



W&M ScholarWorks

Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

2005

An Investigation of Programmer-Identified Concerns

Meghan Kathleen Revelle

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>

 Part of the [Computer Sciences Commons](#)

Recommended Citation

Revelle, Meghan Kathleen, "An Investigation of Programmer-Identified Concerns" (2005). *Dissertations, Theses, and Masters Projects*. Paper 1539626844.

<https://dx.doi.org/doi:10.21220/s2-sg2p-8g53>

This Thesis is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

AN INVESTIGATION OF PROGRAMMER-IDENTIFIED CONCERNS

A Thesis

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Master of Science

by

Meghan Revelle

2005

APPROVAL SHEET

This thesis is submitted in partial fulfillment of

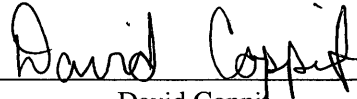
the requirements for the degree of

Master of Science




Meghan Revelle

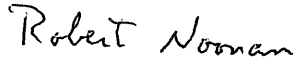
Approved by the Committee, April 2005



David Coppit
Thesis Advisor



Weizhen Mao



Robert Noonan

To my friends for all their support.

Table of Contents

Acknowledgments	vii
List of Tables	viii
List of Figures	x
Abstract	xi
1 Introduction	2
2 Case Study Methodology	6
2.1 Goals	6
2.2 Approach	7
2.3 Analyses	9
2.3.1 Concern Overlap	10
2.3.2 Concern Abstraction	11
2.3.3 Scatter and Spread	12
3 Case Study #1: GNU sort	14

3.1	Concerns Identified	15
3.2	Concern Overlap	17
3.3	Concern Abstraction	19
3.4	Spread	23
4	Case Study #2: Minesweeper	26
4.1	Concerns Identified	27
4.2	Identification Process	29
4.3	Concern Overlap	30
4.4	Concern Abstraction	32
4.5	Abstraction Level and Concern Overlap	35
4.6	Spread and Scatter	36
5	Generalizing the Case Studies	39
5.1	Factors in Agreement Among Programmers	39
5.2	Types of Concerns	42
5.3	Guidelines	43
6	Related Work	49
6.1	Identifying Concerns	50
6.2	Finding Concern Code	51
7	Evaluation and Conclusion	54
A	Illustrations of Guideline Usage	57
A.1	Guideline 10	57

A.2	Guideline 11	58
A.3	Guideline 12	58
A.4	Guideline 13	59
A.5	Guideline 14	59
A.6	Guideline 15	60
A.7	Guideline 16	60
	Bibliography	61
	Vita	63

ACKNOWLEDGMENTS

I cannot claim sole responsibility for this thesis. Yes, I wrote all the words, but I never could have written them without the help of a few key people. I wish to express my appreciation and gratitude to my advisor, David Coppit, for guiding me along the way and making the path smooth, for the most part. I am indebted to Tiffany Broadbent for being my co-investigator, Justin Manweiler for implementing needed features in Spotlight in such a speedy manner, and Robert Painter for being my sounding board. I also need to thank Lee Carver and Bill Griswold for making their concern information available and Elisa Baniassad for taking the time to give me feedback on an earlier version of this work. Also thanks to Andriy Fedorov for volunteering (unprompted) to read this thesis and give me comments.

List of Tables

3.1	Concerns identified by us and Carver and Griswold in GNU sort.	16
3.2	Concern overlap for characters and lines for concerns identified by us and Carver and Griswold in the sort case study.	18
3.3	Character concern overlap for groups of concerns in the sort case study.	21
3.4	Character concern overlap for groups of concerns in the sort case study.	22
3.5	Spread of commonly identified concerns in sort.	24
4.1	Concerns identified by Investigators M and T in the Minesweeper program.	28
4.2	Concern overlap for characters and lines for concerns identified by both investiga- tors in the Minesweeper case study.	31
4.3	Character concern overlap for groups of concerns in the Minesweeper case study.	33
4.4	Scatter of concerns among the methods of the Minesweeper classes.	38

List of Figures

2.1	In the Spotlight editor, the programmer is using the context menu to annotate part of the code with a concern.	8
3.1	Two different taggings of an instance of the <i>Unique</i> concern.	19
3.2	Concern overlap vs. average spread for commonly identified concerns in sort. . .	25
4.1	A screen shot of the Minesweeper game.	27
4.2	Hierarchy of concerns in the Minesweeper program.	34
4.3	Concern overlap vs. abstraction level for concerns identified by both investigators in the Minesweeper study.	35
4.4	Concern overlap vs. number of classes for concerns found by both investigators in the Minesweeper study.	37
5.1	Concern Identification Guidelines	47
5.2	Concern Tagging Guidelines	48
A.1	Illustration of the use of Guideline 10.	57
A.2	Illustration of the use of Guideline 11.	58
A.3	Illustration of the use of Guideline 12.	58

