstable, as they could be the ones that are most active and require the most attention. These components will also show first in the list.

These two visualizations can be used with one or with multiple concerns selected. Figure 26 shows four scenarios of how these two visualizations can be used. In the first scenario, only the “markers”

\[\text{1 The jEdit study used to exemplify ArchEvol only identified concerns in one version of the project. The values shown in the History View are for illustrative purposes only and are not based on existing data.}\]
concern is selected. The components are sorted by the current version values, and therefore show the components “bufferio”, “menu”, “jeditapp”, and so on, in order of the value of the metric. These are the three most important components for the markers concern in the current version, even if in past versions these components did not address the concern (such as is the case with the “menu” and “jeditapp” components. The second scenario shows three concerns selected, namely markers, buffer and encoding. In this case, the order of the components changes, with the “bufferio” becoming the most relevant component because it addresses all three concerns.

The component “io” also addresses the three concerns, but it is shown lower in the list because the sum of the values for the metric is smaller than the values in the other components, even if, for instance, “print” only addresses the buffer concern.

The variation of the concern values can result in different relevance orderings for components based on which concerns are visualized. The third scenario from Figure 26 shows the results of selecting one concern, buffer, and choosing the variation sorting option. The “bufferio” and “textarea” are the components that have had values that varied the most in the previous versions. The “bufferio” had the most dramatic changes in values from the two. In the fourth scenario, both the markers and the buffer concerns are selected. In this case, the order of the components changes with “textarea” becoming first because of the high variation in the values of the markers concern.

5.3.5. Evolution Support

The code, the architectural model, and the concern model are three different representations of the same software system, and therefore are all conceptually linked together. The architectural model contains information about the components and their source code implementation, and the concern model contains information about source code fragment links to code. Since the concern model is stored in ArchEvol inside the architectural description, the only problem that remains to be solved is keeping the architectural description and the implementation of each of its components accurately linked to each other as the system evolves.
Figure 26. Visualization of concerns in the History View.

The visualizations show the evolution of the 1) markers, 2) markers, buffer and encoding, 3) buffer and 4) markers and buffer concerns. 1) and 2) are sorted by the current value of the metrics, while 3) and 4) are sorted by the variation.
ArchEvol implements this functionality on top of a Subversion [CollabNet 2003] versioning repository. The Subclipse [CollabNet 2004] open source project provides an Eclipse-integrated API and an Eclipse plug-in implementation that can be used to manage Subversion repositories from within Eclipse. ArchEvol uses this API to ensure that the Subversion repository is structured in a way that is specific to ArchEvol, and to implement a higher-level versioning API suitable for ArchEvol.

The basic functionality of adding an Eclipse project to a repository, checking-in a version of that project in an Eclipse workspace, and committing the changes to the repository is already provided in Eclipse. However, ArchEvol needs to manage, for each system, one architectural project and multiple component projects that can all be checked-in to different repositories. The identifier of each component project in these repositories (represented by an URL in Subversion) needs to be stored inside the architectural description from inside the architecture project.

ArchEvol uses a specific Subversion repository format and rules on how this repository can be used. Each project managed by ArchEvol is stored in its own repository directory, which is required to follow the Subversion generally accepted guidelines [Collins-Sussman et al.] for repositories. The current version of the project is stored in a subdirectory named “trunk” and the versions are each directories within a “tags” folder. The implicit rules about the use of these directories is that current, working versions of the architecture and its components use the trunk directories, while other versions that are kept for historical purposes are stored in the tags directories.

A version of a component project can be identified by three elements: the ArchEvol repository URL, the name of the component project inside the repository, and the version name. These three values are combined to form a unique URL of the architecture or the component project. In order to add this information to the architectural description, ArchEvol defined a specific extension to the xADL2.0 language, as shown in Figure 27.

As we discussed in Section 4.1.4, ArchEvol needs to maintain a three-way relationship between architecture, code, and the versioning repository. The architectural description is responsible for two of these relations: a component description from the architecture needs to have an implementation
project (denoted by the “project” attribute), and also needs to have an address of the repository where the component’s implementation is stored (denoted by the “repositoryPath” and “version” attributes). The third part of the relationship is due to the fact that the implementation project itself can be checked out from a Subversion repository, which is the information that the Subclipse plug-in maintains in Eclipse.

In order to enforce a consistent three-way relationship, ArchEvol checks whether the repository information from the component description in the architecture actually matches the address of the Subversion repository where this component is checked-in, and informs the user if any of these two values is missing or if they do not match. Furthermore, ArchEvol helps the user in setting the repository path and version from the URL of the implementation’s project, and if this is checked-in to an ArchEvol compliant repository. In some cases, it updates automatically these values once the project is checked-in by the user.

Once the links between all the component descriptions in the architectural description and their
implementation projects are set, all these projects are checked-in to an ArchEvol repository. During this step, the paths to their corresponding URLs are set back in the architectural description, which results in having the entire system state saved as a new version with ArchEvol. In order to do this, two conditions have to be fulfilled: all the component projects have to be checked out in the workspace, and their linked versions to the architecture have to be from the trunk. In this case, creating a new version of the system involves the following steps:

1. Create versions for each component;
2. Update the architectural file to link components to the new version information (in a temporary copy); and
3. Create a version of the architecture.

In this way, the links will be preserved, and it is possible to go back to a point in time and not only see the source code of the system at that point in time, but also examine the architecture of the system and analyze the concern information at that point in time. As shown in Figure 28, the architecture at version 1 will be linked to all the component implementation projects at version 1, the architecture at version 2 will be linked to the component implementations at version 2, and so on.

5.4. Implementation Challenges

One of the main challenges for the implementation of ArchEvol is the management of the large number of events that need to be processed without slowing down the regular activities of the developers. The concern measures are updated for every key stroke, as any change in the text of a file could potentially impact the position of source code fragment links that are referring to the same file. Once the source code fragment links are modified, the visualizations that are based on them need to be refreshed too.

ArchEvol addresses the issue of responsiveness by calculating each metric in a separate thread, which does not impact regular code editing activities. Since most of the searching of source code references related to a concern is performed one file at a time, the source code references for each
concern are indexed by file, which reduces the time necessary for an event to be processed. Other optimization strategies employed by ArchEvol are: (1) using events with the smallest level of granularity possible, (2) updating as little information as strictly necessary, and (3) caching data when possible.

Scalability is another key issue that ArchEvol needs to address. Scalability impacts the data in ArchEvol in multiple ways: the architecture could have a large number of components, the concern model might contain a large number of concerns, and these concerns may each involve a large number of fragments. Each of these scalability factors can impact the effectiveness of the visualizations in ArchEvol.

One of the main factors that impact scalability of the visualization is the physical limits of the monitors that developers use. Even if steps are taken in making visualizations useful as the system scales up in size, having limited resources to display the visualizations is still going to present a problem. This is one of the reasons for which ArchEvol advocates the use of a second display dedicated solely for showing the architectural visualizations, while coding-related editors and views are shown on the first monitor.

In the case of the size of the architecture, ArchEvol accommodates a large number of concerns by
offering two types of automatic layouts in the architectural editor, implemented with the use of the JUNG graph library [JUNG]. However, while generic graph algorithms are useful, they might not be fully satisfactory in all situations. Layouts of architectures are tightly tied to the semantics of those architectures. For instance, a layered architectural style requires that components be laid out in rows (or layers). In other architectural styles, developers might associate sections of the diagram with specific functionality and group related components together. In this situation, using an automatic layout could potentially create confusion by moving components around, which is undesirable. However, since these decisions are based on the particular details of each architecture being developed, specific layouts fall outside the focus of ArchEvol.

Other visualization mechanisms are used to minimize the effects of scalability. ArchEvol moves the focus of the architectural view to relevant concerns and draws the other components in a semi-transparent color. In this way, the attention of the user is drawn to only the important sections of the diagram. Similarly, in the History View, the most relevant concerns are shown at the top of the list, and the list is refreshed every time this ordering changes. As a result, even though the list of components might be large, developers will most likely not need to scroll down to see everything in the list, because the most interesting components are always going to show up at the beginning of the list.

To address the problem of a large number of concerns, ArchEvol uses the Concern Visualization View to let the user select a smaller number of concerns to be visualized in the architecture. Because of the graphical limitations imposed by the size of the components and the size of each concern metric visualization, only a relatively small number of concerns can be efficiently visualized simultaneously. However, the visualizations are equally effective if the user selects, repetitively, a fewer number of concerns instead of visualizing every concern at the same time. To understand whether a concern is scattered, for example, only that concern needs to be selected. Also, in order to understand if three concerns are tangled together, the user can either select all three concerns, or choose combinations of pairs of two concerns, and still obtain the same observations.
In code, the Concern Overview was created with the purpose of showing the presence of multiple concerns on the same line of code. This is a compact visualization that can accommodate a large number of concerns, and that can be used to obtain a quick overview of where all concerns are implemented inside the same file. Of course, this visualization has physical limitations to how many concerns it can show per line, but based on our study from Section 2.2, we expect that a regular development setting will not have at maximum four or five concerns overlapping on the same line.

The Concern Tree View groups the source code references by files and projects. This allows the user to explore a large number of source code fragments. Indexing the fragment links by file also makes processing the code change events in linear time regarding the number of the fragments.
6. Evaluation

ArchEvol was evaluated in three distinct experiments. These three experiments complement each other by targeting different aspects of ArchEvol. The first experiment is a user study, in which we observed a number of subjects while they used ArchEvol to engage in a simulated real-world development scenario. In the second experiment, two expert developers evaluated the effectiveness of the architectural visualizations on a system that they developed. The third experiment is an off-line simulation of the use of ArchEvol over a long time, in which the evolution of an existing open-source system is imported in ArchEvol as if ArchEvol were used throughout its development.

6.1. Evaluation Goals

ArchEvol’s prototype implementation demonstrates the feasibility of the approach. This proof-of-concept implementation shows that an environment such as the one described in our approach can indeed be built. However, this does not necessarily prove that ArchEvol could be used by different people or in a real-world environment, or that ArchEvol can provide useful information about a software system in addition to what developers already know.

Designing the proper evaluation for ArchEvol poses a set of unique constraints that are at odds with each other. One of these constraints is the time aspect of the experiment. ArchEvol is intended to be used throughout the development life of a software system. Throughout this development time, a large number of often small changes to code amount to changes at the architectural level. A short user experiment cannot include a sufficient number of small changes to be able to trigger a significant, higher-level effect.

Another aspect is the level of experience that developers need to have in order to use ArchEvol
efficiently. On the one hand, the concern model should help novice developers in discovering and exploring the system that they are developing. On the other hand, developers have to at least be familiar with the system’s design in order to reason about the architectural metrics and visualizations.

We mitigated these constraints by evaluating ArchEvol in three experiments, two user studies and an offline simulation, each with its own set of evaluation criteria. The purpose of the first experiment, the jEdit user study, is to simulate a real-world development scenario performed using ArchEvol. This study contributes three types of data. First, it provides insight into which types of activities developers perform and how developers switch between these activities while performing a task. This is relevant for ArchEvol because it shows how developers use ArchEvol and to what results. Second, the study provides usability data about ArchEvol, since it was used by different developers. The study was performed on three different development setups, one using ArchEvol and the other two using alternative development environments, which allows us to compare the effectiveness of ArchEvol to the other two development choices. One of these alternatives is the pure Eclipse environment, which acts as a baseline to which to compare the performance of ArchEvol. The other environment to which ArchEvol was compared is FEAT [Robillard and Murphy 2002]. The FEAT environment supports the use of Concern Graphs (sets of conceptually related code elements), organized in a concern model, to help developers in navigating and understanding the code. Although similar in purpose, the concern model in FEAT differs significantly in its approach by using relationships between Java elements instead of links to actual source code fragments as ArchEvol does. Relatively to the features of ArchEvol that were evaluated in this experiment, namely the concern model, the visualization of concerns in code, and the maintenance of concern mappings as the software evolves, FEAT is the closest tool in similarity to ArchEvol, and therefore was important to include in the evaluation.

The study’s tasks consist of developers performing small changes to the existing code of jEdit [jEdit], an open-source text editor. The features of ArchEvol that the subjects used include the use of the concern model, the identification of concerns in code, and the use of visualizations of concerns in
code. The architectural visualizations are not used in this study, because (1) subjects were not required to have working knowledge about software architecture, (2) a few changes will have little to no influence over the architecture of the system, and (3) the other development environments to which we compare ArchEvol do not involve architectural development. In this user experiment, therefore, the users were required not to already know details about the implementation of the project in order to gauge the usefulness of ArchEvol’s features.

The number of small changes that needed to be included in this study imposed a limit on its scope. The tradeoff that was addressed in this study was one between the number of features that could be used by developers versus the duration of the study that is needed to be able to test these features. The study was focused on evaluating the code-related features only, but during a development cycle that included three small tasks that built upon each other.

The second user study addresses the limits of the first experiment, by evaluating the relevance of the architectural visualizations in ArchEvol. The project used as a testbed was ArchStudio, an open source project whose development includes extensive use of an architectural model for its system design. This architectural model was imported into ArchEvol, and then ArchStudio developers were asked to evaluate different concern-related properties and metrics at the architectural level. This study tests, by asking the subjects to comment about the features that they developed, whether the functionality provided by ArchEvol is welcomed by experienced developers. The study also looks at whether the architectural metrics and visualizations are accurate and informative.

The particularities of these two user studies and the time constraints that they imposed made them inappropriate for determining the visible effects of using ArchEvol over a longer time span. Software evolution is a long-term process, and the effects of a few changes to the overall scattering and tangling of concerns will not be significant enough to show any meaningful change in their values.

In order to evaluate the concern visualization and evolution support, we needed to simulate the development of a software system over a longer period of time than just a few changes. In order to simulate this scenario, we imported existing source code changes history data from the versioning
repository of ArgoUML, an open-source UML editor, for the entire lifetime of the project, and used them to simulate the evolution of the project as if it were developed in ArchEvol. One of the contributions of this third experiment is the fact that it shows that ArchEvol can be used for the development of medium-sized, real-world projects such as ArgoUML. The second contribution consists of the data collected while performing the experiment through ArchEvol’s versioning features, data that can determine the importance and relevance of ArchEvol’s evolution visualizations.

Overall, these three experiments provide a comprehensive evaluation of how ArchEvol can be used in software development. The emphasis of these experiments was on breadth of coverage rather than depth, as they tried to cover all the features of ArchEvol and address the duration of development time. The three experiments are small in scale, and the results are not statistically significant. However, they do provide initial evidence that ArchEvol could be used effectively, they show which features worked well and what were the limitations, and they also provide the data that can guide future experiments into focusing on smaller, key aspects of the evaluation.

6.2. jEdit User Study

As we have already mentioned, the goal of this user study was to observe, record, and analyze the actions of a number of developers using ArchEvol. The subjects of the study were asked to perform a series of three small evolutionary changes to an existing system. Two main assumptions are that: (1) ArchEvol has already been used in developing the system, and (2) the developers do not know details about the system’s implementation. In this way, we can observe how developers actually use ArchEvol to explore the code and learn the system while at the same time performing their tasks.

This study evaluated the code-related features of ArchEvol. The main focus is on the concern model, and on whether the users could use and maintain it. Developers should be able to effectively use the concern model to understand the code and to plan their changes, and should be able to update the model successfully after their changes are implemented. Throughout this process, developers
should use the code-related visualizations to understand how the concerns from the model are implemented in the code.

As we discussed above, we intended to obtain data that allows us to compare the results of our approach with other solutions from related work, and we therefore designed this study as a comparative study with three setups. One group of subjects using ArchEvol. Another used the basic functionality in Eclipse (our baseline for the level of support during development). The third group used FEAT, which is the environment with the closest similarity to ArchEvol.

6.2.1. Experiment setup

The experiment used the existing source code base from the open-source project jEdit [jEdit]. A key factor in choosing jEdit as the code base for our study was the fact that, by virtue of it being used in a number of other case studies, jEdit has become a de-facto testbed for case studies in development [Girba et al. 2005, Kirk et al. 2007, Liu et al. 2007, Mariani and Pezzè 2007, Rysselberghe et al. 2006, Zimmerman et al. 2004] [Dagenais et al. 2007, de Alwis et al. 2007, Robillard and Murphy 2002, Robillard et al. 2004]. Being open-source, addressing a domain (document editing) that is easy to understand by the subjects and at the same time having a fairly complex implementation, were other considerations that make jEdit a good choice for the user study.

We already discussed in detail the specifics of the implementation of the jEdit project in Chapter 2, where we used a detailed case study about how concerns are implemented in jEdit to support our motivation for ArchEvol. JEdit is an open source project, hosted by the Sourceforge community [SourceForge 2001], with a medium-sized implementation written in Java. The source code version used in the experiment was the 4.3pre11 version, downloaded in September 2007.

JEdit is an open-source document editor, and the three features that are related to this study are save, backup and autosave. The save functionality is used to save in a text file the buffers (internal representations of documents) that are opened in jEdit. The backup feature saves different versions of the buffer over time. When a buffer is saved, this new version is copied to a specified file-system
directory, creating the equivalent of a basic versioning system. The user can look at the versions in this directory and recover an older version of the buffer, which is useful if the buffer was modified by mistake and then saved. The autosave feature has the purpose of avoiding the loss of data in the eventuality of a crash. In order to do this, jEdit periodically saves a copy for each opened buffer. These autosave copies are meant to keep the unsaved data of the buffers, so every time a buffer is saved, its corresponding backup copy is deleted. When jEdit opens a document, it first looks to find its autosave file, which follows a special naming convention and is located in the same directory as the document file, and asks the user whether to recover or not from the autosave file. The existence of this file is interpreted as evidence of a crash.

Subjects

The experiment used eight subjects, selected from within the Department of Informatics at the University of California, Irvine. The participating students were recruited through advertising material (poster) and were all graduate students within the department. The recruitment material did not mention jEdit, but the subjects were asked at the beginning of the interview whether they saw the source code for the project before. None of the subjects were exposed to the source code of jEdit before.

The subjects’ Eclipse and Java skills were informally evaluated to one of three categories: one for those subjects who only recently started using Eclipse and Java, one for “average” users, and one for experts who were familiar with the functionality of Eclipse and used it extensively. The experience with Eclipse and Java is an important differentiator given the time limit of the experiment, and the less time the subjects will spend on using Eclipse functionality, the more time they will have to understand and perform their tasks.

The subjects were split in three groups, each group having a different setup. Four subjects were included in the main experiment group, who used ArchEvol, while the rest were assigned to two control groups (two subjects in each group), one using the plain Eclipse environment, and the other using FEAT. The assignment of the subjects in the groups was made based on the assessed
experience of the subjects. In particular, we assigned one expert and one regular user to each of the three groups. The other two subjects, one being a novice and one being a regular user, were assigned to the experiment group.

**Environment preparation**

The baseline for the development environment that the subjects in all three groups used was the Eclipse IDE version 3.2 SDK. For the ArchEvol group, ArchEvol and its dependencies were installed. The Eclipse group used the default installation, while the FEAT group used FEAT\(^1\).

The first step in setting up the environment consisted of loading the workspace with the source code for the jEdit project\(^2\). jEdit is written in Java, and its source code was originally organized as a single Eclipse Java project. In order to fulfill one of ArchEvol’s source code organization requirement, the monolithic project organization that jEdit uses had to be broken down in smaller projects. Even though the subjects will not be using the architecture-related features of ArchEvol, splitting the source code into component projects was still relevant in this experiment. When looking at the code, the subjects would have to navigate the Eclipse and ArchEvol views in order to find the relevant source code elements.

The source code from the original jEdit project was split into nineteen smaller projects, each conceptually related to one architectural component. The assignment of source code files to components was based on the name of the Java packages, which already group files into directories. The main keywords in the names of these packages were used to determine the components, with packages that had more than one keyword being grouped together by what we considered the most important of these keywords.

A second part of the study setup consisted of pre-loading the environment with concerns. The study simulates development tasks in an environment where it is assumed that the tools were used previously, that there exists a concern model, and that the concern model is correctly built. The

---

\(^1\) The version of FEAT used was 2.5.4 downloaded in August 2007.

\(^2\) Source code was downloaded from jEdit’s sourceforge.com repository as of July 2007
environment for the ArchEvol group was initially set up with a concern model that included high-level features in jEdit, and in particular, the three features relevant to this experiment were included as top-level concerns with the names, respectively, “save”, “autosave” and “backup”. The concern model was also pre-loaded with links from concerns to the relevant text fragments in the source code. The FEAT group used a concern model that was pre-loaded with the same concerns, while the Eclipse group used a text file with the concern structure only.

The links from concerns to the textual fragments were, in ArchEvol’s group setup, complete with respect to the three features, in that all the source code fragments that were related to these concerns were linked in the concern model. The other concerns in the model were linked to source code fragments too, but their list was not exhaustive. The purpose of these links was to populate the concern model with other data than just the three main feature concerns, but the subjects were not expected to use them as they were related to concerns not relevant to the tasks. In the end, the concern model tree included 13 concerns (“autosave”, “buffer”, “gui”, “browsergui”, “backup”, “save”, “markers” and, as sub-concerns of the “buffer” concern, “save”, “io”, “saveas”, “listener”, “folding documents”, and “deleted text”), with a total of 117 source code fragment references. All fragment references were found by manual inspection of the code.

The concern model in FEAT was also loaded to include the same concerns, but the resulting format of the model differed significantly because of the different paradigms that ArchEvol and FEAT use to link source code fragments to concerns. One difference stems from the fact that ArchEvol links text fragments to identify the code belonging to a concern, while FEAT links Java elements. A second difference is that ArchEvol uses an explicit list of fragments, while FEAT uses queries on relationships between Java elements. As a result, a number of lines in configuration files could that were present in ArchEvol’s concern model could not be added in FEAT, and also, in FEAT, text fragments representing local variables or arbitrary lines of code inside a larger method were identified by the entire method. In order to add these fragments in FEAT, the entire method has to be added. As a result, the FEAT’s concern model included the same 13 concerns but had only 107 references.
Tasks

The subjects were asked to perform a series of three tasks, each involving a change to the jEdit project. One of the tasks that was performed by subjects in our study was also used by Robillard, who performed a user experiment that evaluated the use of FEAT in code exploration [Robillard et al. 2004], and in a different user experiment conducted by de Alwis et al., who analyzed the implications that the use of a concern model has in program comprehension by examining the use of three concern-modeling tools, namely jQuery, Ferret, and Suade [de Alwis et al. 2007].

The number and the choice of tasks is one of the main differences between our experiment and the mentioned related user studies. Since the other experiments looked at how a concern model can be used in program investigation activities, a single change to an existing project was sufficient for their goal. However, our study is geared towards determining problems related to maintaining the concern model during development. Therefore, we needed more than one task in order to show that users can create, use, and maintain the concern links during development.

The three tasks were each designed to simulate an evolutionary change to jEdit. One adds a new feature, a second removes a feature, and a third modifies and combines both features from the first and the second tasks. The tasks were selected so that their implementation would result in changes to the source code that test the concern model updating capabilities in ArchEvol, by adding new code to an existing concern, removing code from a concern, and linking an existing fragment to two overlapping concerns.

The environment setup, including the concern model references loaded, the choice of tasks, and the time limits for these tasks, were evaluated using a small pilot study involving two graduate students. These students were excluded from participating in the study proper.

6.2.2. Experiment procedure

Each subject participated in a two and a half hour experiment session. First, the subjects were
introduced to the goal and format of the experiment for 15 minutes. Then, the subjects were asked to perform the development tasks. The three tasks had time limits of 40, 20 and 30 minutes respectively. If a subject did not finish a task, we would allow a few minutes for them to finish it, but this extra time is not considered in the analysis of their performance. The goal of this extra time is to make sure that the subjects made all the changes to the concern model that they believe were necessary, in order to have the next task start with an up-to-date concern model. In performing the tasks, each subject used the setup for the group to which he or she was assigned. The actions of the subjects were recorded using the Morae recorder [TechSmith], using both screen capture and voice recording.

Introduction

At the beginning of the experiment, the subjects were given to read the Study Information Sheet, a document describing the goals of the experiment, the associated benefits, potential discomfort, and the rights of the subjects. After they read this document, the subjects were asked for their verbal consent to participate in the study. Then the subjects were informed about the background recording of their computer screen and voice, and asked to verbalize their thought process in order to show the motivation behind their actions.

The subjects were then presented their development environment. The environment was identical to the one they would be using to perform the tasks, except that it was loaded with a small sample project rather than jEdit. We limited, in this way, the possibility of the subjects being exposed to jEdit before their tasks started.

All the three groups of subjects were then reminded of the common facilities that Eclipse offers for searching its workspace for information. The three mechanisms that were explained are: (1) the ability to perform a search, either textual or among Java elements, by using the “Search/File...” menu and the dialog box that shows up, (2) the ability to navigate from the point of call of a method to its declaration by using the contextual menu in the Java editor, and (3) the ability to search for all the references to the declaration of a method, also done by using the contextual menu.
The rest of the introduction phase proceeded differently for the subjects based on their group. The subjects in the experiment group were explained and demoed the features of ArchEvol pertinent to this experiment. The subjects were introduced to the concern model and the format of the source code fragments, then were presented the two views for the concern model followed by the two methods that can be used to update the concern model, automatic recording and manual addition. They then were presented the Concern Overview view and explained how to interpret the visualizations in the view and inline in the editor.

The subjects in the Eclipse control group were shown the text file containing the concerns, and were instructed to document their changes to the code base by adding a short description next to the concern in this file. This step was needed in order to evaluate how accurately developers describe their changes and at what level of granularity and precision their descriptions would be. This is the equivalent to manually maintaining a concern model.

The subjects in the FEAT group were presented the FEAT concern model and its associated views. They were explained how to add or remove elements from the editor or the Package Explorer to the concern model, and how to explore the structure of the code by using FEAT queries on the elements in the concern model.

All of the subjects in all groups were explained that the purpose of the concern model is to link its concerns to the source code elements so that the next developers working on the project would be able to find, using the concern model, all the source code fragments related to these concerns. The decision of what source code fragments to add or remove was left to the discretion of the subjects.

At the end of this introductory phase, the subjects were given the opportunity to ask questions that would clarify the instructions given. After all questions were answered, they were presented with the actual project to be worked on, and the task-related part of the study started.

For each of the three tasks, the subjects were given verbal instructions for the task, and shown how the existing jEdit functionality behaves. A written copy of the instructions was also provided in a text file on the experiment computer’s desktop so that the subjects could return to it in case they forget
the exact requirements of a task. The subjects were supervised while performing the tasks, but the conversations were strictly limited to clarifications about the tools or troubleshooting problems that were not related to the task itself.

**Task 1. Enhance the autosave feature.**

For the first task, the subjects were required to change the existing implementation of the autosave feature. The subjects were described the autosave functionality by running jEdit and observing how the special autosave files are created when a buffer is changed, and deleted when a buffer is saved. The subjects were then shown the options pane where the timer value for the autosave is set. Figure 29 shows this options dialog and the text control where the timer value is set with the label “Autosave frequency (secs)”. The change that the subjects are required to perform in this task is to add a new option that will allow the user to explicitly turn on and off the autosave, in contrast to the existing implementation in which a value of the timer equal to zero will turn off the autosave. This change requires the subjects to add a new user interface element to the options dialog, a checkbox whose enabling and disabling will enable or disable the autosave feature.

We used this change as a baseline for our comparison with the two pre-existing user experiments that used jEdit, which also included this change as part of their experiments, so we can benchmark our results against the results of these other experiments. However, there are a few differences between the way we conducted the experiment and the other two experiments. First, while the other experiments separated the code inspection phase from the task implementation phase, we did not make such a differentiation. At the same time, while the other experiments forbade the use of debuggers, we allowed them as well as any other functionality that the subjects needed from Eclipse. This decision was taken on the principle that the subjects should be allowed to use any tool that they consider necessary in order to perform the changes, and that the experiment should not interfere with their activities, which will provide a more accurate simulation of a real-world development scenario. By allowing the subjects to use debuggers and run their changes, however,
our experiment would include the time that the subjects spent on these two activities in the total implementation time. To differentiate between the time spent on each activity, we used the screen recordings taken during the experiment and identified all activity switches that subjects made while implementing the task, as described in more detail in the data analysis section.

The subjects were then reminded to maintain the concern model as accurate as possible, and were explained a series of steps that they could use to determine that jEdit was behaving as expected. The same steps were listed in the written instructions in order to be accessible during the task.

From this moment on, the effective task implementation phase started. This part of the experiment was strictly timed, and the subjects were told about the time limit, which for this task was set at 40 minutes.

**Task 2. Change the backup feature.**

The second task required a change to the implementation of the backup feature. The backup feature is responsible for saving copies of the buffers, giving the user the ability to go back and restore the content of the buffer from a previous saved state. While the autosave file always contains
the unsaved state of the buffer, the backup files are only created from saved versions. Multiple backup files can be kept, in which case an elementary versioning system is created.

The user has the ability to specify a number of parameters to how the backups take place, using the same options dialog shown in Figure 29. One of the options is the maximum number of backups kept for a file. When multiple backup files are kept for the same buffer, they are identified by an order number, ranging from one to the maximum number of backups specified. These names are rotated between backups in such a way that the newest backup is always number one and older backups are re-numbered to start from two. Another option is the file system directory where the backup files are created. This is a different directory than the one where the buffer is saved regularly. Two other options specify a prefix and a suffix that will be used to create the name of the backup files. The user can also specify whether the backup will take place every time the buffer is saved instead of the default, first-time behavior.

The change that this task required was to remove the use of a user-specified prefix and suffix in the naming of the backup files and instead use only a fixed prefix “backup” and no suffix. The purpose of this task is to determine if the subjects can use the already linked fragments to the backup concern in order to identify where the prefix and suffix elements are used.

**Task 3. Combine the backup and autosave features.**

We mentioned previously that the user can specify in jEdit whether the backup files should be created on every save of the buffers. In this task, the subjects are asked to modify this feature so that the backups are created every time an autosave of the buffer is triggered. This change will change the backup feature storing versions of the saved documents, to storing versions of the changes made to the document, but not yet saved.

An intentionally subtle aspect of this task was the fact that it is not just about implementation, but also the meaning of the existing code is changed. Code that belonged previously to the backup concern, will become now associated with both the backup and the autosave concern.

There are different ways in which this task could be implemented, and to do a good implementation
requires more knowledge about the jEdit’s design than the subjects could reasonably be expected to grasp during this study. Even if they could find a working solution, there might potentially be critical differences between the solutions chosen by the subjects, which would make a comparative evaluation impossible. To avoid such a possibility, we limited the scope of this change by providing the subjects with a hint about how to perform the task: the subjects were told that the code related to backup from the BufferSaveRequest class had to be moved into the BufferAutosaveRequest class. This hint would allow the subjects to spend less time on planning the change, and instead focus on maintaining the concern model consistent with the changes in the code that they are performing.

**Verification phase**

At the end of the three tasks, we asked the subjects to spend a few minutes inspecting the concern model that resulted after the three changes. This part was needed in order to make sure that subjects who intended to change the concern model, but ran out of time performing the task itself, could get a chance to update the concern model as they intended. The subjects were asked to look at all the fragments related to the backup concern and determine whether the concerns remained marked as intended by them. This provided a simple validation step for the concern model and its algorithms for keeping the source code fragment references accurate as the subjects modified them while performing their tasks.

**Exit questionnaire**

At the end of the experiment, each subject was asked to fill out a questionnaire. The questionnaire asked details about their past programming experience and about their impressions regarding the environment that they used. The subjects who used the regular Eclipse environment answered a smaller set of questions, which asked them about the potential use of a concern model in general. The users who used ArchEvol and FEAT were also asked specific questions about their respective environments.

At the end of the study, the subjects were compensated for their participation with 50 US dollars.
6.2.3. Data Analysis

The recordings of the computer screen activity during the experiment were used as the basis for the analysis of the performance of the subjects. The recordings were processed using the Morae server and transformed to video files that were used for the analysis.

The process that we used in analyzing the videos was a manual recording of context switches that the subjects made during their tasks. We identified the following types:

- **Navigate.** This category contains the time spent by a subject in searching information in the Eclipse workspace. This includes: (1) looking and examining the workspace resource tree view, the projects, and their packages, (2) starting searches using the Eclipse search wizards, and (3) examining the results of a search. For subjects in the ArchEvol group and the FEAT group, this also included navigation by using the concern model.

- **Inspect.** By inspection, we mean the perusal of a single code file, the time spent by the subjects reading and understanding the content of a file.

- **Modify concern model.** This category included the time spent on selecting code in the editors and assigning it to a concern, as well as the removal of existing links from the concern model.

- **Code and test.** Under the code category was included the time spent on typing text in the editors. Time assigned to testing included not only the time spent using the debugging features in Eclipse (which the subjects used very little), but also all the instances when jEdit was started in order to look again at the options pane, or to determine if jEdit behaved as expected after the changes were implemented.

- **Talk or read instructions.** Subjects would sometimes stop working so that they could finish verbalizing their action. We wanted to be able to quantify this time, as well as the time spent on reading the instructions in order to be more accurate reporting the actual time spent on the other activities.

A typical subject switched quite frequently between these tasks, which made the analysis process
tedious. Because of quick mouse movements, it sometimes was difficult to differentiate between two different activities. For example, subjects would typically use a list of results for a search by clicking on an element in the list, quickly perusing the code that Eclipse brings into focus, then clicking on the next item in the results list, and so on. Moreover, the subjects would sometimes start to move the mouse from the editor, thus ending a coding activity, towards one of the views, where a navigation activity would start, but would hover for a few seconds in transition. This transition time could be split in between the two activities, or, if it is long enough and the subjects talk during this time, it would form a new activity by itself. Because of these cases, the overall accuracy of the context switching duration that we used in collecting data is within a boundary of about 2 to 3 seconds.

We used a timeline visualization to show the information we collected from analyzing the recordings. In order to achieve this, we adapted the SIMILE Timeline [The SIMILE Project], a web-based timeline visualization tool. SIMILE Timeline displays event-based data in an interactive HTML visualization. We used a small program to transform the data from our Excel spreadsheets in an XML format that SIMILE could understand.

Screenshots of the resulting visualizations, for each task, are shown in Figure 30, Figure 31 and Figure 32, respectively. For each developer and task, the timeline visualization displays the types of activities that a developer is engaged in while performing a task. Each activity is depicted as a small colored bar, with the length of the bar representing the duration of the activity.

The duration of the task is shown on the X-axis, which indicates minutes with markers. Each task had a time limit, which is indicated with a darker background color, but the subjects were allowed a few minutes to finish their tasks after that limit, which shows in the timeline as activities that are past the highlighted background. The times past the deadline were not considered, however, in the following calculations but can indicate whether the subjects were close to finishing the task or not.
Figure 30. Task 1 analysis results. Shows subjects a) navigating, b) inspecting, c) modifying the concern model, d) coding and testing and e) talking or reading.
Figure 31. Task 2 analysis results. Shows subjects a) navigating, b) inspecting, c) modifying the concern model, d) coding and testing and e) talking or reading.
Figure 32. Task 3 analysis results. Shows subjects a) navigating, b) inspecting, c) modifying the concern model, d) coding and testing and e) talking or reading.
The activities are grouped, on the Y-axis, by the five categories mentioned above. The type of activity in each of these categories is identified by a different color. For example, in the category “code and test”, shown in the timelines on line (d), coding and testing are represented with different colors. The categorization of the activities is helpful in showing how often the subjects switch context by moving from an activity in one category to another activity in another category, and how much time they spend performing activities in the same category.

The data about the activities performed was used to analyze the behavior and performance of the subjects. We performed the following analyses:

- **Analysis of development time.** Using the timeline visualizations produced by transcribing the screen recordings, we look for patterns in the activity switches and correlation with other characteristics of the subjects.

- **Analysis of performance.** This analysis looks at the performance of each subject, by scoring the implementation choices and the number of sub-tasks implemented.

- **Analysis of activity times.** The concern model in ArchEvol can be used to navigate the code based on concerns, and the visualizations that ArchEvol provides should help developers understand the code quicker. In this type of analysis, we test this hypothesis by comparing the total time subjects spent on navigation and inspection times to their performance.

- **Analysis of concern model maintenance.** We look at how well the subjects were able to maintain the concern model after the task was performed. This analysis can provide an estimation of how much effort is needed to maintain a concern model and to what extend the concern model is updated after the tasks are performed.

- **Analysis of the exit questionnaire.** The exit questionnaire asked subjects about their overall impression of using ArchEvol, and about their view on the potential benefits and drawbacks of using ArchEvol.
Analysis of development time

One of the goals of the study was to build an understanding of how subjects used ArchEvol as compared to the other two environments in performing their tasks. For this purpose, we used the data shown in Figure 30, Figure 31 and Figure 32 timelines as the basis of our analysis. The first observation that can be drawn from this data is:

*Observation 1. The process through which developers understand an existing implementation is characterized by frequent switches between navigation and inspection.*

This pattern of quick switches between navigation and inspection activities can be seen in all tasks, being performed by all subjects. A typical sequence of actions is searching for a keyword, then selecting one of the results, double-clicking it and inspecting the source code context for the result, and, after determining that the context is not related to the code, moving to the next result. Subjects tend to repeat this sequence of actions until they found the code that is relevant for their search.

Overall, there are different factors that influence the length of these context switches. One is the ability of the subject to produce an educated guess on the keyword that would return the best results. A second factor is the user’s familiarity with the tools. For example, Subjects 5 and 12 have considerably longer navigation intervals than the rest of the subjects. In analyzing this discrepancy, we found that these subjects tried to search for a keyword at the beginning of Task 1, but failed because they did not find an option in the search wizard that would let them control case sensitivity. Some subjects tried two or three times to perform the search, without being able to see that the options in the wizard would limit their search, and then gave up when no results were found.

*Observation 2. There are no meaningful differences in the patterns of the activity switches that differentiate the subjects based on the tools they use.*

A visual analysis of these timelines did reveal that the patterns of development for the subjects vary greatly. However, it is impossible to strongly associate these patterns with the use of one tool or the other. Rather, there are individual differences in the ways subjects try to approach the tasks that equally well could explain the different patterns. For example, Subject 1 (Eclipse) was noticeably
more reluctant to changing the code before trying to determine what the effects of the change would be. As a result, we can see in all of the tasks performed by Subject 1 more navigation and inspection than the other subjects.

The length of the activity blocks and the variation between them also seems more related to the subject than the tool. For example, Subject 6 spends more time per activity than Subject 9, and therefore the length of each activity bar in the visualization is longer. Subject 7 has a different pattern, with longer activity times for the first task, smaller for the second, and mixed for the last task.