A.4	Illustration of the use of Guideline 13.	59
A.5	Illustration of the use of Guideline 14.	59
A.6	Illustration of use of Guideline 15.	60
A.7	Illustration of the use of Guideline 16.	60

ABSTRACT

Much of the complexity of software arises from interactions between disparate concerns. Even in well-designed software, some concerns cannot always be encapsulated in a module. Research on separation of concerns seeks to address this problem, but we lack a fundamental understanding of how programmers conceptualize the notion of a concern and then identify how such concerns are represented in source code. In this work, we have conducted two exploratory case studies to better understand these issues. The case studies involved programmers identifying concerns and their associated code in existing, unfamiliar software: GNU's `sort` and a Java implementation of the game Minesweeper. Based on our experiences with these two case studies, we have developed a taxonomy of concern types and have identified a number of factors that impact programmer identification of concerns. Based on these insights, we have created two sets of guidelines: one to help programmers identify relevant concerns, and another to help programmers identify code relating to concerns.

AN INVESTIGATION OF PROGRAMMER-IDENTIFIED CONCERNS

Chapter 1

Introduction

A key problem that software engineers face in trying to develop, maintain, or even just understand a piece of software is that software can be very complex. The complexity that software developers and maintainers are confronted with can be derived in large part from the interaction of concerns in source code. Techniques for separation of concerns [3, 14] seek to cleanly separate concerns in source code in order to reduce complexity and increase comprehensibility [9, 13]. As Murphy and Lai note, many of the existing approaches for separation of concerns are still maturing, so there is no widely-accepted definition of what constitutes a concern [11].

Software engineering research currently lacks a fundamental understanding of the nature of concerns and therefore does not have a universally accepted definition for them. There is an intuitive understanding of concerns, but a concrete definition is lacking [21]. Of the definitions provided by researchers, most are either very broad and general or very narrow and specific. The closest to a standard definition comes from the IEEE [8], where “concerns are those interests which pertain to the system’s development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders,” but this description is still very indefinite because it leaves many things

open for interpretation. Many others have offered their definitions of “concern.” Robillard describes a concern as “any consideration ... about the implementation of a program” [16]. Similarly, Ossher and Tarr define a concern to be a part of a software system that is relevant to a specific concept or purpose. They also note that there can be many different kinds of concerns at the different stages of the software life cycle [13]. For example, views can be used in the requirements phase to address only the criterion (concern) of interest [12]. Sutton [21] adds his own general characterization of a concern as “any matter of interest in a software system.” While there is nothing inherently wrong with these definitions because they are so flexible, their generality leaves the meaning of “concern” unclear.

Other definitions of “concern” are more specific and narrow. Lai and Murphy [10] as well as Turner et al. [23] consider concerns to be features. A feature is a functional property of a system that would be known to the user. This definition is too exclusive as it ignores many possible non-functional concerns such as performance optimizations, error handling, and debugging code. Aspect-oriented programming (AOP) [9] proposes a definition of *aspects*. Aspects are a new programming language abstraction that seek to cleanly encapsulate program properties that cannot be modularized by procedural and object-oriented languages. These program properties are often called *cross-cutting concerns*. However, AOP’s aspect definition is too restrictive to be a definition for all concerns because it excludes anything that can be encapsulated by other existing techniques.

Clearly, there is no consensus on the meaning of “concern” since researchers’ definitions range from the vague “any consideration” to functional properties such as features to AOP’s new encapsulation units known as aspects. We believe that this lack of consensus is due at least in part to the lack of understanding of how programmers think about concerns and identify them in source code. While we acknowledge that the flexibility in the notion of what a concern is useful, a clearer

understanding of possible types of concerns would be a valuable guide for programmers and would help elucidate the terminology of researchers. From a practitioner's standpoint, a programmer faced with the task of identifying concerns in source code has only intuition and experience to guide him or her. Just as there is no definition of a concern, there is no agreement on how to decide what concept within an implementation is significant enough to be a concern. How does a programmer determine the presence of a concern in software? Once a programmer does decide on the existence of a concern, how does he or she identify the manifestation of that concern in the source code? How does a programmer find the fragments of code that implement that concern? The purpose of this work is to help answer these questions.