This observation suggests that the difference in development habits between different individuals might be a more important factor than the use of a concern model. The observation is in line with previous user studies, which also showed that the variation between individuals was greater than the variation between different concern identification tools [de Alwis et al. 2007].

**Analysis of Performance**

While the timelines are useful in showing how subjects spent their time, it does not tell us anything about the quality of the task. In order to quantify this quality, we used a score system that assigns a value to each small part of a task that the subject performed. Each of these sub-tasks could have been implemented in different ways, and to be able to compare them we assigned each implementation choice a score. We scored an implementation based on its logical correctness (the change had to do what the task required), on the subject’s adherence to the programming style of the surrounding code, and on possible side-effects that the change introduced but of which a subject was not aware. An implementation that satisfies all these three criteria receives a score of 1, an average implementation a score of 0.5, and a “bad” implementation a score of 0.

The total scores received by each subject are shown in Figure 33. Each subject is shown on a separate line, and for each subject there are three segments, one for each main task. The exact values for the score are shown in the middle of the segment. The results show that, overall, the performance of the subjects is in agreement with our assessment of their Eclipse and Java
knowledge. The “expert level” subjects had similar results, better than the “regular level” subjects. From the two subjects in the “beginner level”, only Subject 2 had a score comparable with the ones at the “regular level”, even though the subject was not able to finish the first task at all. But, overall, the performance of both Subject 2 and Subject 5 was less than satisfactory, with both subjects not being able to implement any of the sub-tasks that were part of task 1. Subject 5 had difficulty in performing the rest of the tasks too.

The low scores of the two subjects in the beginner level, Subjects 2 and 5, were due in a large part to their inability to understand a Java program of the size of jEdit in the time allocated. We believe that their inexperience with Java and Eclipse was the biggest factor that stopped them from performing the tasks. Although these two subjects used ArchEvol, all the other views and wizards that the Eclipse-only installation provides were readily available to use, and the subjects were not restricted to using only ArchEvol views. However, even thought the records show that the subjects tried to explore the code with using the Eclipse wizards, they were missing some option in the wizard, such as forgetting to set the “ignore case” option when searching for text, which returned an empty search result several times. Another example is that, although the subjects were presented the dialog in jEdit where the save and backup options were displayed, their inexperience with GUI
programming in Java made the search for the code elements where those options were implemented difficult.

**Analysis of activity times**

An advantage of having a concern model lies in its grouping of all source code fragments that are related to a concern in a single place. Therefore, the subjects should be able to navigate the source code by concerns and not by the results of generic searches, which theoretically should be much faster. However, the time alone is not a good criterion on which to evaluate if the subjects were successful in performing their tasks. In order to help gauge whether the subjects were successful in using the concern model, we need to correlate their time spent on navigation and inspection to their performance.

The first analysis that we performed looked at the time the subjects spent on navigation only, by adding up the durations of the times spent on navigation by the subjects as a total, over all three tasks. We used the results of the performance analysis, as shown in Figure 33, and plotted the total navigation times against the performance scores. The results of this analysis are shown in Figure 34. The subjects who used ArchEvol are depicted with round marks, Eclipse with a small diamond marker and FEAT with a square marker. The total time spent on navigation is shown on the X-axis, and the total scores of the subjects are shown on the Y-axis.

A second plot shows the sum of the durations of the intervals that the subjects spent on both navigation and inspection against the subject performance, shown in Figure 35. The rationale for having both navigation and inspection times added together is that, in essence, both activities help the developer in understanding how the code is organized. Navigation looks at the structure of the code as shown by a view in the environment, as for example the Package Explorer View in Eclipse, which shows the packages and the names of the contained classes. While inspecting a piece of code, developers are trying to understand local code organization. Together, these both activities help the developer create a mental picture of the organization of the code.
Figure 34. Navigation time versus performance.

Figure 35. Navigation and inspection time versus performance.
By looking at both Figure 34 and Figure 35, it becomes clear that there is a strong correlation between the experience of the developers and their performance. The subjects who were evaluated as experts had considerable better scores and finished their tasks in considerably lower time than the other subjects. From the group of subjects using ArchEvol, Subject 2 had the shortest time, about 1 minute less time spent on navigation compared to Subject 7, but the score was much lower (16.5 compared with 23.5). For the subjects using Eclipse only, Subject 1 had both the best score and spent the least amount of time navigating the code. Subject 12 and Subject 11, who used FEAT, had comparable performances. Subject 12 spent less time on navigation, but with a lower overall score, while Subject 11 spent about 27% (9'18'' versus 6'47'') more time with a result of about 30% (22 versus 16.5) better score.

The differences for subjects with comparable levels of experience across the three environments are also considerable. Subject 1, using Eclipse, had the highest score, but Subject 7, while having a slightly lower score (23.5 versus 24 points), had a considerable lower navigation time, more than 50% lower (3'30'' versus 7'02''). The difference against FEAT is also considerable, as Subject 7 had a better score and took about a third of the time of both Subject 11 and Subject 12. For almost all the subjects, the results from Figure 34 are preserved when we add the inspection times as shown in Figure 35. The only notable exception is Subject 9, who seems to spend a lot more time on inspection relative to navigation than the other subjects.

One of the differences that could account for the difference in time between subjects using different environments is that, by using the concern model from ArchEvol (or FEAT), subjects perform a different type of navigation of code, based on concerns instead of files or classes. In order to evaluate the impact of these differences, we looked at the times spent on the concern model.

We plotted the sum of the navigation time and the use of the concern model time versus the performance in Figure 36. The results show that the subjects with the Eclipse environment spent overall less time navigating than the other subjects did navigating and using the concern model. One plausible explanation for this result is the fact that subjects would sometimes start searching for a
feature using the Eclipse search wizard (which is included in the navigation time), but upon an unsatisfactory hit they would turn to the concern model and search for the same features again, and as a result the search would take longer to perform.

Yet another factor that could influence these results is the fact that the subjects might not be accustomed to the environment while performing task 1, which would result in the times calculated for task 1 to be an outlier with respect to the rest of the data. If we consider only the times for the tasks 2 and 3 combined, we have the results shown in Figure 37. In this case, the scores of Subject 2 are better than the scores of the other subjects. This is due to the fact that this subject was not able to finish the first task without getting assistance, and therefore had to be explained what to do.

In this case, we have stopped taking the rest of that task into consideration (also shown in the shorter timeline in Figure 30). However, for the second and third task, this subject used the concern model in ArchEvol in a more effective manner, which is reflected in the score received.

For the rest of the subjects, however, it can be seen that the differences between the subjects are noticeably smaller than in the case with task 1 included, and this is not only due to the fact that the tasks themselves have been shorter. The times of the subjects Subject 9 and Subject 5 are noticeably shorter than the times of the subjects Subject 1 and Subject 12, respectively.

The overall observation stemming from these four graphs is:

Observation 4. For subjects that achieved similar completion scores, the times spent on navigation and inspection of the code are shorter for the ArchEvol group than the ones obtained by the Eclipse and FEAT groups.

The results are consistent whether we look at navigation times only, at navigation times and inspection times added together, at navigation and concern model use, and at times spent on the second and third tasks only.
Figure 36. Navigation and Concern Model use time versus score.

Figure 37. Navigation and Concern Model time versus score for tasks 2 and 3.
We attribute these results to the fact that the subjects are not spending as much time to search for features in code, and they do not spend the same amount of time in inspecting the code since the concern model in ArchEvol provides them with pinpoint information. This is an encouraging result for the potential of ArchEvol, as it represents initial evidence that the use of ArchEvol helps developers understand the code of an existing system.

**Analysis of concern model maintenance**

We also looked at how well the concern model was maintained during the three tasks. ArchEvol offers two methods that make this easier. One is manual assignment, in which a subject selects a source code fragment and assigns it to a concern. The second is automated recording, in which new code is automatically assigned to one or more selected concerns. In addition to these two methods, ArchEvol uses simple heuristics to add adjacent code additions to an already marked code fragment to the same concerns.

In order to be able to compare the performance of the subjects in maintaining the concern model, we first defined a set of fine grained code related actions that an “ideal” implementation of the task should have achieved, then we matched the implementations of the subjects to these actions. A total of 34 actions were identified for the three tasks combined. For each of these actions, a subject should have added, removed, or modified a small block of code. As examples, in Task 1, the subject should have declared a new checkbox and initialized it. Task 2 implied the removal of the prefix textbox element, and Task 3 required changing the label of the backup checkbox.

We counted, for each subject, if the code related to each action was marked in the concern model by the subjects themselves, or if it had been automatically added to the concern model by ArchEvol. Not all of the code fragments that were not added to the concern model were overlooked by the subjects. Some of them would not apply because of the particular implementation that the subjects chose for a task, and, in the case of FEAT, marking up text in property files was impossible to do.

Figure 38 shows the performance of each subject, on each row, as segments of different color. The length of the bar for each subject is the same, and corresponds to the total number of 34 actions.
Each segment represents one of the types of results related to the maintenance of the concern model, and their length is corresponding to the number of actions that fall in the category. The segments are, from left to right: (1) manual maintenance of the concern model by the subject, (2) automatic maintenance by the tool (only for ArchEvol and FEAT), (3) fragments of code that should have been added but were not because of omissions by the user, (4) fragments of code that could not have been added because they were irrelevant in the implementation chosen by the subject, and (5) fragments that could not be added because of the way the environment functions (FEAT only; the difference between the two FEAT subjects is that Subject 11 had more of these types of fragments in the implementation chosen than Subject 12).

The analysis of the ability of the subjects to maintain the concern model is based on the data shown in Figure 38. The value in which we are interested is the ratio of the fragments that were added or removed from the concern model over the total number of the fragments that could have been added or removed. Translated into the fragments on the figure, we are looking for the sum of the first two segments over the sum of the first three segments, for each subject.

The Subjects 7, 9, and 5, who used ArchEvol, maintained a percentage of 60%, 76% and 68% of the total number of fragments, respectively. These three subjects used the manual assignment option, and therefore they were responsible for all updates to the concern model. The fact that their scores are well lower than 100% is due to the fact that in some cases they did not think that adding a fragment to the model is necessary or helpful, or they simply forgot to take action.

The fact that the subjects forget to accurately document their changes is one of the common threats to maintaining separate documentation. Other reasons that can explain this score are the fact that the subjects were not accustomed to the role and goal of the concern model yet, and were not aware of the value of the concern model. With time, if the users determine that the concern model is useful in guiding their work, they would be more motivated for their future changes to maintain it as accurate as possible.

One of the subjects from the ArchEvol group, Subject 2, used the automatic recording of concerns,
with substantially improved results. The score for the Subject 2, who used ArchEvol, was 93%, due to the fact that the responsibility for adding text fragments to the concern model was taken by the environment rather than the user. This result suggests that the automatic recorder would be more reliable than the user. Unfortunately, the subjects were explained at the beginning of the study that they can use any of the two methods of adding concerns to the model, manual and automatic recording, only Subject 2 chose to do so, which makes the result not strong enough from a statistical point of view.

For the subjects in the control group that used Eclipse only, the concern model was a simple textual list of concerns, which is a different setup from the other two groups. The subjects were instructed to make a note of the changes in a text file, with the intent of observing what type of comments they would use and how effective would they be. In the instructions given, the granularity of the comments was not specified. As a result, the subjects wrote high-level descriptions of the task, which did not cover all the small changes that they implemented. This explains the results of less than 40%.

Regarding the control group FEAT, a number of source code fragments that were found in

![Figure 38. Concern model maintenance results.](image)
configuration files had to be excluded from calculations since FEAT cannot trace them. The results for the two subjects are, if we do not consider these fragments, 70% and 87.5%, respectively. If we take these fragments into consideration, the results are 50% and 77%. These values come, however, at the cost of accuracy of the concerns. In FEAT, arbitrary code fragments cannot be linked to the concern model, and therefore their enclosing methods have to be added. This action has the side-effect of including far more text than just the relevant code fragment, which is less accurate than in ArchEvol since the users still have to inspect the entire method to find out exactly where the fragment related to a concern is. As Figure 36 and Figure 37 show, the subjects using ArchEvol spent considerably less time on navigation and concern model use than the subjects using FEAT, which indicates that the benefit of having a more precise concern model can translate in shorter time spent in examining the code based on the concern model.

Another side effect is that, by adding an entire method to the concern model, the users would automatically include all other fragments that were found in the same method, even if they did not intend to. Our scores were calculated to include all fragments, even the ones incidentally added by referencing the enclosing method. This situation is reflected in the high score of the subjects using FEAT.

The results of this analysis support the following observation:

*Observation 5. Concern model maintenance was performed with similar results in both ArchEvol and FEAT. The automatic recording option (although used by only one subject) had noticeably better results.*

The subjects using ArchEvol were able to maintain about 68% of the concern model on average (considering the three subjects that modified the model manually), which is comparable to the results of the FEAT group, whose results were 78.75% (if fragments in text files are excluded from calculations) and 63.5% (with text file fragments included). The tradeoff, based on our results, seems to be between a slightly better score for the FEAT group versus a better accuracy of the fragments for the ArchEvol group.
A second encouraging observation is related to the potentially much better performance of the automatic recording of concerns. Although we did not have enough data for a statistical conclusion for the performance of automatic recording of concerns, since only one subject used it, the results are much better than manual recording. A further study would need to be carried out to validate this hypothesis more rigorously.

**Analysis of the exit questionnaire**

The first part of the questionnaire asked the subjects about their general level of experience in programming large applications. The result of this evaluation was in line with our initial estimates of the experience of the subjects, based on which we assigned them to the three experiment scenarios. Subject 5 and Subject 6 had one year of programming experience in Java, while all the other subjects had between 3 and 4 years of experience. Subject 9 had no experience with programming parts of a large-scale system, while all the others had some experience with developing such systems.

The second section contained questions about the usefulness of a concern model in general, and of the tools used in the experiment in particular. The first question asked subjects to evaluate the usefulness of a concern model that keeps references from concerns to implementation, in general. The choices for the answer were: (1) “Definitely not useful”, (2) “Not useful”, (3) “Useful” (4) “Somewhat useful”, and 5) “Definitely useful”.

The subjects who used Eclipse answered “somewhat useful” (Subject 1) and “definitely useful” (Subject 6). The subjects using FEAT also answered “definitely useful” (Subject 11) and “somewhat useful” (Subject 12), with the comment that it might be used more by novice developers trying to understand the system. The subjects using ArchEvol answered “definitely useful” (Subject 2, Subject 6, Subject 7 and Subject 8) and “useful” (Subject 5). Comments in support of the answer mentioned the fact that it helps users in finding classes and methods to change and therefore make changes easier, and that it helps new developers in understanding a new system.

Question 2 asked about the specific usefulness of ArchEvol and FEAT. The subjects using FEAT
answered “definitely useful” (Subject 11) and “somewhat useful” (Subject 12), while subjects using ArchEvol answered “definitely useful” (Subject 9), “somewhat useful” (Subject 2 and Subject 7), “useful” (Subject 5). The two subjects in the Eclipse group did not answer this question.

The third question asked whether using ArchEvol or FEAT helped the users in making more considerate decisions about where to place the additional functionality for their change. For ArchEvol, 3 subjects responded “yes” and one responded “no”, while for FEAT, one responded “yes” and one responded “no”. The notes detailing the answers included arguments in favor of ArchEvol, such as the fact that the concern model shows parts of the relevant code that are not obvious otherwise, or that it helped the users remember the roles of files in a large project. One of the observations related to ArchEvol pointed out that if a concern has too many code snippets, then finding the right ones that are related to the change might be time consuming.

The subjects were then asked about their opinion as to whether the use of ArchEvol and FEAT would lead to a system with better managed concerns than a regular Eclipse environment. All subjects responded with either a “better” or “slightly better”, the two top choices. The other choices that they could have chosen were “the same”, “slightly worse”, and “worse”.

Questions 5 through 8 were specific to ArchEvol. We used these questions to understand the overall impression that using ArchEvol made on the subjects, and therefore asked them whether they think that ArchEvol can be used in a real-world development setting, whether they would recommend ArchEvol to their team if they were employed in a company, whether they would use ArchEvol themselves, and whether they had any suggestions for improvement. The answers were overall very positive, the subjects answering “yes” to the first three questions, except for an answer where the subject would not use ArchEvol because they did not use Eclipse as a development environment. The suggestions that the subjects proposed were simple user interface enhancements, such as double-clicking on a file in the concern model to open its associated Eclipse editor, or the ability to choose different colors for the highlight of concerns in code. One of the two problems pointed out was that a red color for highlighting code induces the idea of an error marker (which in Eclipse is
typically a red squiggle line under the text with the error), which is confusing. The second problem
was that two concerns with very similar colors, such as two shades of green, for example, are
difficult to differentiate if they were visualized together.

The third and last part of the questionnaire asked the subjects about their particular experiences in
performing the tasks. The first question asked them to evaluate the difficulty of the task on a scale of
five values, from “very difficult” to “very easy”. Six subjects chose “fair”, the middle option, as their
answers, while Subjects 5 and 6 from ArchEvol and Subject 11 from FEAT considered the tasks
difficult. In all the answers, the comments of the answers made it clear that the main difficulty that
the subjects had was the fact that they needed to familiarize themselves with how JEdit works, and
that the second and third task were considerably easier.

The second question asked the subjects to evaluate the usefulness of the concern model. This is
essentially the same question as the first question in the second part of the questionnaire, and was
used here to cross-check the subject answers. Only Subject 1, who was included in the plain
Eclipse scenario, said that the concern model was not useful because it was initially empty. As this
subject was using the Eclipse environment, the subject was asked to simulate a concern model by
noting changes in a simple text file, so determining that this text file
was not useful is not a surprise.
All other subjects answered “yes”.

The third question asked the subjects about the difficulty of maintaining the concern model itself.
This question is related to the second question in the previous part of the questionnaire, which
asked them about the usefulness of ArchEvol. The answers here were varied. The two subjects who
used Eclipse answered “fair” and “easy”. Two subjects who used ArchEvol answered “difficult” and
three answered “fair”. One subject who used FEAT chose “easy” while one described it as “difficult”.
The main observation from the answers that chose “difficult” was that the activity of maintaining the
concern model interrupted the thoughts of the users as they were concentrating on the task at
hand.

The last two questions asked the subjects about their evaluation of the usefulness of the highlight
colors in text and of the side Concern Overview visualization in ArchEvol. For the highlighting of text, two answers were “useful”, one was “not useful”, and one was “very useful”. Comments from these answers pointed out that highlighting makes obvious where concerns are implemented in code, but also that the color of the highlighting can be confusing if it looks like a common indicator in Eclipse for syntax errors. For the concern overview window, one answer was “useful”, one was “not useful”, and two answers were “very useful”.

6.2.4. Threats to validity

The purpose of this experiment was to observe the subjects in using ArchEvol, as well as to provide a comparison with two other existing setups. While the experiment provides an evaluation of ArchEvol, there are a number of factors that need to be taken into consideration in interpreting the results of this experiment. The most important threats to validity discussed here are the relatively low number of subjects, their selection from a small subset of the overall population, and the small scale of the changes that the subject implemented.

The relatively small number of subjects will not allow us to provide strong statistical evidence for our conclusions. This study was nevertheless important in uncovering a series of important observations about ArchEvol and its comparison with the other two environments, Eclipse-only and FEAT. The results are encouraging, and can form the basis of future studies that can focus on a larger number of subjects.

The subjects were all selected from graduate students in the Department of Informatics at the University of California, Irvine. As such, their Java and Eclipse skills might not be representative of a larger population of all Java and Eclipse developers.

The changes that the subjects were required to implement were of relatively small nature. The focus of our study was to include a few changes that are related to each other and that cover a number of different operations that might take place in the concern model. The concern model used was populated with a few concerns, and the changes were not too big so that the subjects would have time to finish them in the allotted time. In a real-world scenario, however, developers might work with
larger systems that include many more concerns in the concern model, and perform changes that are larger in scale. The limitation of our study is a general limitation that any user study has. To ensure that our changes were not too simple, we chose changes comparable in size to changes in other experiments [Dagenais et al. 2007, de Alwis et al. 2007, Robillard and Murphy 2002, Robillard 2003, Robillard et al. 2004].

6.2.5. Conclusions

The purpose of this study was to provide an initial evaluation of ArchEvol and a basis for its comparison with the other existing development environments. The study unveiled a number of encouraging observations about ArchEvol, which can be used as the basis for future experiments.

Conclusion 1. The subjects in the experiment understood the features in ArchEvol and were able to use them during development towards performing the required tasks.

One of the first results is the fact that the study, by the virtue of having a number of different developers as subjects, evaluated the basic usability of ArchEvol. We found that the subjects could understand the role of the views and menu options in ArchEvol without difficulty, and used the features in ArchEvol to perform their tasks. We have seen that the concern model in ArchEvol can be used effectively to navigate the source code based on concerns.

Although the evidence was not statistically significant, this experiment provides encouraging initial data that supports the fact that concern model can be effectively maintained by the users through development. ArchEvol provides two mechanisms that can be used to maintain the concern model, and the results of the study show that manual selection effectively maintains 70% of all fragments, while automatic recording can potentially be used to maintain 93% of all fragments.

Conclusion 2. The results of the subjects using ArchEvol were largely better than the ones in the Eclipse group, and at the same time were comparable to the ones in the FEAT group.

Our study provided data that can be used to compare the performance of ArchEvol with the baseline formed by the Eclipse platform, and with an alternative environment, FEAT. The data
collected from the study shows that, for subjects with the same level of experience, the use of ArchEvol leads to a better performance compared to either Eclipse only or FEAT. The analysis of the concern model maintenance shows that the subjects using ArchEvol were able to maintain a comparable percentage of fragments as the subjects using FEAT.

6.3. ArchStudio User Study

The purpose of this user study was to evaluate the features of ArchEvol related to the visualization of concerns, and the associated metrics, at the architectural level. While the jEdit user study focused on the identification of concerns in source code and on the evolution of the concern model as the software system is being evolved, this study focuses on the analysis of the data collected in the concern model, and on the effectiveness of the architectural visualization. Together, these two studies offer a complementary evaluation of all of the features of ArchEvol.

In designing this study, we considered the most important factors that influence the evaluation of the architectural features in ArchEvol. First, the system that ArchEvol is used on should be a medium-size system that already has an explicit architecture. The medium-size requirement is a slightly imprecise measure, but it is important that the system should be large enough that its architecture is not trivial. At the very least, the components in the system have to be bigger than a single class, since the metrics that ArchEvol uses are aggregated at a class-level.

It is also important that the system has an explicit architecture in place. Although we can attempt to recover the architecture of any system based on various heuristics (such as package or class names, hints from documentation, and so on), in the end architectural recovery is still an imprecise exercise. Decisions that are made while recovering the architectural model might not be shared by the developers, so the recovered architecture might not be the same as the conceptual architecture. Having an explicit architecture in place eliminates such discrepancies.

A third factor that influences the evaluation of ArchEvol’s architectural features is the requirement that the participants understand the concept of software architecture, and have experience using it
actively. Without having used an architectural view during development, participants would have an extra hurdle in interpreting ArchEvol’s visualization, and in associating the components in ArchEvol’s view with components in the implementation. The architectural view that ArchStudio developers use is shown in Figure 39.

A fourth requirement is that the participants should be knowledgeable about the system being developed. In order to form opinions about where and how concerns are implemented in the system, the participants should have worked on the implementation of the system. Without any knowledge about how different concerns are implemented, the participants will not be able to tell if the metrics shown by ArchEvol are reasonable or not.

All these requirements come in sharp contrast with the setup of the jEdit user study. Although jEdit is a medium-size system, it has no explicit software architecture. In the jEdit study the participants were not exposed to jEdit’s implementation before the study. They did not need to know where concerns are implemented in the system, and were also not required to understand software architecture, or to have used an architectural model before. As a result, the same study setup that was used in jEdit’s user study was not adequate for evaluating the architectural features of ArchEvol.

The requirements enumerated above impose constraints on the selection of the systems that can be used and the participants who can be asked to participate in the study. The environment chosen for this user study is ArchStudio [Archstudio 2008], an architectural development environment developed and maintained by the Institute of Software Research at the University of California, Irvine.

Two ArchStudio developers were asked to participate in the study, both being graduate students in the Department of Informatics at the University of California, Irvine. They have both contributed to ArchStudio’s development, and they have even collaborated in implementing some of its features.

Only one of the participants had prior knowledge of ArchEvol’s features, but did not work with the architectural visualization or architectural metrics before. The other participant had no previous exposure to ArchEvol.
6.3.1. Background

The ArchStudio environment consists of a set of tools that support the development of software systems that have an explicit architecture described in the xADL2.0 architectural language [Dashofy 2003, Institute for Software Research]. ArchStudio includes a number of features that make architectural development easier: a graphical editor, a development framework, a testing framework, architectural critics, and other specialized architectural tools.

ArchStudio is used to further its own development, and has an explicit architecture described in the xADL2.0 language\(^1\). As Figure 39 shows, the architecture of ArchStudio is not trivial, including 46 components, 32 connectors and a relatively dense diagram with a large number of links between these architectural elements. Components are denoted by a slightly darker color than the connectors.

The layout of the architecture is important because it includes clues that are not formally encoded in the architectural description. For example, the bottom layer in ArchStudio’s diagram consists of components that are mostly UI-oriented. In general, the higher components are laid out in the architectural diagram, the more data-oriented they are. The top component (xArchADT), in particular, holds the entire data model for the architecture being developed.

Grouping provides another clue from the layout of the diagram. While any two components that are linked together are related because they produce and consume the same data, the visual grouping of components is not only based on links. It is also based on the knowledge of the developers who created these components, and who have placed them next to each other because they conceptually address similar concerns. The clues that are encoded in the layout of the architecture are important because they influence the answers that participants give when asked about which components implement a given concern. For example, if the majority of components in a layer

---

\(^1\) When talking about ArchStudio, it is important to differentiate between the architecture of ArchStudio itself and the architecture of the systems that are developed with ArchStudio. Components in ArchStudio’s architecture manipulate data about the architectural elements in the architecture of the system being developed with ArchStudio. To avoid confusion, we will always refer to the architecture shown in Figure 39 as the “ArchStudio architecture”.

157
implement a concern, then there is the possibility that the participants assume that all components in that layer are implement the same concern.

6.3.2. Setup

The development environment was set up to simulate the use of ArchEvol in the development of ArchStudio. The first step in our setup was to adapt the organization of the ArchStudio project’s source code into the component-oriented format that ArchEvol uses. Although the explicit architecture for ArchStudio was already in place, the large majority of the source code files were grouped together into the same Eclipse project. However, the architectural principles that drove ArchStudio’s development resulted in the code being organized into packages that corresponded to architectural components. Due to sharing the same ADL, the ArchStudio’s architectural model could be imported directly into ArchEvol. After the source code was split into individual component projects, these projects were linked using ArchEvol’s user interface features to the corresponding components in the architecture.

The next step in the setup process was to determine a set of high-level concerns that were relevant to the two participants and related to their prior work to ArchStudio. The main concept that characterizes the work that the two participants have done in ArchStudio is that of a change set. The concept of a change set denotes a new architectural construct, meant to group together a set of variation points in an architecture. Change sets are “first-class entities consisting of sets of closely-related additions, removals and (property) changes performed on the architectural description” [Hendrickson and van der Hoek 2007].

The change set concern constituted the first concern we used in populating the concern model. A number of components were identified as being related to the change set concern based on their names, which included the “changeset” keyword. The source code from these components was inspected and the fragments that were related to the change set concern were linked to the concern model. By following the uses of the various interfaces, classes, and methods that were among the linked source code fragments, it became clear that a number of other components were related to
the change set concern, and were included in the same process. A textual search in ArchStudio’s source code on other keywords such as “change” or “archcs” revealed further source code fragments to be included.

During this inspection process, a number of other keywords with relative high frequency suggested other concerns that were related to the change set concern. The concerns identified in this way were change set sync, change set event, detach, relationships, and explicit ADT. For each of these concerns, the source code was inspected based on keywords, definitions of interfaces and classes, and their uses. The resulting concern model has, for each of these concerns, source code fragments from a number of different components.

An overview of which components were related to the concerns is shown in Table 5. From the total of 46 components, the change set concern is implemented in a subset of 11. The other concerns were implemented in fewer components, with some overlap. The sets of concerns that are found in the Variability View, the Explicit ADT, and the xArchChangeSet concerns are all distinct. Even from the overview offered by Table 5, it can be observed that the concerns are both scattered over a subset of the components, and tangled together in each of these components.

6.3.3. Preliminary session

The change set concern and the associated sub-concerns are specific to a particular feature in ArchStudio, and therefore their implementation is limited to a small number of components. One of the limitations of this user study could be this particular nature of the change set concern, and the fact that its implementation is not scattered over a larger number of components. Another limitation is that the selection of the implementation fragments was performed by the author instead of being done by one of the experienced developers. Lack of clarity of the code or subtleties that only a developer knows make determining the rationale behind a source code fragment difficult, which might result in some fragments being missed or being wrongly categorized.
In designing the user study, we performed a preliminary session meant to test alternatives to the selected concerns and the source code inspection process. In this session, we asked one of the two ArchStudio developers to identify all components that are related to the user interface (UI) concern. We selected the UI concern because it is a high-level, general concern that is not related to any specific feature, and therefore has the potential of being scattered over a large number of components.

The data collected from this preliminary session is interesting by pointing out evidence related to scattering and tangling of the UI concern in ArchStudio. At the end of the identification process performed by the participant, the UI concern was identified in a total of 27 components. Figure 40 shows ArchStudio’s architecture with the components that are related to the UI concern highlighted.

Two key observations are immediately available when looking at the architectural diagram through ArchEvol. First, the UI concerns do not all reside at the same levels in the architecture. While an unwritten convention in ArchStudio suggested that user interface components should be placed on the lower levels of the architecture, it is clear that in practice this is not always the case. Some
components that have UI elements are dispersed throughout the diagram, and even one component that is placed in the lowest level has no user interface elements. The second observation is that, while in some components all files contained at least one source code fragment, most of the components had half or fewer than half of the files related to the UI concern, which indicates that these are not inherently UI components. These two observations indicate that the UI concern is both scattered throughout the system, and it is tangled in the majority of these components with other concerns.

The lessons learned from this preliminary study are two-fold. First, after performing this study, it became clear that the UI concern is too general to be used in the study, because it can potentially be implemented in any component. The second lesson is that having the developer inspect the source code and identify the concerns is extremely time-consuming, and the time of the study can be better used on asking the participants questions about the visualizations and the metrics shown by ArchEvol. As a result of these observations, we changed the format of the main study, in that we selected more specific concerns, sub-concerns of change set concern, and we identified the concern in the code in advance.

Regarding the architectural model used in ArchStudio, a valuable observation is that even in systems with an explicit architecture, as in the case of ArchStudio, and with informal guidelines and
conventions to assist in maintaining a “good” separation of concerns, such as the use of layers in the architectural diagram, we can still observe scattering and tangling of concerns. This is a confirmation that scattering and tangling of concerns are natural phenomena and they are present not only in systems that do not have an explicit architecture, but even in those that do.

6.3.4. Procedure
At the beginning of the study, the participants were asked to assess their knowledge of the implementation for each of the components in ArchStudio’s architecture. The different levels of knowledge that the participants could have about a component’s implementation was described as one of the following choices:

- **name only**: the purpose of the component was known from its name, description, documentation, or use of its interface, but the participants were not familiar with its implementation details;

- **changed code**: the participants occasionally changed the code of the component, and have some familiarity with its implementation; or

- **written**: the participants have partially or completely written the component, and they are intimately familiar with its implementation.

Although the participants might have written the source code for a component, a vital factor that influences their ability to comment about the implementation of the component is the length of time that passed since they saw the source code for that component. The assumption is that, if they changed a component a long time ago, the participants will not be able to recollect details with the same type of accuracy as for components that were changed more recently. In order to factor in the recency of changes in the study, the participants were asked to assess, as best as they could recollect, how long before the time of the study they worked on the implementation of each of the components. The scale used for the time interval was “yesterday”, “days ago”, “weeks ago” and “months ago”.

162
At the beginning of the study, the participants were asked to give their own definitions of each of the concerns that were used in the study. By requesting the participants to give their definition of these concerns, we ensured that the concerns used are meaningful concepts that the participants understand. Besides providing basic validation of the concerns used in the study, the definitions given by the participants can be used to see if the two participants share the same understanding of the meaning for each concern.

The rest of the study was split into three distinct phases. The first phase of the study asked the participants to use only their existing knowledge, helped by the original ArchStudio’s architectural diagram, to give their best estimates of where a number of concerns were implemented in the system, and also to give estimates for metric values for these concerns. In the second step, the participants were asked to use ArchEvol’s concern model and the visualizations on top of ArchStudio’s architectural diagram and to report the values that they observed. The third phase asked them to comment about the discrepancies between the data obtained in the first and second phase.

Each of these three phases has a different contribution to the study. The first phase gives us a baseline against which to compare the accuracy of the ArchEvol’s information. Since both of these participants are considered experts in relation to the change set concern implementation in ArchStudio, they are expected to be able to give fairly accurate assessments about where this concern and other sub-concerns are present. By also asking them to give estimates for the metrics used by ArchEvol, they would become more familiar with what these metrics mean and what their purpose is. The second phase of the study provides usability data about ArchEvol, especially its architecture-based visualizations. By using ArchEvol on the same problem that they had to solve themselves, the participants are able to compare and determine how much easier it is to use ArchEvol. For the last phase, we expected them to record observations about the study in general and about ArchEvol in particular.

There are three types of situations that might take place: the estimates that the participants give are
in agreement with the data shown by ArchEvol, the participants might give inaccurate estimates, or the data shown by ArchEvol is wrong. Information about any of these three situations is useful in evaluating the effectiveness of ArchEvol. The information in ArchEvol should be, in general, in agreement with the opinion of the participants, which means that the visualization is accurate, and we consider this to be the baseline from which our evaluation starts. Cases in which the participants give inaccurate estimates while ArchEvol shows correct information indicate that ArchEvol has good potential of being used by expert developers, and is considered a plus for our evaluation. If, in some cases, the information in ArchEvol is wrong, then we need to determine what is the cause of this discrepancy, and determine if it is something that could be avoided in the future.

During both phase one and phase two, the participants were asked a number of questions. The first set of questions asked participants, for each of the change set, change set event, detach, and explicit ADT concerns, to:

- give their estimate of which components in the architecture are related to the concern;
- give an approximate value, on a scale of 1 to 3, where 1 means “very little”, 2 means “some”, and 3 means “most or all”, of how much of a component’s implementation is related to the concern; and
- give an approximate value, on the same scale of 1 to 3, of how much of the concern’s total implementation is contained in each of the components.

The next set of questions asked the participants to identify components based on which concerns they implemented. More explicitly, the components had to be identified based on the following criteria:

- those that are related to both the user interface and change sets concerns;
- those that are related to both the change set event and change set sync concerns;
- those that are related to the change set sync, relationships and detach concerns;
- those that are related to both the detach and explicit ADT concerns;
- those that are related to only one of the concerns; and
those that are related to most concerns.

At the end of the study, the participants were asked to fill out a questionnaire about their experience. The participants were asked to evaluate how easy or difficult it is to identify concerns in code, to give estimates for metrics related to concerns, and to identify which components implement a set of concerns, both in situations where they were asked to answer the questions without ArchEvol and with using ArchEvol.

6.3.5. Observations

The user study provided us with data that leads to a number of observations. While the primary goal was to evaluate the use of ArchEvol’s architectural visualizations, the study provides evidence that concerns can be elements of a common vocabulary that developers use in describing the system on which they are working. We first analyze the definitions that the participants provided for the concerns that were used in the study, then look at how participants performed the tasks requiring them to reason about one concern or a combination of multiple concerns in the architecture.

**Concern meaning**

During the first part of the study, when the participants were asked to write their own definition of the concerns, both participants immediately identified all of the concerns as being important concepts that they have worked on. Even though, when preparing the study, these concerns were identified syntactically, by using hints in the source code names and keywords, it became very clear that the participants have an understanding of what these concepts mean and how are they used and implemented.

While both participants might share the same general understanding of what the concern means, the definitions that they gave leave room for interpretation on the details of what should or should not be included as being related to a concern. For example, the *change set event* is described as dealing with “events related to the selection of an active change set of the change in which change sets are applied” by one participant, with a note that events related to explicit change sets, detached events, or synchronization events should not be included. The other participant described the
concern as the feature that “is used to synchronize the changes that are made to an architecture document”, which is more generic and might or might not include the other types of events. The detach concern is described as support for “faster merging by hanging on to removed elements so that they can be re-added later” and “supporting the explicit concern” by one participant. The second participant gives a longer description of the rationale for a detach concern: “when an artifact is deleted from an ArchStudio editor, the artifact is not actually removed from the xADL2.0 document. Rather, the xADL2.0 document stores the details that a specific change set was involved in removing the component”. For the explicit ADT concern, one of the participants objected to the label in that the “ADT” part suggests that there is no user interface concern, which is the case if the concern comprises everything related to “explicitly visualizing the content of a change set”.

The main observation that we draw related to the relevance of concerns selected for this study is:

Observation 1. Developers share a number of domain concepts that form a common vocabulary, but their definitions of what those concepts represent can be (slightly) different.

The definitions that the participants gave confirm our initial assumptions about concerns being concepts shared by multiple developers, but which, without a link or representation in the code, might have different meanings. This motivates the need for a concern model, in which: (1) the definition of a concern can be found and edited, so that everybody has the ability to get to the same understanding, and (2) the concern is further defined by its associated source code fragments, making it even more clear what is or is not part of a concern. Developers might have different definitions of what a concern means, but they need to reconcile their differences by virtue of using the same concern model.

Evaluation of Concerns in Architecture

The following two stages in the study required the participants to evaluate: (1) concern relevance in a component, and (2) concern spread over components for four concerns and all relevant components in the architecture. In the first stage, the participants were asked to give these values by using only the architectural diagram. Then, in the second stage, they were asked to use ArchEvol
in determining the values. We did not require the participants to specify fixed percentages, although ArchEvol does show these. Instead, the participants used a 3-level scale, from (1) meaning “very little”, (2) meaning “some”, and (3) meaning “most or all” of a component’s implementation is related to a concern (for the first metric), or of a concern’s spread over the component (for the second metric). When using ArchEvol, it was left to the participant’s latitude to interpret the progress bar and assign one of these values.