We completed an exploratory study to discover how programmers think about concerns, how they identify them, and how they link concerns to specific fragments of source code. Our investigation involved two case studies in which two different pairs of programmers identified concerns and the code associated with them. In both case studies, we compared the set of concerns the investigators found in order to better understand how the investigators identified concerns in the system. We also compared the similarity of the code each investigator associated with those concerns, relationships between concerns, and how dispersed the manifestation of a concern was in the code. We did this in order to better understand the causes of similarity (or lack thereof) in how programmers understand and approach concerns.

There are several contributions of this work. The first contribution is the presentation of data on the presence and relationships among concerns in real code. A second contribution is a classification of different types of concerns that we observed. The third contribution is insights gained regarding factors that contribute to consistent concern identification. The final contribution is a set of guidelines that can help programmers to more consistently identify concerns and their manifestation in

source code.

The rest of this thesis is structured as follows. In Chapter 2, we present the methodology for our case studies. Chapters 3 and 4 describe the two case studies we completed, and Chapter 5 generalizes those case studies. Chapter 6 provides a summary of related work. Finally, Chapter 7 offers a discussion of possible future directions and concludes.

Chapter 2

Case Study Methodology

In this chapter, we explain the methodology behind our two case studies. We focus on the goals these studies were intended to achieve, the approach used in them, and the metrics we used to analyze the concerns and related code.

2.1 Goals

There were several over-arching goals of our two case studies. The first was to gain insight into the ways programmers think about concerns in order to develop a better definition of “concern” that is precise but not too general or too specific. We chose to consider concerns in implementation by having programmers identify concerns in existing code. Another way to investigate concerns would be to study requirements documents. Our second goal was to develop a set of guidelines based on our new-found understanding of concerns. The purpose of these guidelines is to provide programmers with a suitable framework for thinking about concerns by telling them where and how to look for concerns in source code. In order to achieve these goals, there were several questions we had to take into consideration. What process do programmers use to find concerns in code?

Once a programmer identifies a concern, how do they decide what code is associated with it? Or is it the other way around and a programmer considers some code that causes them to identify a concern? What sort of factors cause different programmers to identify similar or dissimilar concerns and code associated with those concerns? Are there different types of concerns that programmers identify? The answers to these questions helped us in developing our definition of “concern” and our guidelines for concern identification in source code.

2.2 Approach

Our two case studies were GNU’s `sort.c` and a Java version of the Minesweeper game. These systems were selected for several reasons. First, we needed relatively small code bases (`sort` is approximately 2100 lines and Minesweeper is roughly 2800) because of the time-intensive, manual nature of the work involved in identifying concerns. It would not be infeasible, but it would be unreasonable, to have a single programmer be as rigorous in his or her concern identification in a larger system because of the amount of time it would take, especially if the programmer was not familiar with the source code.

One reason for choosing `sort` was that we had access to existing, independent concern data by Carver and Griswold [1]. This pre-existing data both eliminated the need for us to have another programmer identify concerns and their related code, and it provided us more independently derived data. `sort` is a very functional, batch, and feature-oriented program written in C. For our other case study, we wanted a different type of system implemented in a different language so that we could observe trends that cut across dissimilar software. Minesweeper, unlike `sort`, is an interactive, graphical, and object-oriented system written in Java. We did not have existing concern information,