The results for these two metrics are shown in Table 6, for concern relevance, and Table 7, for concern spread. For each of the four concerns that were analyzed, change set, change set event, detach, and explicit ADT, the answers for both participants are shown next to each other. For each participant, in turn, the answers are shown in two columns, the first column showing the results without ArchEvol, while the second column shows the answers from the evaluation performed with ArchEvol.

The results shown in these two tables support the observation that:

**Observation 2.** Participants were able to agree on the implementation of a high-level concern, but there were notable differences in the assessment of the smaller concerns.

One of the results that can be observed from the data shown in Table 6 and Table 7 is that the participants were able to determine which components implement the high-level change set concern. The only discrepancy, participant 2 identifying ArchEdit as one of the components that implemented the concern, was due to the fact that the feature was implemented in a newer version of the code than the version used in the study. Also, for the same change set concern, the participants were able to identify, in general, the correct values our two concern metrics. For the first metric (Table 6), only participant 1 identified XArchDetachADT critically different from the value shown by ArchEvol, while for the second metric (Table 7), participant 1’s results had 4 discrepancies, and participant 2’s results had 2.

Regarding the other three, smaller-level concerns, the results were very different. Not only were there major discrepancies between what the participants believed the answers should be and what
ArchEvol reported, but also there were also major discrepancies between the evaluations of the participants themselves.

None of the answers between the two participants were in complete agreement, for any of the three concerns. Their answers differed on which components implement which concerns, on how much of each of the components is related to a particular concern, and on how much of each concern is implemented in a given component. The only two answers that were in agreement were the fact that the xArchDetachADT implements the *detach* concern, and the ExplicitADT component implements the *explicit ADT* concern.

<table>
<thead>
<tr>
<th>Component</th>
<th>change set</th>
<th>change set event</th>
<th>detach</th>
<th>explicit ADT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P1</td>
<td>P2</td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>xArchChangeSet</td>
<td>most</td>
<td>most</td>
<td>most</td>
<td>most</td>
</tr>
<tr>
<td>VariabilityView</td>
<td>most</td>
<td>most</td>
<td>most</td>
<td>most</td>
</tr>
<tr>
<td>ChangeSetId</td>
<td>most</td>
<td>most</td>
<td>most</td>
<td>most</td>
</tr>
<tr>
<td>xArchDetachADT</td>
<td>most</td>
<td>little</td>
<td>most</td>
<td>most</td>
</tr>
<tr>
<td>Archipelago</td>
<td>little</td>
<td>little</td>
<td>little</td>
<td>little</td>
</tr>
<tr>
<td>xArchChangeSetSync</td>
<td>most</td>
<td>most</td>
<td>most</td>
<td>most</td>
</tr>
<tr>
<td>xArchCSParser</td>
<td>most</td>
<td>most</td>
<td>most</td>
<td>most</td>
</tr>
<tr>
<td>ChangeSetsViewer</td>
<td>most</td>
<td>most</td>
<td>most</td>
<td>most</td>
</tr>
<tr>
<td>ExplicitADT</td>
<td>most</td>
<td>most</td>
<td>most</td>
<td>most</td>
</tr>
<tr>
<td>ChangeSetIdRelationships</td>
<td>most</td>
<td>most</td>
<td>most</td>
<td>most</td>
</tr>
<tr>
<td>ChangeSetIdRelationshipManager</td>
<td>most</td>
<td>most</td>
<td>most</td>
<td>most</td>
</tr>
</tbody>
</table>
The third part of the study required the participants to identify where multiple concerns are implemented in the system. In order for the participants to think about these values independently from the first part of the study, the questions included some new concerns, different from any of the concerns evaluated in the previous part. As this question involved multiple concerns and combination of concerns, we did not ask the participants to evaluate how much of the implementation of each component is associated to the given combinations of concerns. Instead, we only asked them to mark whether the component is or not associated at all.

The results, aggregated in Table 8 show that for all of the combinations of concerns given, the
participants missed one or two components. In the table, this can be seen in the rows where only one of the participants identified the component as belonging to a concern, which is marked with an “x”. For the UI and Change set concerns, both participants correctly identified the VariabilityView and Archipelago components, but missed the xArchChangeSet component. For the change set event and change set sync, they identified the xArchChangeSet component, but not the other two. For change set sync, relationships, and detach, one of the participants correctly determined that there were no components supposed to address all three concerns, while the other participant mistakenly named two components. For the detach and explicit ADT, each participant named one component correctly, but each named a different one.

The result of this analysis shows that:

**Observation 3.** Participant’s ability to determine where concerns are implemented in a system, and to evaluate the tangling of concerns, deteriorates as the number of combined concerns increases.

In the case of the question that asked which component has the most concerns, they named the
xArchChangeSet component, but did not name the VariabilityView. This question was considerably more demanding than the previous ones, because it required the subjects to think about multiple concerns at once. The answer, too, was not the intuitive choice. Since the xArchChangeSet component is responsible for keeping the state of the entire change set feature implementation, and it is the main component where the change set concern is implemented, it is intuitive to name it as the component that addresses each of the concerns. The Variability View component is, in turn, the complementary component for xArchChangeSet, in that it provides the user interface elements for some of the concerns. However, neither xArchChangeSet nor VariabilityView address all of the concerns. One does not address the relationships concern, while the other does not address the detach concern.

Participant observations

The last part of the study asked the participants to document their experience, and to try to give an explanation for the inconsistencies between the initial evaluation for concern metrics and the values that they discovered by using ArchEvol.

A high-level observation about the study, given by participant 1, was that:

Observation 4. Thinking about where concerns are implemented in the system, and in which components, was a novel exercise for developers.

This is even though the participant was actively involved in architecting, designing, and implementing a large part of the changes related to the change set feature. The value that the participant saw in relation to using ArchEvol is that ArchEvol required the participant to take a step back and think about where concerns are implemented in the system.

Related to the questions about concerns, participant 1 mentioned surprise in seeing the values shown by ArchEvol. The participant gave comments in 6 questions. In 5 out of the 6 situations discussed, the participant agreed with the results of ArchEvol. In one of the answers, the participant expressed surprise about the metrics for the concern spread over components, but did not give an answer regarding which answer is more accurate. Causes mentioned for the other discrepancies
were: (1) omission of the component by overlooking it, (2) some implementation was not actually what the participant thought, and they would change the component’s implementation to match the initial estimate, (3) the participant was used to thinking about which components produce or consume a service, not which components address the concern (in which case, both the producer and the consumer would be linked to the concern), and (4) false expectations about the features that a concern should implement (the participant expected the UI to be implemented in the Explicit ADT component, but it was not).

An overarching observation stemming from the comments of the participants was that:

Observation 5. For an overwhelming majority of the answers, participants agreed with the results shown by ArchEvol, after verifying the facts by inspecting the source code.

Only 2 out of the 9 answers for participant 2 had circumstances that implied that the data from ArchEvol was not correct. In the case of the change set event, the participant argued that the value for the metrics should have been different, while in the case of the question about the explicit ADT concern, the participant worked on implementing the feature in one of the components mentioned, but the study used an older version of the source code from ArchStudio. In the remaining questions, the participant agreed with the results shown in ArchEvol. Among the causes mentioned for the discrepancies are omissions because the component was overlooked, assumptions about the component’s implementation that did not match the actual facts in the code, or misleading component names.

6.3.6. Conclusions

This study was performed with only two participants and targeted only one system. While these are valid limitations of the study, imposed by the constraints that we set forward during set up, there are a number of conclusions that we believe extend to a more general scenario.

The first of these conclusions is related to the participants using ArchEvol:

Conclusion 1. Participants proved that they used ArchEvol to visualize concerns and to
In performing the required tasks, the participants were able to visualize concerns using ArchEvol and to understand the metrics shown in the visualization. These tasks included both determining which components in the architecture implement a concern, discerning them from the rest of the concerns, determining values for the metrics, and determining scattering and tangling of concerns. As the data presented in the previous section shows, both participants provided the same answers about using ArchEvol (except in the case of the first question shown in Table 6, when participant 1 failed to identify one component). The screen recordings from the study showed that answering the questions by using ArchEvol was done in significantly less time than without it.

**Conclusion 2. The results of the study provide encouraging validations for the usefulness of the architectural features included in ArchEvol.**

The data collected from the study provides initial evidence that even experts in a system will not be able to reliably and accurately estimate how concerns are implemented in the system, or reason about their scattering and tangling in a precise manner. At the beginning of the study, the participants were asked to assess their knowledge about components in the system. While both participants were knowledgeable about the components discussed in our study, either from implementing them entirely or from implementing one part, the participants were still susceptible to give answers to the questions about concerns that were less precise than when they used ArchEvol.

In the large majority of cases, the participants agreed, upon verification with the code, that the results shown by ArchEvol were more accurate than the values that they initially estimated. The causes that the participants identified for the discrepancies were either outright omission of a component, or a misleading idea about what the implementation really does.

Furthermore, comments from participants show that ArchEvol provides a novel way to look at an existing architectural design, by forcing participants to think about scattering and tangling of concerns. However, the time limit for the study is too short to determine what types of actions, if any, would be inspired by the data shown in ArchEvol.
6.4. ArgoUML Evolution Study

The purpose of this study was to investigate the evolution of concerns and evaluate the possible effects of using the concern model over a longer time period. The previous two studies used a single version of the system being studied, frozen at a certain point during development. By contrast, this study experiments with a large number of changes that take place over the lifetime of a system.

As a candidate for the project being examined in this study, we selected the ArgoUML project [Argo/UML] ArgoUML is a popular UML editor, developed as an open-source project by a community of dedicated developers. As a result, the project was continuously evolved over a number of years, which makes it an ideal candidate for our experiment.

The experiment simulates the use of ArchEvol over a relatively long period of time. The first step is to select an initial version of the source code for ArgoUML and set up its source code in the format that ArchEvol needs. This means recovering its architecture, capturing it into an architectural diagram, and splitting the monolithic source code organization into individual projects that store the source code for the components. This is the same process that we already used in the ArchStudio case study discussed in Section 6.3. An initial concern model, with four concerns that we intend to trace over time, was created at this step by, once again, manually inspecting the code.

The second step of the experiment was to simulate the evolution of the system while using ArchEvol. This consists of applying, in order, source code patches from ArgoUML’s source code revision system. These patches, which are written using the unified diff format, each represent a set of transformations to the source code files, in terms of removal, addition, and replacement of text. These actions were applied as if a human developer were to type the changes in the Eclipse editors. In this way, the simulation is accurately imitating the actions of a developer, and leaves the concern model synchronization to perform as usual.

At specific revision numbers, we analyzed the resulting concern model, which was modified by the importing of changes, for the accuracy of its source code fragment associations to concerns. New

\[\text{1 The HTTP address for the ArgoUML repository is: https://jedit.svn.sourceforge.net/svnroot/jedit}\]
fragments that were not automatically captured were added to the concern model at these times, so that at each of these milestones the concern model is accurate and can be used for the next batch of change inputs. The status of the code, architecture, and concern model were saved as a new ArchEvol version. The milestones were selected to correspond to the main releases of ArgoUML. As shown in Table 9, ArgoUML had a total of 11 main revisions, and we determined the revision number at that release by matching the date of the release with the date of the revision number.

6.4.1. Recovery of architecture

In order to use ArgoUML with ArchEvol, we needed to create an architectural model of the system. As ArgoUML does not have an explicit architectural model, our decision of what the architectural model should be had to be based on information that we could find from either existing documentation or from the source code.

Two documents available from the argouml.tigris.org website discuss the architecture of ArgoUML in detail, but both these documents are out of date as they refer to older versions of the project and have not been regularly updated\(^1\).

The entire source code for ArgoUML is organized as a single project, and inside this project, the package organization provides hints that identify high-level concerns. For example, the package org.argouml.ui contains classes that are related to the UI concern, org.argouml.i18n contains source code related to the internationalization concern, and so on.

An important observation about the package names is that, in almost every case, package names indicate that a package addresses more than one concern. For example, the packages org.argouml.ui and org.argouml.uml.ui both have the UI concern in their name. At the same time, both packages org.argouml.uml and org.argouml.uml.ui contain implementation files related to the uml concern.

---

This observation is important because it provides evidence that crosscutting of concerns (albeit of high-level concerns) is a natural phenomenon that cannot be completely avoided. If we try to encapsulate the concern *UI* in its own component containing both packages org.argouml.ui and org.argouml.uml.ui, then the *uml* concern will be scattered in this and the component containing only the org.argouml.uml package. If, instead, we decide to create a component based on the *uml* concern, then the *UI* concern becomes scattered.

There are, therefore, different architectural views that could all be considered reasonable choices to represent the architecture of ArgoUML, based on how we decide to define what a component contains. We need, however, to split the content of the project among distinct components. Depending on how packages are assigned to each component, there would be different resulting architectures and mappings from components to implementation files. These choices then also influence the values of the concern metrics shown by ArchEvol.

In this experiment, we selected components based on the first relevant keyword from the package names (i.e. not “org” or “argouml”). In cases where there are multiple packages with the same first keyword, and some packages had a second keyword, we split the component into two components. For example, the “org.argouml.uml” package would be included in the “uml”

<table>
<thead>
<tr>
<th>Release</th>
<th>Date</th>
<th>Revision Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release 0.24</td>
<td>February 12th, 2007</td>
<td>12046</td>
</tr>
<tr>
<td>Release 0.22</td>
<td>August 8th, 2006</td>
<td>10972</td>
</tr>
<tr>
<td>Release 0.20</td>
<td>February 9th, 2006</td>
<td>9719</td>
</tr>
<tr>
<td>Release 0.18.1</td>
<td>April 30th, 2005</td>
<td>8085</td>
</tr>
<tr>
<td>Release 0.17*</td>
<td>January 19th, 2005</td>
<td>7555</td>
</tr>
<tr>
<td>Release 0.16.1</td>
<td>September 4th, 2004</td>
<td>6674</td>
</tr>
<tr>
<td>Release 0.16</td>
<td>July 19th, 2004</td>
<td>5631</td>
</tr>
<tr>
<td>Release 0.14</td>
<td>December 5th, 2003</td>
<td>5377</td>
</tr>
<tr>
<td>Release 0.12</td>
<td>August 18th, 2003</td>
<td>4608</td>
</tr>
<tr>
<td>Release 0.10.1</td>
<td>October 9th, 2002</td>
<td>2645</td>
</tr>
<tr>
<td>Release 0.9</td>
<td>September 4th, 2000</td>
<td>434</td>
</tr>
</tbody>
</table>

Table 9. ArgoUML revision history.
component, the “org.argouml.uml.cognitive” and “org.argouml.uml.cognitive.critics” would belong to the “uml.cognitive” component, and “org.argouml.uml.cognitive.critics.ui” would belong to the “cognitive” component, but because we used a first-keyword association, it will not be associated with the “critics” concern, which is in the second position after “cognitive”.

The resulting architecture had 25 components in Release 0.9 and 34 components in Release 0.24.

### 6.4.2. Simulation of the evolution

The main phase of our study consisted of the simulation of using ArchEvol throughout the evolution of ArgoUML. The goal of this simulation was two-fold. One was to determine, as accurately as possible, if ArchEvol could be used in a developing a project for a longer period of time and with what results; the other was to obtain, at the end of the simulation, data necessary for the visualization of the evolution of concerns over a long period of time.

The import of changes was done in a way that simulates as closely as possible the use of ArchEvol. Each change from Subversion is represented as a unified diff file. We transformed this diff file into individual text actions that were either insertion of some character at a specified position in a file, or removal of some sets of characters. Then, using the Eclipse API, we opened the file in Eclipse in the Java text editor, and performed these insertions and deletions. In this way, the changes were imported as if a real developer would have performed them.

**Import of revision patches**

The transformation of the patches into actions is the critical piece of functionality that allows us to perform this experiment, because it allows us to simulate the evolution of concerns without requiring manual effort in doing so. However, the transformation from the unified diff format to the list of text actions is not entirely accurate, due to the fact that the diff format is not precise in its description. The purpose of the unified diff format is to show the difference between two files in a precise textual format. The diff format is precise in that it describes which lines of text need to be removed and which ones need to be added from the first file in order to obtain a file identical to the second file. For this purpose, the granularity that the diff format uses, which is entire lines of code, is sufficient.
However, for our purposes, it is not.

In order for ArchEvol to preserve its source code fragment links, the granularity of the differences needs to be transformed from lines of text to characters. For example, in Figure 41, the diff file shows the effects of formatting the source code of a file. As a result, at line 64 in the file, the type of a variable was changed. The diff file format shows that line 64 was replaced with a new line of text.

Let us assume that ArchEvol tracks the content of line 64 as addressing a concern. If we were to follow the changes from the diff file on a line-by-line basis, then this fragment will be lost once line 64 is deleted. When the new line is inserted back, the link about the concern in this line of code will remain lost.

The solution that the ArchEvol experiment simulator uses is to transform the differences to a character-based diff. The transformation algorithm tries to detect cases in which the lines of text have small changes by looking at changes between a set of lines of text that were deleted immediately followed by a set of lines of text that were added. The algorithm first tries to match lines based on the Levenshtein distance [Levenshtein 1966] between all pairs of lines (a,b), where a is a deleted line and b is an added line. The lines from the deleted lines group are matched, in order, with the first line that has the minimum Levenshtein distance from the group of added lines, while preserving ordering. After the two lines are matched, a character diff searches for differences of characters that were inserted or removed at the beginning or the end of the line.

In the example from Figure 41, the unified diff format shows that the line with the number 197 was deleted and replaced with two lines, at positions 197 and 198. However, this change was due to a new line of code being inserted before the original line 197. Therefore, the old line with the number 197 was not replaced by the line with the number 197. The matching algorithm will correctly detect that, in this case, the old line 194 corresponds to the new line 198.

At fixed intervals in ArgoUML’s development timeline, we manually corrected the source code fragments for the watched concerns, and we created ArchEvol versions of the entire project. This allowed us to record and visualize the evolution of the concern metrics over the project’s timeline.
Reconciliation of inaccuracies in the concern model

One limitation of this study is that a developer using ArchEvol typically will associate different parts of a change that is recorded in the versioning repository as one revision change with one or more concerns from the concern model. A few factors made simulating such behavior impractical. First, it is very difficult to automatically determine to which concerns the code in a change – which sometimes is as large as hundreds of lines of code – belongs. A manual approach is equally impossible, due to the large number of changes in our simulation.

Second, a single change that is committed to the versioning repository commonly consists of multiple smaller changes, which may be unrelated to each other. The decision on how many conceptual changes to include in a single check-in, and whether to limit each submission to the repository to a single conceptual change or to group together multiple such changes, is a choice that developer made at the time when the change was submitted. A revision change might include code that adds a new feature, fixes a bug, and changes the format of the code, for example. By just looking at the changes in the versioning repository, it is very difficult to deconstruct the revision
changes to determine accurately which fragments of these changes are associated with concerns in ArchEvol. Given the scale of this experiment, and the large number of revisions, this option becomes virtually impossible to pursue.

In order to mitigate this problem, we have not associated whole changes to concerns. Rather, as these changes were imported, ArchEvol modifies the existing source code fragments accordingly. Then, at the end of each milestone revision, new fragments were added to the concern model. This was a manual process that included: (1) checking the existing concern model and verifying that ArchEvol modified the existing fragments correctly, and (2) adding new fragment links to the concern model.

The first part was performed by manually inspecting every source code fragment from the concern model, and determining whether the bounds of each fragment remains correct (for instance, if an entire method was linked to a concern, is the new fragment still containing the entire method, or is it pointing to a different fragment) or if the meaning of the code changed (the same method changed its meaning, and therefore should not be linked to the concern anymore). Any such errors were corrected.

For the second part, we used a helper tool that searched for keywords in the text of the source code, and determined whether the text around it was already linked to a selected concern. This greatly reduced the number of code inspections, as we focused only on new fragments (the old fragments were already inspected in the previous step). By using a combination of keywords and by inspecting the code around them, the concern model was augmented with new source code fragments for concerns.

After the new fragments were added, we refreshed the metric values for the entire project. Then, each milestone revision was saved as a revision in ArchEvol. Each of these ArchEvol versions contained our best approximation of how the code, the architecture, and the concern model would have looked at that point in time if ArchEvol would have been used in the development of ArgoUML.
Table 10. Number of fragments associated to the concerns at each milestone revision.

<table>
<thead>
<tr>
<th>Release</th>
<th>Concern</th>
<th>Number of Fragments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release 0.24</td>
<td>state</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td>diagram</td>
<td>320</td>
</tr>
<tr>
<td></td>
<td>association</td>
<td>270</td>
</tr>
<tr>
<td></td>
<td>actor</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>256</td>
</tr>
<tr>
<td>Release 0.22</td>
<td>diagram</td>
<td>513</td>
</tr>
<tr>
<td></td>
<td>association</td>
<td>382</td>
</tr>
<tr>
<td></td>
<td>actor</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>262</td>
</tr>
<tr>
<td>Release 0.20</td>
<td>diagram</td>
<td>445</td>
</tr>
<tr>
<td></td>
<td>association</td>
<td>436</td>
</tr>
<tr>
<td></td>
<td>actor</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>333</td>
</tr>
<tr>
<td>Release 0.18.1</td>
<td>diagram</td>
<td>406</td>
</tr>
<tr>
<td></td>
<td>association</td>
<td>443</td>
</tr>
<tr>
<td></td>
<td>actor</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>334</td>
</tr>
<tr>
<td>Release 0.17*</td>
<td>diagram</td>
<td>416</td>
</tr>
<tr>
<td></td>
<td>association</td>
<td>427</td>
</tr>
<tr>
<td></td>
<td>actor</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>293</td>
</tr>
<tr>
<td>Release 0.16.1</td>
<td>diagram</td>
<td>410</td>
</tr>
<tr>
<td></td>
<td>association</td>
<td>369</td>
</tr>
<tr>
<td></td>
<td>actor</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>288</td>
</tr>
<tr>
<td>Release 0.16</td>
<td>diagram</td>
<td>457</td>
</tr>
<tr>
<td></td>
<td>association</td>
<td>453</td>
</tr>
<tr>
<td></td>
<td>actor</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>329</td>
</tr>
<tr>
<td>Release 0.14</td>
<td>diagram</td>
<td>470</td>
</tr>
<tr>
<td></td>
<td>association</td>
<td>447</td>
</tr>
<tr>
<td></td>
<td>actor</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>295</td>
</tr>
<tr>
<td>Release 0.12</td>
<td>diagram</td>
<td>516</td>
</tr>
<tr>
<td></td>
<td>association</td>
<td>359</td>
</tr>
<tr>
<td></td>
<td>actor</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>248</td>
</tr>
<tr>
<td>Release 0.10.1</td>
<td>diagram</td>
<td>540</td>
</tr>
<tr>
<td></td>
<td>association</td>
<td>402</td>
</tr>
<tr>
<td></td>
<td>actor</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>302</td>
</tr>
<tr>
<td>Release 0.9</td>
<td>diagram</td>
<td>560</td>
</tr>
<tr>
<td></td>
<td>association</td>
<td>384</td>
</tr>
<tr>
<td></td>
<td>actor</td>
<td>23</td>
</tr>
</tbody>
</table>
The numbers of individual source code fragments associated with each concern during the experiment are shown in Table 10.

6.4.3. Data analysis

By simulating the use of ArchEvol over the entire development time of ArgoUML, this study provided data that can support the relevance of architectural metrics and visualizations in the evolution of a system. We have evaluated these two aspects by analyzing the concern model and the source code fragments that are related to the four concerns used in the study.

During the study, a large number of changes were imported into the editors of ArchEvol. Each of these changes was broken down to basic textual insertions and deletions. As these insertions and deletions were performed, ArchEvol updated the concern model by either removing or modifying existing source code fragments, using the algorithm described in the previous section.

We measured, while inspecting the concern model after each cycle of importing changes, the accuracy of the concern model. Figure 42 shows the percentage of fragments that were modified or added during the inspection process for a large part of the history\(^1\). The percentage of fragments that needed to be adjusted to reflect accurately the concerns was relatively small, with only three instances in which 15% more fragments needed to be added or modified. The observation related to this figure is that:

\[\text{Observation 1. The concern model remained, to a large extent, precise after the concerns were imported into ArchEvol.}\]

The reasons for why these fragments were missed vary, and can be attributed to the inherent limitations of the importing process or to errors in human judgment at the time when these fragments were assigned to concerns. As we discussed above, the importing process does not

\(^1\) For the few versions before Release 0.14, we used smaller cycles of import and fix of the model; we used those as a test before switching to larger import cycles.
attribute a concern to new code. A fragment of code will be added to a concern automatically if it is inserted inside or next to an already existing fragment associated with that concern. If the code is added in a remote, unrelated file, then it will not be associated at all, and was only added during the inspection process. Also, the algorithm for importing the unified diff files from the Subversion repository into ArchEvol uses a heuristic to transform it into insertion and deletion actions. These actions would, in some cases, replace a code fragment that was already associated with a concern with a slightly different code fragment that did not associate with the concern.

Overall, however, these two causes did not contribute to a large number of concerns being added or modified. A large number of fragments (varying in our experiment from 78% to 100%) remained the same or were modified correctly by ArchEvol. Based on this evidence we conclude that most of the concerns will remain stable in a real-development setting without requiring a considerable effort to maintain the concern model.

The second type of evolution data that our experiment provided was the evolution of the architecture of the system. One of the main benefits of using the versioning model of ArchEvol is the capability of storing and analyzing the architectural models throughout the development of a project. We used this feature to analyze ArgoUML’s architectural models stored throughout the versions imported in
ArchEvol. The evolution of the number of components and links between these components is shown in Figure 43.

*Observation 2. The architectural model changes very little throughout the evolution of ArgoUML.*

The number of components changes only very slightly. There are no major additions to the number of components after the Release 0.10. There is, however, a significant variation in the number of links among components. In our architectural model, a link represents a Java import between two Java classes. The links are uni-directional, and there can be two opposing links between any two components.

The picture of the evolution shown in Figure 43 shows a typical scenario, constituting evidence for the motivation of our approach. The architecture of the system, in this case, is determined largely at the beginning of development. The components and their roles are set up, and unless major changes are needed, the rest of development commits to preserving this initial architecture. One of the reasons for why the architecture is not changed more often might be precisely the fact that there is no architectural model in use. Without being able to visualize the architecture, the perceived
benefits of refactoring the system at a high level are more difficult to observe.

However, there are substantial changes that take place, and that are not visible as changes in the number of components. Hints for these changes are given by the variation in the number of links. As new features are added, and existing code is modified, new dependencies are introduced and old dependencies are removed, which is reflected in the number of links in the architecture. However, even this type of data does not indicate accurately what happens to the concerns that the system addresses. For this, we need to use the evolution visualizations of ArchEvol.

The first such visualization is the visualization of the architecture of ArgoUML. Figure 44 shows the architectural model at the last version that was imported into ArchEvol. There are two observations that can be made based on this visualization.

The first observation is that:

\textit{Observation 3. The architecture of ArgoUML contains an excessively large number of links between components, amounting to an almost complete graph.}

This, in general, denotes a low level of design quality from the perspective of the architectural organization. This is, however, not due to the quality of the code produced by developers. These two issues, the quality of the architecture and the quality of the code can be disconnected, because developers did not have a direct representation of the architecture available during development, which makes the code the daily focus of developers. During the evolution of a system, developers can introduce new dependencies between classes that, at the code level, would still preserve the quality of the code. However, these developers would be unaware that their architecture changed, and over time, the number of links can grow significantly.

The image shown in Figure 44 is representative for a large number of software systems that are similar to ArgoUML. We attribute this phenomenon to the lack of support for an architectural model to which developers can refer during development. ArchEvol provides such support, by integrating architecture and source code development, and by making the architectural view a permanent fixture in development.
Observation 4. The four concerns are scattered over a large number of components, covering almost the entire architecture, and at the same time they are tangled together in a significant fraction of the components.

Only few components have none of these four concern, or encapsulate exactly one. This denotes a low level of separation of concerns. Since in a real-world scenario developers would use a larger number of concerns, the scattering and tangling of concerns can only be worse. We attribute this phenomenon to the lack of awareness of scattering and tangling of concerns, exactly the type of problem ArchEvol is meant to address.

The evolution of concerns during the development of ArgoUML can be seen in ArchEvol’s History View. Figure 45 shows the History View when only one concern is selected, in two cases: one with

Figure 44. ArgoUML’s architecture.
the diagram concern selected, and another with the state_diagram concern selected. In both cases, the views are ordered by the values in the latest version of the system.

From Figure 45, it can be observed that most of the changes related to the concerns took place in the first half of the evolution of the system. The diagram concern had a number of sharp changes in its implementation. The most notable one is between revisions 0.16.1 and 0.17 in the kernel component. We used ArchEvol to determine the cause of this difference, by importing both the architecture and the “kernel” component at both revisions 0.16.1 and 0.17. The reason for the change in value is two-fold: (1) the number of classes in the component reduced from 15 Java classes to just 7, and (2) the number of files that contained the diagram concern keyword increased from 2 to 5. These two types of actions suggest that a design refactoring related to the diagram concern took place in between these two revisions.

The state_diagram concern also had a number of changes, even in more recent revisions. The fact that the concern was not found in revision 0.17 in components “notation” and “uml.notation” is due to the fact that these components did not exist before that revision.

Figure 46 presents a comparative evolution of two concerns: diagram and state_diagram in the first case, and state_diagram and association in the second. The evolution of the diagram and state_diagram concerns confirms intuitively a normal development path. The state_diagram concern is addressed mainly in the “diagram.state” and “diagram.activity” components, as both a state and an activity diagram in UML are types of state diagrams. The number of components, the values for the metrics, and the levels of tangling between these two concerns did not change significantly from the beginning of development.

For the second diagram, showing the state_diagram and annotation concerns, there are some differences. First, the component with the most variation was the internationalization component, which at revision 0.16.1 was taken out of the main development directory and moved to a separate resources directory. For component “diagram.activity”, for example, only the state_diagram concern was addressed, while at the last revision the association concern was addressed too.
Figure 45. Evolution history visualization for the diagram concerns (top) and the state machine concern (bottom).
Figure 46: Evolution history view showing diagram and state machine concerns (top) and state machine and association concerns (bottom).
6.4.4. Conclusions

The purpose of the ArgoUML experiment was to evaluate the feasibility of using ArchEvol on a real-world project for a longer period of time. This is the main difference as compared with the other two user studies, which focused on evaluating short-term changes. As the support for evolution of a software system is one of the key features of ArchEvol, the concern model and the architectural versioning features had to be evaluated.

Conclusion 1. ArchEvol can be used in development of a medium-size, real-world system.

The experiment tried to simulate as closely as possible the use of ArchEvol, by importing the history of the changes to ArgoUML’s codebase. This is not a substitute for a real human editing the source code, but it is a close approximation. The details of how the concern model can be maintained manually are explored in the jEdit user study. However, the results of the ArgoUML experiment show that the concern model maintains a high level of precision over time even without its default behavior and without any concern selection.

Conclusion 2. ArchEvol makes information about the concerns easily accessible.

Although the architectural model did not change significantly during almost the entire life of ArgoUML (having the same components and only the number of links varying), the architectural visualizations from ArchEvol can show where concerns are implemented and how they evolved. As we have shown, if developers observe a change in the values of the metrics shown by ArchEvol, they can easily go back and analyze the system in order to determine what caused that change.

6.5. Discussion

The three evaluations for ArchEvol covered different features of the environment and different situations that take place during software development. The user study looked at the process of using and maintaining the concern model while coding. The ArchStudio experiment analyzed the usefulness and the effectiveness of the architectural visualizations. The ArgoUML experiment tested the use of ArchEvol over an existing system over a long period of time. Together, these three
evaluations complement each other and provide a comprehensive evaluation of ArchEvol. The results of the user study show promising results for ArchEvol. In particular, it shows that the developers who used ArchEvol were able to perform their tasks better than the ones using Eclipse only and with the same level of success as the ones using FEAT. The subjects who used ArchEvol were also able to maintain the concern model with a success rate comparable to FEAT. The subjects who participated in the study liked the idea of having a concern model to help them navigate the code. Although the study was not large enough to derive statistically strong results, these are nevertheless encouraging results for ArchEvol.

The second experiment confirmed one of the main hypotheses of ArchEvol, that developers can be more aware of how concerns are scattered and tangled in a system by using architectural visualizations of concerns. The study showed that even experienced developers can have a difficult time in evaluating scattering and tangling of concerns, even in components to which they made significant code contributions. The architectural visualizations are not only more accurate, but also much more easier to use in determining scattering and tangling values for concerns. The results were in favor of ArchEvol, especially in the context of reasoning about the tangling of two or more concerns. Provided that the results of the first study hold, and that a concern model can be maintained accurately over time, this study shows that the data collected, aggregated, and visualized in the architectural model is useful and difficult to obtain through other means.

While the first two experiments asked the participants to use ArchEvol for a very short period of time (the duration of the experiment), the third experiment evaluates how ArchEvol could be used for a longer time. By importing the code changes from the versioning history of ArgoUML, this experiment showed two types of results. One, is a confirmation that ArchEvol could be used on a real project, and that the heuristics that ArchEvol used to maintain the concern model do work. A second type of result is unveiling that, even if projects such as ArgoUML do not use software architectural models, this does not mean that concern metric values do not change over time. The data collected from this experiment was used to show the visualization of the evolution of concerns in code.
7. Related Work

The fundamental problem in software development that underlines our work is the mismatch between the modularization of software systems and the conceptual separation of concerns in these systems. While separation of concerns was one of the earliest principles in software engineering, the problem of the separation of concerns over modularization has become more prevalent in the literature only relatively recently.

By and large, approaches that attempt to solve this problem can be categorized in two broad types. The approaches in the concern-based programming group treat the invisibility of concerns as an implementation problem. The solution proposed by this type of approaches is to change the underlying programming language in order to accommodate the crosscutting nature of concerns.

The second type of approaches take a different path, by leaving the implementation in place and modeling concerns as separate entities. The support provided by these approaches for managing explicit concerns range from offering concern models that hold the concerns, to methods of identification of concerns in code, to environment support for visualization of concerns. Some approaches focus only on one of these methods, while some offer a combination of methods.

Supporting the dichotomy between separation of concerns and modularization is only one of the contributions of ArchEvol. Another major contribution of ArchEvol is the use of architectural models to reflect the implementation of concerns in code, in order to make developers aware of their system design. Although there is a consistent body of research literature on explicit architectural models, the practice of actively using an explicit architectural model is not widely used in software development today. Research literature that addresses these issues focused on the mapping between architecture and implementation, and some even addressed the problem of modeling concerns at the architectural level.
7.1. Concern-based Programming

We grouped under this category approaches that aim to change the underlying programming model to adapt to the crosscutting nature of concerns. Aspect-Oriented Programming (AOP) is the canonical example, but other approaches exist. We discuss them here.

By and large, these approaches share the same goal with ArchEvol, but take a fundamentally opposed solution. While ArchEvol aims at keeping the modularization of the code as a choice of developers, and maintain a separate concern model, these approaches take the route of proposing new language mechanisms that can represent concerns and be composed together to form a software system.

**Multi-Dimensional Separation of Concerns**

The Multi-Dimensional Separation of Concerns (MDSOC) paper by Tarr et al. [Tarr et al. 1999] `was a seminal paper in signaling the problem of separation of concerns over a system’s implementation and on introducing the term “concern”. A key observation in this approach is the acknowledgment of the fact that formalisms, at all lifecycle phases, consider decomposition and composition of software along only one dimension. This phenomenon, called “tyranny of the dominant decomposition”, is one of the main motivations for the work behind ArchEvol. The solution proposed by MDSOC, however, takes a different path. MDSOC proposes a new method of decomposition and composition of software artifacts, where each concerns will be encapsulated in its own hyperslice, which is a set of modules that encapsulate a single concern. The units that are encapsulated in a hyperslice might belong to other hyperslices, making separation of overlapping concerns possible. Hyperslices are similar to program slices [Weiser 1984], but are more general since the units are not limited to source code implementation only.

The process of composing two or more hyperslices involves matching of concepts from the units in the hyperslice, reconciliation of differences, and integration of the units. The rules are similar to the ones in subject-oriented programming [Harrison and Ossher 1993, Ossher et al. 1996].

The composition needs to involve human judgment, since the problem of determining if two units
represent the same concept is undecidable, although it can use heuristics based on strict naming.

The MDSOC approach has been implemented in Java under the name of Hyper/J [Ossher and Tarr 2001].

**Aspect-Oriented Programming**

Aspect-Oriented Programming (AOP) [Kiczales et al. 1997] is a mechanism for encapsulating crosscutting concerns in special modules called aspects. Early implementations of AOP were domain-specific, such as the implementation of concurrency and distribution aspects that are implemented in the D language [Lopes 1997]. A more generic implementation of the AOP concepts was done in AspectJ [Kiczales et al. 2001].

AspectJ changes the execution flow of a regular program by inserting code when specific events in its execution take place. These execution times are called “join points” and a set of join points is expressed as a predicate expression written in a specific language. A code weaver inserts specific code, called “advice”, in the regular program at the join points, which can be, for example, when a method is executed, a field is accessed, or an object is created. A set of advices and their join points specifications, called “pointcuts”, can be grouped together in a new code construct called “aspect”.

Because AOP differentiates between a regular program and crosscutting behavior, it can only be successfully used to model some types of concerns. Recent evidence shows that most common concerns modeled with AOP and AspectJ in industry are debugging and instrumentation code, such as tracing, logging, testing, profiling, monitoring and asserting [Lopes 2002]. Other examples of concerns that have crosscutting behavior in any application domain are synchronization, parameter passing, persistence, and security.

**Composition filters**

Composition filters [Aksit et al. 1992, Bergmans and Aksit 2001] is an enhancement to the object-oriented programming model with a new type of construct named filter. A filter will interpose between an object and the rest of the program by capturing all the messages that the object sends or receives. If these messages match the criteria specified in the filter, then the code of the filter gets
executed. In this way, filters can be used to encapsulate crosscutting concerns. As in AOP, composition filters change the execution of a regular program to include additional functionality for crosscutting concerns.

**ArchEvol vs. Concern-based Programming**

The main difference between all these approaches and ArchEvol lies in the philosophical approach to concerns. MDSOC, AOP, and Composition filters invent programming constructs for crosscutting concerns, while ArchEvol leaves the organization of the source code at the liberty of the developer, but groups the code related to a concern together through the concern model. The concern modeling in ArchEvol aims to model more general concerns than AOP and Composition Filters, which only target crosscutting concerns.