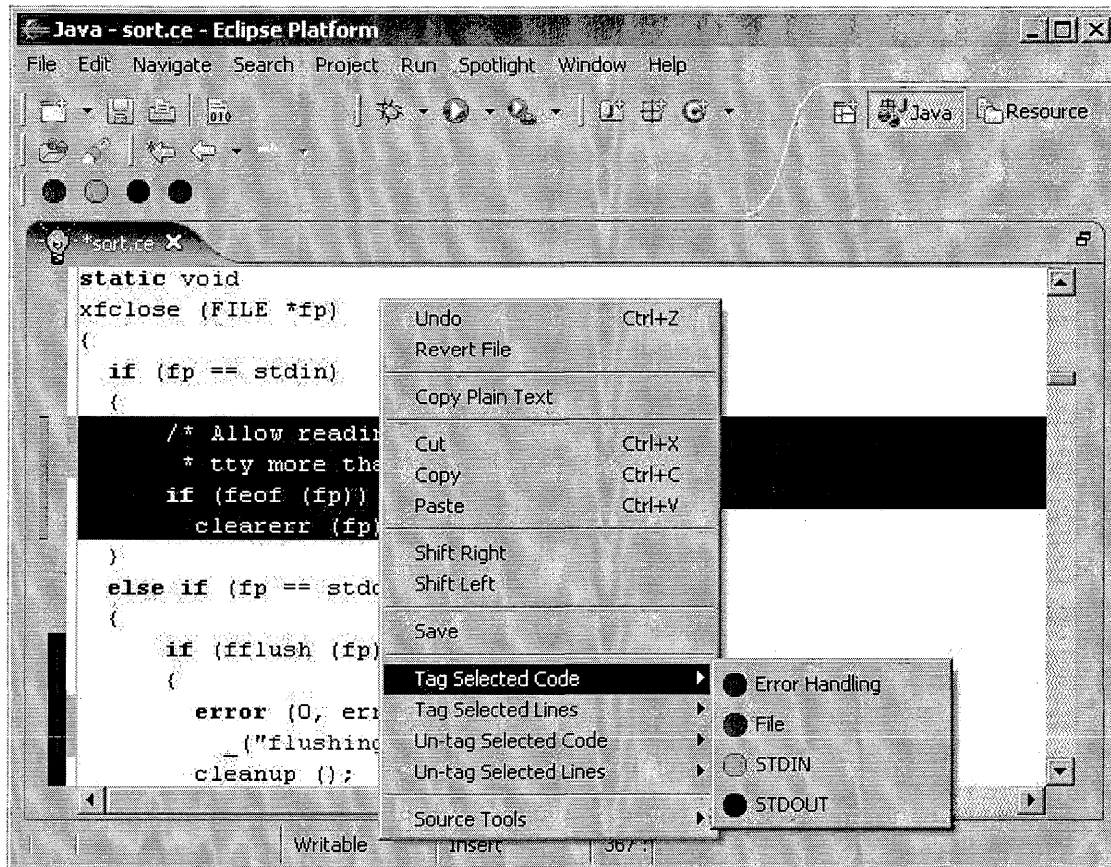


Figure 2.1: In the Spotlight editor, the programmer is using the context menu to annotate part of the code with a concern.

so two investigators had to identify concerns in this case study.

Both case studies that we completed involved two programmers independently investigating code they did not write. There was no communication between the investigators as they reviewed the code to find concerns. In the sort case study, since concern information was already available, the investigator did not review it until after she had completed her task. The investigators' task was to simply identify concerns and the code they believe is related to each concern. The programmers annotated the code with concern names using Spotlight, an Eclipse [4] plug-in that we developed. They were given no initial guidance on how to identify concerns or their related code.

Figure 2.1 shows a screen shot of Spotlight. Spotlight allows the programmer to associate fragments of source code with one or more concerns, where a fragment is any arbitrary continuous sequence of characters. We refer to this process of associating fragments of code with a concern as tagging or annotating the code with a concern. To display the taggings or annotations, Spotlight has a vertical ruler on the left-hand side of the editor screen. As shown in the figure, each concern that the programmer creates has its own column and color in the “concern ruler.” When a segment of code is annotated as belonging to a particular concern, a vertical bar appears in the corresponding column for the concern. The user can also manage the ruler annotations by rearranging the order of the concerns, or by associating multiple related concerns with a single color. As another way to view annotations, the user can also tell the tool to underline the particular characters in the code that are associated with a concern. Figure 2.1 also shows the context menu that is displayed when the user selects a segment of code and right-clicks on it. This menu allows the user to easily edit the annotations for a fragment of code. For the case studies, the investigators became familiar with how to use Spotlight before beginning, but they were given no formal training. The existing concern information for the `sort` case study was not done in Spotlight, so we faithfully annotated the program in Spotlight to make comparisons easier.

2.3 Analyses

After the investigators finished reviewing the code and finding concerns, we performed several analyses to gain insights into how the investigators thought about concerns. With these analyses, we seek to discover trends in the way people consider concerns in source code. We used what we learned from those common trends to drive the development of our concern definition and guidelines

presented in Chapter 5.

2.3.1 Concern Overlap

Since we were interested in how programmers associate a concern with specific fragments of code, one metric we created was concern overlap. Concern overlap measures how similar the code one programmer associated with a concern is to the code associated with the same or a similar concern found by another programmer. One of our colleagues implemented new features in the Spotlight tool for us to use in our analysis of concern overlap. These new features compute concern intersection and subtraction to aid us in comparing code tagged as part of one concern to code tagged with another concern.

We also investigated concern overlap for groups of concerns. A concern group is a selection of concerns that are related. For example, in a program with a graphical user interface, one programmer may identify an *Event Listeners* concern while another programmer has two concerns, one for *Keyboard Listener* and another for *Mouse Listener*. The concern intersection or subtraction features allow us to group the *Keyboard Events* and *Mouse Listener* concerns into a single concern group and compare the combined taggings of both concerns to the annotated code of the *Event Listeners* concern.