### 7.2. Explicit Concerns

**Cosmos**

Sutton et al. define the Cosmos schema for concern modeling, which consists of categorizations of concerns, relationships between concerns, and predicates. Concerns are categorized as being either logical or physical, where the logical concerns represent conceptual “matters of interest” and the physical concern represents elements in a software system. Relationships between concerns can be categorical (which relate concerns based on their categories, such as “MemberOf”), interpretive (which relate concerns based on their interpreted semantics), physical (which relate concerns based on the physical relationships), and mappings (which relate a logical and physical concern). Predicates have the purpose of expressing consistency conditions between concerns, which can be evaluated in order to evaluate inconsistencies. While the Cosmos schema acknowledges the need for such predicates, they are undefined in Cosmos.

A taxonomy of concerns such as the one presented by Cosmos is outside the scope of ArchEvol. While providing functionality for managing concerns in the concern model, and tracing the source code fragments related to these concerns over the evolution of a system, ArchEvol does not make
any semantics assumption about the concerns. Concerns can have various sizes and can be at different levels of abstraction. In this regard, the concerns in ArchEvol are similar to the mapping relationship between logical and physical concerns in Cosmos.

**CME**

CME [Harrison et al. 2005] is a development environment that embraces the principle of promoting concerns to be first-class entities in software development, and provides tools for creation, manipulation, and evolution of aspect-oriented systems. An important principle behind the design of CME is the ability to model concerns in a neutral way with respect to the development approach.

CME uses a slightly different taxonomy than Cosmos in its ConMan schema, where the two types of concerns are identified as abstract concerns, which are concepts, and concrete concerns, which are physical elements. Examples of concepts are features, properties, or topics of interest, while physical elements can be software development artifacts, such as files, classes, and so on. The ConMan schema considers relationships between concerns and constraints on concerns as first-class elements in concern modeling.

CME influenced the early development of ArchEvol. The ConcernHighlight [Nistor and van der Hoek 2006] project was an extension of CME that highlighted source code fragments related to concerns from CME, and was one of the precursors to ArchEvol. The structure of the concern model in ArchEvol is a simpler version of the concern model in CME, and concern highlighting in code is only one of its features. The maintenance of the concern model during development, and the entire set of functionality related to software architecture are unique features of ArchEvol.

**Seesoft**

Seesoft employed one of the first visualizations of concerns over the structure of a software system [Eick et al. 1992]. An example of such a visualization, as taken from the AspectJ Development Tools website [AspectJ Development Tools 2007], is shown in Figure 47. Typically, a box represents a source code file, with stacked lines representing the lines of code in the file. Each line in the box is colored with the color of the concern that is linked to the corresponding line of code in the file. In this
way, the visualization can show which is the predominant concern, whether there are large contiguous fragments of code related to the same concern, and how many concerns are addressed in a file. The visualization in Seesoft became a de-facto standard used by the other environments: the AspectJ development tools, CME, and AspectBrowser.

The Concern Overview visualization of ArchEvol shows the same type of information, but on a single file, which is the file opened in the current editor. The Concern Overview can show, however, more information because it can show multiple concerns on the same line of code. The architectural visualizations in ArchEvol are also somewhat similar to the Seesoft visualization, but the main difference is that ArchEvol shows components, which contain multiple source code files, as high-level elements. The color-coding in ArchEvol is used for concern metric values, while in Seesoft it is used to indicate the physical location of the concern in the file.
Concern Graphs/FEAT

Robillard uses Concern Graphs to model concerns, and the definition of a concern is given in terms of its representation [Robillard and Murphy 2002]. This approach uses programming elements (classes, fields, and methods) and static or dynamic relationships between them to represent a program as a dependency graph. A Concern Graph is a “compacted subset of P documenting the implementation of a concern in P”, where P is the complete program graph. The set of elements is compacted by including only higher-level elements in the set if all their children are unambiguously associated to a concern. A concern is “a named collection of fragments”, where the term fragment denotes a relation between different program elements [Robillard 2003].

FEAT is the development tool that implements the Concern Graphs representation. FEAT is a tool for discovering and documenting concerns in code, implemented as an Eclipse plug-in. The central part of FEAT is a hierarchical concern model, to which the user can add Java program elements while navigating the source code. For each of these elements, the user can also invoke one of six predefined queries that will show what other elements in the code are related to it. The user has then the option to add all or any of these elements to the concern model, thus incrementally enhancing the concern model as the exploration of the code goes further.

ConcernMapper [Robillard and Weigand-Warr 2005] is a simpler version of FEAT, allowing the users to add methods and field references from Java programs to a concern model. ConcernMapper models concerns extensionally, as the set of the elements that are added to the concern model. In this regard, ConcernMapper has a similar functionality with ArchEvol in linking concerns in the concern model with an explicit set of source code fragments. However, in ConcernMapper, these fragments are Java lexical elements (classes, methods, variables), while in ArchEvol the underlying representation is textual.

As with CME, FEAT advocates the same idea as ArchEvol, which is the use of a concern model that structures concerns, is linked to source code fragments by the developer, and is maintained throughout development. While the focus of CME is on modeling the concerns and on creating a
general concern model, and the focus of FEAT is on helping developers understand the code and investigate an existing source code base, the focus of ArchEvol is on using this concern model throughout development to raise the awareness of developers about their design quality. Besides the concern model links to code, ArchEvol integrates with an architectural description to provide metrics and visualizations at a higher level of design than just code.

**AspectBrowser**

AspectBrowser [Griswold et al. 2001] is one of the first tools to allow modeling of separate source code fragments that belong to the same concern and support visualization of these concerns in the editor through highlighting. The metaphor used to motivate this environment is that of a map: the map of a city, for instance, shows an abstraction of the city, where the organization of the city is shown annotated with different points of interest. A concern has a color assigned, which is used to visualize concerns in both a Seesoft-like visualization and in the editor. The Seesoft-like visualization highlights lines of code belonging to a concern and folds away the files that do not contain a concern. In the editor, implemented in Emacs, source code related to a concept is highlighted with the assigned color. To find the source code elements belonging to a concern, AspectBrowser uses regular expressions, inspired from the functionality of the grep command. The highlighting of concern-related code is a similar feature to the one in ArchEvol. ArchEvol is different by using an explicit representation of concerns, and by managing the links of concerns to code throughout development.

**jQuery**

Another Eclipse plug-in based tool for navigation of source code based on concerns is jQuery [Janzen and Volder 2003]. In jQuery, the user can define queries over source code elements using a specific query language based on programming logic. The tool evaluates the query and displays the elements that match the query in a separate tree view. This view can then be incrementally enhanced by adding, for any of the displayed elements, other elements that are related to it. Like FEAT, CME and AspectBrowser, jQuery uses an implicit representation of concerns, by identifying all
the source code fragments through a query. ArchEvol takes a different approach by using an explicit representation of concerns. Also, ArchEvol uses the concern information to raise the awareness of developers to the higher level of abstraction that is the system’s architecture.

**CIDE**

CIDE [Kästner 2007, Kästner et al. 2008] is a tool that supports traceability of features in a Software Product Line (SPL) to source code. The CIDE Eclipse plug-in allows developers to associate elements in an Abstract Syntax Tree (AST) representation of the source code to colors. The source code textual fragments are then highlighted in the Eclipse editor, in a similar fashion to ConcernHighlight in CME and ArchEvol. The association of concerns (or features, in the terminology used by CIDE), is also used to hide from the editor source code parts that the developer considers irrelevant. By being based on the AST representation of the code, CIDE has the advantage as compared to ArchEvol in that source code fragments can be checked to include only complete elements (such as statements or methods). However, the advantage of fragment representation in ArchEvol is that a single text fragment can include a number of different, smaller, AST elements, which makes it more compact. CIDE is in an incipient phase, and can become a possible alternative to the mechanisms used by ArchEvol to map source code fragments to concerns.

**Mylyn**

Mylyn [Mylyn] is an Eclipse plug-in based on the Mylar project [Kersten and Murphy 2005, Kersten and Murphy Gail 2006] that monitors developer’s use of resources in the Eclipse workspace to determine a smaller, more focused set of resources relevant for a task at hand. Mylyn is similar to ArchEvol in that it identified the need for a task-oriented model in Eclipse, in which a series of development activities is related to a task. In Mylyn, this task can be designated by the developer, or imported from a bug management application, and then this task is used to learn, from the behavior of the developer, which code elements are related to this task. ArchEvol uses a similar listening mechanism for a different purpose, which is to record changes in source code editors in Eclipse in order to maintain the links from concerns to source code.
Aspect Mining

Aspect Mining is the research literature branch geared towards discovering where crosscutting concerns are implemented in an existing system’s implementation, using fully- or semi-automatic methods. The approaches in the Aspect Mining group are tangentially related to ArchEvol, because the result of the mining activity is a set of source code fragments that are related to a concern. Because in ArchEvol we designed the concern model to impose very few restrictions on the format of the fragments related to a concern, the concern model can be enhanced to be partially populated with the results of an aspect mining activity.

Some approaches look at clues or heuristics in the way programs are organized in order to identify methods that might qualify for refactoring as aspects. Marin et al. look at methods that have a high fan-in to identify aspect candidates [Marin et al. 2004]. Breu and Krinke [Breu and Krinke 2004] search for patterns in method call traces to identify aspect candidates. The results of investigating crosscutting concerns in Eclipse are presented in [Breu et al. 2006]. While these methods might successfully identify candidates for aspects, their analysis is inherently targeted at source code elements that are used from multiple places. Therefore, most of the concerns that do not exhibit a behavior above the set threshold will not be identified. This is a natural consequence of the fact that these methods are targeted at finding crosscutting concerns in regular methods whose rewriting as aspects would take advantage of AOP mechanisms.

Shepherd et al. use search for aspects using clues from natural language, working by assuming that the concern will be implemented using similar names [Shepherd et al. 2007]. This, and other methods that take into consideration comments, are based on the assumption that programmers adhere to strict coding standards.

However, in practice, evidence seems to support the thesis that the variability of real-world programming makes finding all places where a crosscutting concern is implemented impossible with just using automated methods [Bruntink et al. 2007]. A similar study evaluates different code clones detection techniques with manual documentation of concerns [Bruntink et al. 2004].
7.3. Architecture and Implementation

One of the main insights supporting ArchEvol is that software architecture, which usually tends to be designed at the beginning of the development of a system and then abandoned quickly after the implementation starts, can be used as support for visualization of concerns. In ArchEvol, the information about source code fragments related to concerns is aggregate and presented at the architectural level. Concerns, in our view, provide the link between architecture and implementation. A number of related approaches have been focused on the very same problem.

Source code generation

An earlier approach tried to use program synthesis to derive source code implementation from higher-level models. Earlier methods have tried to use formal transformations that could be proved correct [Moriconi et al. 1995]. More recent methods, such as MDA [Mukerji and Miller 2003] give high-level guidelines on how to achieve such transformations.

Approaches such as ArchJava try to sidestep the issue of code generation altogether by using a programming language to express both the architecture and the implementation. In this way, the architectural description and the implementation can be checked at the same time, using compilation checks. At the same time, the mapping between the architecture and implementation is implicit in that the source code belonging to a component is contained in the file where the component is implemented.

A different approach is used in Koala [Ommering et al. 2000, Ommering 2002]. Koala uses an architectural language based on Darwin to describe the architecture of a system where components can be interchangeable and be either hardware or software based. The source code for a software component is implemented in the C language, and components can be developed without knowing beforehand in which architecture they will be used. Special code modules will use C renaming macros to bind statically the implementation of components to their architectural interface.

Some ADLs have an associated implementation library that eases the implementation of an architectural model. However, this feature is highly dependent on the specific architectural language.
and of its description capabilities. UniCon [Shaw et al. 1995] offers a set of already implemented connectors that can be used, and a set of component types. Rapide [Luckham 1996] does not provide an implementation framework, but can instead annotate Ada programs with code that captures the flow of events between classes. These events are then analyzed against the architectural description.

A similar approach for the xADL 2.0 language is performed by its associated ArchStudio environment [Institute for Software Research, Oreizy et al. 1998]. ArchStudio can instantiate a component based on a main class in the component’s implementation that has to implement a certain interface. ArchEvol extends the xADL2.0 architectural language with new constructs that define concern-based metrics of components and connectors. The xADL2.0 language was chosen because of its ease of use and extensibility.

**ArchTrace**

A solution to keeping architecture and code mappings consistent over time is ArchTrace [Murta et al. 2006]. In ArchTrace, a component in an architecture can be linked to a number of different source code files. This allows for a larger degree of freedom by being able to map multiple components onto the same file. ArchTrace uses policies to determine how the mappings should be updated over time. These policies can determine if files should be added or removed from a component mapping, and are triggered before or after the files are submitted to a Subversion repository. A user can activate or deactivate a number of policies by using the ArchTrace user interface, which also offers a means for the user to see and check the current status of the mappings.

The mappings between architectural elements and source code files are implicit in ArchEvol, as each component is assigned its own Eclipse project. Any source code file that is added to the project becomes automatically part of the implementation of this component. Compilation dependencies between component projects are solved by Eclipse, because projects can be linked to each other. In this way, the need for continuously monitoring the links and the need for heuristics to determine
whether a new file should be assigned to a component or another, as it is the case with ArchTrace, no longer exists in ArchEvol.

**Architectural recovery**

A large number of approaches try to recover an architectural view from an existing implementation. This analysis is usually performed by tools, but assisted by the user in specifying the types of analyses to be used or in validating the results.

An early example of an architectural recovery tool is ManSART [Harris *et al*. 1995, Yeh *et al*. 1997], which uses an AST representation of a program to look for certain architectural styles. A similar solution was used in the FMAT tools [Fiutem *et al*. 1996a, Fiutem *et al*. 1996b], which use patterns in the organization of the AST to determine different components and connections. These patterns represent typical implementations for different architectural connections. For example, a shared file, shared variables, pipes or sockets will all be considered connections. The reliance on the AST representation of the systems, the lack of an architectural language, and the need for human intervention makes these approaches suitable for one-time architectural recovery but not for long term maintenance of an architectural view of a system.

However, the lesson learned from this and other approaches is that having an architectural description of the system is useful even after the system is developed. One of the most referenced examples in this regard is a study of the Linux kernel done by Bowman, Hold, and Brewster [Bowman *et al*. 1999]. The study recovered an architectural view of the Linux kernel, using a combination of sources, including documentation, directory structure and file names, control flow and data flow analysis. The resulting architecture uncovered far more connections than the active developers thought that these should be. This is an important case study in proving that the conceptual architecture is not the same as the implemented architecture. Examples of possible causes identified by Bowman *et al*. included functionality that was supposed to be implemented by only one component but ended up being implemented by multiple components, and additions of connections between components that did not respect the original architecture.
Since the architectural recovery process involves manual adjustments, Kazman et al. propose the Dali [Kazman and Carrière 1999] workbench, which includes storing the results of the recovery analyses in a common repository. The visualization of this data is performed using the Rigi tool [Müller and Klashinsky 1988]. Analysis of the data in the repository is a process similar to the previous methods, which look for patterns in the data collected [Guo et al. 1999].

Krikhaar et al. describe a method for recovering a hierarchical software architectural model from an existing implementation [Krikhaar et al. 1999, Krikhaar 1997]. In this method, software elements such as methods and files are assigned to components. Then, relations between lower level elements are recursively “lifted” to relations between their enclosing parents, and become relations between components.

Focus [Medvidovic and Jakobac 2006] is a recovery method that can be used incrementally, limiting the effort of recovery to only a changed part of the system. Components are identified by clustering classes in a UML diagram, either semi-automatically or manually. An idealized, conceptual architecture is also needed to guide the process.

Murphy et al. describe a method for identifying the inconsistencies between high-level models and source implementations [Murphy et al. 1995]. Based on an already existing conceptual architecture and mapping to source code elements, the reflexion model shows the differences between this intended model and the one recovered from the source code implementation. Three types of differences are highlighted: convergences are connections that conform to the initial model, divergences are the connections that exist but are not described in the initial model, and absences are connections that should have taken place, but do not.

7.4. Architecture and Concerns

Relatively recent interest in using aspect orientation at the architectural level has raised the question of how and why aspects should be used in architecture, which, in turn, led to a number of new architectural languages to be proposed. A discussion of the different possible ways to integrate
architecture and aspects is presented in [Batista et al. 2006]. While these and other efforts try to bring the same concepts from AOP and MDSoC to the architectural level, research on how to effectively combine architectural development and aspect-oriented programming is still immature. We discuss here the most relevant contributions to date.

**Early Aspects**

Early aspects is the term given to aspects that are identified at the requirements or architecture phases. At the architectural level, the approaches related to early aspects usually use the module, component-and-connector, and allocation of source code to components and connector views to represent a software architecture [Clements et al. 2002]. Architectural aspects is the name given by the early aspects community for crosscutting concerns at the architectural level, in a similar fashion to aspects at the programming level. A discussion of how to manually identify, capture, compose, and analyze early architectural aspects is presented in [Baniassad et al. 2006]. However, these instructions are vague, such as “we document aspects in the aspect view using the same language and notations that are used to describe the architecture’s corresponding non-aspectual parts”. ASAAM and other approaches try to describe a predefined set of steps, or decisions, that an architect must take in order to identify architectural aspects [Bakker et al. 2005, Tekinerdogan 2004].

The approaches in the Early Aspects group take a similar approach to AOP and MDSoC, but extend the use of concerns to the entire software lifecycle instead of source code only. Their focus is on modeling the architecture, and then design and implementation, based on concerns. The same main difference between AOP, MDSoC and ArchEvol applies for Early Aspects too, namely the fact that ArchEvol does not try to enforce a modularization of the system, and that ArchEvol was designed to support a very generic definition of concerns. Early aspects can be some of the concerns that developers might want to use in ArchEvol, but ArchEvol supports other concerns too, as long as developers can identify the source code elements that are related to it.
DAOP-ADL

DAOP-ADL [Pinto et al. 2002, Pinto et al. 2003] provides first-level aspectual components constructs that mimic the AOP modeling and weaving techniques by applying them to components instead of objects. DAOP is also a middleware platform, whose services are used by the components to interact with each other, and which applies the aspect code to the components. The joinpoints are defined as the times when events are sent between components and the aspects are called by the platform before or after an event is sent or received by a component. The platform uses named parameters to exchange data between aspects.

DAOP is a runtime middleware platform, which combines components and aspects at runtime. The difference between a component and an aspect is only related to the runtime composition rules, the platform-deciding if and when to call an aspect.

In essence, DAOP offers a new type of interaction, based on timing of events, to the usual event-based architectural interaction model in the same way AOP extended regular object-oriented programming. This solution requires that the crosscutting concerns be extracted from all components and encapsulated as an aspect. This might not be as simple as it sounds, since the code related to a concern might not have the exact behavior in all components in the system. For instance, a security concern might need to insert one type of functionality on the client side, and another type on the server side. This might lead to a variation of aspects based on the same concern, and to a large number of components and aspects that need to be managed together in an architecture. This complexity goes squarely against one of the main uses of software architecture, which is to provide a clear, simpler picture of the system.

Katara and Katz

Katara and Katz [Katara and Katz 2003] propose modeling of different architectural views that describe the aspects in an implementation, and which are then combined in an aspect-oriented way to form the whole architecture of the system. UML is used to define aspects as package stereotypes of type uses or definitions, and a concern diagram is used to show the various aspects in the system.
and their dependency on each other. In this approach, an aspect does not necessarily implement only one concern. Rather, a concern can become scattered over a number of aspects, while an aspect can implement the common implementation parts to two or more concerns. The aspects that are common to multiple concerns are specified explicitly. The approach is similar to the hyperslices approach from MDSOC in that an aspect is a view of the system that focuses on describing a single functionality in isolation from the others.

**Architectural MDSOC**

[Boucké and Holvoet 2006] propose a method inspired from MDSOC to address crosscutting concerns at the architectural level. The terminology here is different, in that an architectural concern is introduced to describe a concern that has significance at the architectural level, and even further, architectural drivers are sets of architectural concerns that drive the architect in defining an architecture. An architectural slice is used to capture the modeling of a single architectural concern. A slice composition diagram specifies how two slices can be bound together, using naming reconciliation between architectural elements in the same way that hyperslices are combined in MDSOC. This is also exploratory work, and there is still more research needed to identify the validity and the comprehensive nature of the language that defines the combination of architectural slices.

ArchEvol takes a different stance on the role of architecture modeling. In ArchEvol, developers are free to change the architecture of the system as they see fit. There is no architectural decomposition imposed at the beginning of development. Rather, developers can start with an architecture, and then observe how other concerns are scattered and tangled over this architecture. As developers move code around in the system to refactor its structure, the new scattering and tangling of concerns will be reflected over the same architecture.

**Architectural View Crosscutting**

A traceability model that uses an XML representation for different views and either XML links or XML queries to express crosscutting concerns between views is proposed in [Tekinerdogan et al. 2007]. Although this solution takes advantage of existing XML-based infrastructure to query the
architectural views and trace definitions that link concerns to architectural elements, there is no evaluation regarding how difficult it is to express a concern in this format or how difficult is to maintain the traceability links as described. Although this paper presents an example of using the traceability model, the work is still in incipient phase, and more validation is needed to prove its viability.

ArchEvol uses only one architectural view, the source code implementation view, in which each component is associated with its own source code implementation. The reason why ArchEvol focuses on this view is the need for clear mappings between architecture and implementation that make the aggregation of concern metrics at the architectural level possible. The general approach described in ArchEvol could be extended to accommodate other architectural views, with the condition that a link to source code concerns can be defined.
8. Contributions and Future Work

The work presented in this dissertation contributes to pushing further the software engineering field in making concerns central elements throughout software development. This starts with our vision and approach to making concerns explicit elements of software development, that can be used to make developers aware of how the design of their system evolves over time. We implemented and presented here a prototype that embodies this approach, the ArchEvol development environment. Both the approach and the environment were motivated and evaluated by experiments and user studies.

Although concerns are accepted as important concepts in software development, state-of-the-art development practices today do not have concerns expressed explicitly. Rather, they are implicitly encoded in different parts of the implementation. Our approach addresses this very problem. We proposed here a new software development paradigm, where concerns become central in software development. Although there are similar approaches that recognize the importance of concerns, and of modeling them explicitly during development, our approach is unique in its focus towards software evolution management. Concerns not only have to be identified in code, but they need to be actively maintained throughout development, as the software system evolves.

A second contribution of our research, which is unique to the work presented here, is our focus on using concerns to evaluate a system’s design. We believe that concerns should not only be linked to source code, but they should also play a major role in the design of a system. Our approach outlines an environment where software architectural models of a system are enhanced to show where concerns are implemented as well as their levels of scattering and tangling over the system’s main components. By presenting this information to the developer, and by continuously maintaining it up-to-date while the system evolves, developers are made aware of how their system’s design evolves.

This is a major shift from today’s development environments, where developers do not have a
concrete depiction of the system’s design and are unable to be accurate in their evaluation of a system’s design.

A contribution stemming from the two mentioned above is the shift of the role of software architecture and the use of architectural models that our approach advocates for. The overwhelming majority of software projects today do not use an architectural model. We believe that one of the reasons for this situation is the fact that, once development of a system passes the initial high-level design phase, architectural models as they have been presented in the existing literature offer little leverage to developers. The high-level architecture of a system is unlikely to change dramatically over time. Our approach, by showing concern metrics onto the architectural model, transforms the architecture from a mostly static depiction of the system to a dynamic one. The structure of the architectural model might not change, but the concern information shown in it changes with every little source code change. Every time a small source code fragment is added to the implementation of a concern, the architectural model will be updated to show the new values for the scattering and tangling of that concern. This is a contribution that could potentially have a major impact on how architectural models are used in software development, because it provides a reason for why architectural models should be continuously maintained, updated, and analyzed by developers throughout development.

The prototype implementation of our approach, ArchEvol, shows how a development environment can become concern-driven. We enhanced the regular Eclipse platform with a number of tools that help developers express the main concerns in the system, listen to changes in the source code editors to update the concern-to-source code mappings, provide different methods that allow developers to enhance and maintain these links, integrate architectural descriptions of the system with the source code implementation, continuously update and present concern-based information onto the architectural model, and maintain the concerns, source code and architecture links throughout the evolution of a system. ArchEvol is unique in providing these features and integrating them into a coherent development environment.
The experiments and case studies that were used to motivate and evaluate our approach contribute important initial data about its relevance and effectiveness. We presented as support for motivating our approach an experiment that uncovered the levels of scattering and tangling for thirty-two concerns in a medium-sized software system. The results of this study are relevant because they show just how scattered and tangled concerns can be in a software system. Although other similar studies exist, very few of them had the same focus of comprehensively linking concerns to source code, and most of them only looked at a few concerns. Our study motivates the need for an environment where concerns are made explicit.

The two user studies presented in our evaluation provided data to support the feasibility and usefulness of our approach. The first one, the jEdit study, evaluated the source-code related features of ArchEvol, and showed initial results that indicate that ArchEvol matches the effectiveness of other existing, similar approaches. The second study, the ArchStudio study, shows initial results that indicate that the concern-based metrics presented by ArchEvol are useful and effective in providing information about where concerns are implemented in a system.

Our last experiment was a simulation of using ArchEvol in a long-running development effort. Besides providing data that could be used to showcase ArchEvol’s features for maintaining and visualizing the evolution of concerns over a system, this experiment showed that, for a typical development system, the concerns could be maintained without significant extra effort required from developers. This is an encouraging result that lets us to believe that ArchEvol could be used in the development of a system over a long period of time.

The work presented in this dissertation opens up a number of different questions that need to be addressed in future work. One of the main goals of our approach, implementation and evaluation, was to seamlessly integrate a number of different features related to concerns in an overall development environment. The focus of our research was, therefore, more on breadth than on the depth of each of these features.

Some of the problems that further work can address, and that our research enables, are related to
different mechanisms of managing and using concerns during development. We presented a concern model and two methods of maintaining the concern model links to source code, but further improvements should be researched. For example, annotations or other source code markers could be used to link source code to concerns. Determining how developers might prefer these mechanisms in addition to the ones provided by ArchEvol is an interesting research question.

In our experiments, we used a number of different concerns that varied by their conceptual generality, their number of source code fragments, the size of these fragments, and their corresponding scattering and tangling. While ArchEvol was designed to accommodate all these types of concerns, it would be interesting to observe whether there are specific types of concerns that developers are inclined to manage in their systems.

We also presented an approach to maintaining software architecture, concerns, and source code mappings during development. Future research efforts could evaluate whether this method is applicable to all software projects, or that future improvements are necessary.

Our aggregation of source code data to architectural components only showed one type of metrics that were based on file-level aggregation. Further efforts are needed to determine whether other types of aggregation, for instance at the granularity of lines of code, or programming elements, would be useful.

Future work can also address the limitations of our prototype, some of which are germane to all prototype implementations in general. Issues that still need to be addressed include speed improvements and optimizations, extensive testing, and recovery in the events of a crash. Speed improvements and optimizations are important in the event-tracking part of ArchEvol, where every source code change is listened to and acted upon, to make ArchEvol more responsive and not slow down the developer. The current prototype works for tens of concerns, and includes optimizations such as file-based indexing of the source code fragments, but further improvements could make it possible to use ArchEvol for tracking larger orders of magnitude more concerns and source code fragment links.
Finally, future changes to ArchEvol should address the usability of the environment from a human-interaction perspective. Visualization is an integral part of ArchEvol: a graphical model of the architecture is an inherent component of using an architectural description, but to that ArchEvol adds visualization of concerns; at the same time, concerns are visualized during development, in the source code editor, to make the users aware of: (1) where concerns are implemented in a single source code file, and (2) how many concerns currently overlap in the active source code fragment being edited. In all these visualizations, color-coding is the main mechanism through which the concern information is shown to the user.

ArchEvol could benefit from existing human-interaction related literature to improve the effectiveness of its visualizations. Questions related to mechanisms of choosing colors for concerns that are easily distinguishable from each other, of ways to show concerns in the architectural model that would be easier to find and interpret by the developers, of providing intuitive highlighting of concerns, and of mechanisms of showing information that would attract the user’s attention in an unobtrusive manner could be addressed by future work.

Finally, more users studies are needed to evaluate the effectiveness of our approach, especially as compared to related approaches. Our studies and experiments provide only initial data in this respect, but they are not statistically significant. More rigorous experiments that look at a larger number of features, and employ larger numbers of subjects, are needed.
REFERENCES


85. Medvidovic, N. and Taylor, R.N. A Classification and Comparison Framework for Software


126. Weiser, M. Program Slicing. IEEE Transactions on Software Engineering. SE-10(4), p. 352-

APPENDIX A: Questionnaire used in jEdit user study

I. Background

1. What is your degree level?
   □ Undergraduate
   □ Graduate

2. Please rate your experience level in Java.
   □ Novice
   □ Beginner
   □ Proficient
   □ Experienced

3. Have you used Eclipse before? If yes, for how long?
   □ Yes, for ____ years/months
   □ No

4. Have you used a configuration management system before?
   □ CVS
   □ Subversion
   □ ClearCase
   □ Visual Source Safe
   □ Other __________

5. Do you have experience with developing large scale systems? If so, please describe your role and the approximate size of the system.
   □ No
   □ Yes

____________________________________________________________________________________

____________________________________________________________________________________

____________________________________________________________________________________

222
II. Overall Impressions

6. Do you think that having a concern model that keeps references to where those concerns are implemented in the source code is, in general, useful?
   - Definitely not useful
   - Not useful
   - Useful
   - Somewhat useful
   - Definitely useful

7. How useful do you think ArchEvol is during software development?
   - Definitely not useful
   - Not useful
   - Useful
   - Somewhat useful
   - Definitely useful

8. Did ArchEvol lead to you taking more considerate decisions on where in the system to place the additional functionality? Explain.
   - Yes
   - No
9. Overall, do you think that the use of ArchEvol would lead to other developers managing concerns in code better or worse than through a typical Eclipse environment?

- Worse
- Slightly worse
- The same
- Slightly better
- Better

10. Do you think that ArchEvol can be successfully applied to a real-world development setting? Explain.

- Yes
- No

11. If you were employed in a software development company, would you recommend using ArchEvol to your team or manager?

- Yes
- No
12. Would you like to download ArchEvol and use it in your next development project?
   □ Yes
   □ No

13. Do you have any suggestions on how ArchEvol can be improved, or can you point out features that you think should be included in ArchEvol?
III. Task Related Questions

14. Rate the overall difficulty of the tasks. Explain your reasons.
   - Very difficult
   - Difficult
   - Fair
   - Easy
   - Very easy

15. Do you think that the concern model is useful?
   - Yes
   - No

16. How difficult was it to keep references to your changes in the concern model?
   - Very difficult
   - Difficult
   - Fair
   - Easy
   - Very easy
17. During source code editing, the various concerns that are traced in the file are shown with colors. Rate how useful you think highlighting concerns in code is. Explain why.

☐ Not useful
☐ Useful
☐ Very useful

18. During source code editing, the various concerns that are traced in the source code file are also shown on the left side in the Overbar View. Rate how important do you think that the Overbar View is.

☐ Not useful
☐ Useful
☐ Very useful
I. Initial setup

The experiment focuses on a number of concerns pertinent to the ArchStudio application. These concerns have been identified based on keywords from package names, classes and interface names, and comments in the source code. Please describe below, in a short definition, what your understanding of what each of the following concerns is for you:

User Interface (UI):

Change set:

Change set sync:

Change set event:

Detach:

Relationship:

Explicit:
II. Evaluation of concerns based on architecture diagram only

With only using your existing knowledge about the system, and the architectural diagram as a guide, please check which of the following statements are true, and give, where requested, your best approximation of the values asked for.

1. For the Change sets concern:
   a) Which components or part of components have functionality that is related to the concern? In other words, if you were to remove or change the functionality that implements this concern, which components would be affected?

   While each of the components above implement the Change sets concern, they also implement other concerns too.

   b) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), for each of the components above, of how much of the component's implementation is related to the Change sets concern? In other words, if you were to remove or change the functionality that implements this concern, how much do you think that you will need to change from each component?

   From the components that implement the Change sets concern, some contribute more than others to the implementation.

   c) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), of which components contribute more to this concern’s implementation. In other words, if you were to remove of change the implementation of the concern, in which concerns would you have to do the most work, compared to the others?

<table>
<thead>
<tr>
<th>Component (a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. For the *Change set event* concern:

a) Which components or part of components have functionality that is related to the concern? In other words, if you were to remove or change the functionality that implements this concern, which components would be affected?

While each of the components above implement the *Change set event* concern, they also implement other concerns too.

b) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), for each of the components above, of how much of the component’s implementation is related to the *Change set event* concern? In other words, if you were to remove or change the functionality that implements this concern, how much do you think that you will need to change from each component?

From the components that implement the *Change set event* concern, some contribute more than others to the implementation.

c) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), of which components contribute more to this concern’s implementation. In other words, if you were to remove or change the implementation of the concern, in which concerns would you have to do the most work, compared to the others?

<table>
<thead>
<tr>
<th>Component (a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. For the Detach concern:

a) Which components or part of components have functionality that is related to the concern? In other words, if you were to remove or change the functionality that implements this concern, which components would be affected?

While each of the components above implement the Detach concern, they also implement other concerns too.

b) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), for each of the components above, of how much of the component’s implementation is related to the Detach concern? In other words, if you were to remove or change the functionality that implements this concern, how much do you think that you will need to change from each component?

From the components that implement the Detach concern, some contribute more than others to the implementation.

c) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), of which components contribute more to this concern’s implementation. In other words, if you were to remove or change the implementation of the concern, in which concerns would you have to do the most work, compared to the others?

<table>
<thead>
<tr>
<th>Component (a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. For the Explicit concern:

   a) Which components or part of components have functionality that is related to the concern? In other words, if you were to remove or change the functionality that implements this concern, which components would be affected?

   While each of the components above implement the *Explicit ADT* concern, they also implement other concerns too.

   b) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), for each of the components above, of how much of the component’s implementation is related to the *Explicit ADT* concern? In other words, if you were to remove or change the functionality that implements this concern, how much do you think that you will need to change from each component?

   From the components that implement the *Explicit ADT* concern, some contribute more than others to the implementation.

   c) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), of which components contribute more to this concern’s implementation. In other words, if you were to remove or change the implementation of the concern, in which concerns would you have to do the most work, compared to the others?

<table>
<thead>
<tr>
<th>Component (a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
III. Questions related to relationships between concerns:

Which components have implementation related to both the User Interface and Change sets concerns?

Which components have implementation related to both the Change set event and Change set sync concerns?

Which components have implementation related to Change set sync, Relationships and Detach concerns?

Which components have implementation related to Detach and Explicit ADT?

Are there components that have only one concern, from the ones enumerated at the beginning of the study?

Which component is related to the most concerns?
IV. Evaluation of concerns using ArchEvol

At this step, you will use ArchEvol to determine the same values. Use the concern selection feature from the concern model, and the visualizations from the architectural diagram, and write down the values as reported by ArchEvol.

1. For the Change sets concern:
   a) Which components or part of components have functionality that is related to the concern? In other words, if you were to remove or change the functionality that implements this concern, which components would be affected?

   While each of the components above implement the Change sets concern, they also implement other concerns too.

   b) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), for each of the components above, of how much of the component’s implementation is related to the Change sets concern? In other words, if you were to remove or change the functionality that implements this concern, how much do you think that you will need to change from each component?

   From the components that implement the Change sets concern, some contribute more than others to the implementation.

   c) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), of which components contribute more to this concern’s implementation. In other words, if you were to remove of change the implementation of the concern,

<table>
<thead>
<tr>
<th>Component (a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. For the _Change set event_ concern:

a) Which components or part of components have functionality that is related to the concern? In other words, if you were to remove or change the functionality that implements this concern, which components would be affected?

While each of the components above implement the Change set event concern, they also implement other concerns too.

b) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), for each of the components above, of how much of the component's implementation is related to the Change set event concern? In other words, if you were to remove or change the functionality that implements this concern, how much do you think that you will need to change from each component?

From the components that implement the Change set event concern, some contribute more than others to the implementation.

c) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), of which components contribute more to this concern's implementation. In other words, if you were to remove or change the implementation of the concern, in which concerns would you have to do the most work, compared to the others?

<table>
<thead>
<tr>
<th>Component (a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. For the Detach concern:
   a) Which components or part of components have functionality that is related to the concern? In other words, if you were to remove or change the functionality that implements this concern, which components would be affected?

While each of the components above implement the Detach concern, they also implement other concerns too.

b) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), for each of the components above, of how much of the component’s implementation is related to the Detach concern? In other words, if you were to remove or change the functionality that implements this concern, how much do you think that you will need to change from each component?

From the components that implement the Detach concern, some contribute more than others to the implementation.

c) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), of which components contribute more to this concern’s implementation. In other words, if you were to remove or change the implementation of the concern, in which concerns would you have to do the most work, compared to the others?

<table>
<thead>
<tr>
<th>Component (a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. For the Explicit ADT concern:

a) Which components or part of components have functionality that is related to the concern? In other words, if you were to remove or change the functionality that implements this concern, which components would be affected?

While each of the components above implement the Explicit ADT concern, they also implement other concerns too.

b) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), for each of the components above, of how much of the component’s implementation is related to the Explicit ADT concern? In other words, if you were to remove or change the functionality that implements this concern, how much do you think that you will need to change from each component?

From the components that implement the Explicit ADT concern, some contribute more than others to the implementation.

c) Give an approximate value, on a scale of 1 to 3 (1=very little, 2=some, 3=most or all), of which components contribute more to this concern’s implementation. In other words, if you were to remove or change the implementation of the concern, in which concerns would you have to do the most work, compared to the others?

<table>
<thead>
<tr>
<th>Component (a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
V. Questions related to relationships between concerns:

Which components have implementation related to both the User Interface and Change sets concerns?

Which components have implementation related to both the Change set event and Change set sync concerns?

Which components have implementation related to Change set sync, Relationships and Detach concerns?

Which components have implementation related to Detach and Explicit ADT?

Are there components that have only one concern, from the ones enumerated at the beginning of the study?

Which component is related to the most concerns?
VI. Observations

Please detail your observations about the use of ArchEvol, and explain the discrepancies, if any, between the answers from Section 2 (without ArchEvol) and Section 3 (with ArchEvol).

General observations

For the *Change sets* concern:

For the *Change set event* concern:

For the *Detach* concern:

For the *Explicit ADT* concern:

For the questions related to relationships between concerns:
   For question 13
   For question 14
   For question 15
   For question 16
   For question 17
   For question 18