These new Spotlight features perform a character-by-character comparison of the code tagged with a concern or concern group to determine either the concern intersection (the number of characters tagged with a concern from each group) or the concern subtraction (the number of characters that one concern group contains that the other does not). These features enabled us to calculate the percent overlap of concern code tagged by the investigators for two concerns or concern groups.

To quantify this, first let us define two variables. Let c_1 be the set of characters tagged with the

first concern group, and let c_2 be the set of characters tagged with the second concern group. We calculate the percent of concern overlap metric as follows:

$$concern_overlap(c_1, c_2) = \frac{|c_1 \cap c_2|}{|c_1 \cup c_2|} * 100$$

Realizing that the concern overlap metric could be skewed by minor differences such as one investigator tagging newlines and other white space while the other did not, we also considered line overlap. We consider a concern to be present in a line if any portion of that line is tagged with that concern.

2.3.2 Concern Abstraction

Programmers think about concerns in different ways. Two programmers may think about a concern at different levels of detail or generality. One programmer may prefer to consider several related yet independent aspects of a system separately as different concerns while another may only wish to think about a single, simplified representation of all those aspects together as one concern.

To represent this fact, we developed the idea of *concern abstraction*. Concern abstraction is the mapping of a related group of concerns to a single concern. For example, in a simple graphical paint program, one programmer may identify a single concern for drawing shapes while another programmer may identify several concerns for drawing different shapes such as circles, squares, and triangles. We say that the drawing shapes concern of the first programmer subsumes the drawing circles, squares, and triangles concerns of the second programmer because the drawing shapes concern conceptually includes drawing specific types of shapes. When initially comparing the set of concerns that these two programmers found, it would appear that they had none in common. However, using concern abstraction, we notice that the one concern of the first programmer covers

all three concerns of the second programmer, so the set of concerns found by these programmers is not as dissimilar as it originally seemed. Concern abstraction helps to bridge the gap between the differences in the way individual programmers think about concerns.

2.3.3 Scatter and Spread

Scattering of concerns is one cause of complexity in software [9]. Scattering occurs when a concern is not well-encapsulated, so it is dispersed throughout the modules of a system. Scatter is an important idea in the area of separation of concerns, so we consider it in our analysis of concerns in general. We use the terms scatter and spread interchangeably.

As a way of measuring the scatter of a concern, we borrow Lai and Murphy’s spread metric [10], replacing “feature f” in their definition with “concern c:”

$$spread(c) = \frac{\# \text{ of classes containing concern } c}{\text{total } \# \text{ of files}}$$

This metric measures the number of files in which a concern appears. This spread or scatter metric is useful because it helps us to determine whether programmer identification of concern code is more likely to agree when the concern code is localized to a small number of files. However, this spread metric is too insensitive to file size and does not take into account the modules of a system. Therefore, we also consider spread in other ways.

To account for varying file sizes, we look at the number of lines separating an instance of a concern within a file. This metric tells us if intra-file locality is a factor in concern identification. The problem with this metric is that scatter is defined in terms of program modules, not lines of code, so we have an additional metric to measure spread within a single file or class. This metric looks at the number of functions (or methods in object-oriented code) in which a concern is present

in a file or class compared to the total number of functions or methods in that file/class. We call this metric *function scatter* or *method scatter*, and it gives us a finer grain idea of concern scatter than the file or line spread metrics.

Chapter 3

Case Study #1: GNU `sort`

The GNU `textutils-1.22` implementation of `sort` was the subject of our first case study to investigate how programmers think about concerns in code. We wanted to know the types of things programmers consider to be concerns and where those concerns are manifested in code. `sort` is an approximately 2100-line C program that sorts lines of input either from files or standard input. The resulting lines are written to standard output by default or to a file if specified. Among other features, `sort` will automatically use a temporary file if the output file is also the input file. There are 18 command line flags the user can specify when executing the program. The `-c` and `-m` flags change `sort`'s mode of operation to check if the given files are already sorted or to merge the given files, respectively. The user can specify one or more key fields to control how input is sorted. The user can also provide a number of global sort options, such as sort in reverse order or ignore non-printing characters. For this case study, we compare concerns we identified in `sort` to concerns found by Carver and Griswold [1] in previous work.

3.1 Concerns Identified

We identified 50 concerns in `sort`. We had used C previously but not extensively and did not have much knowledge of the standard libraries used. Many of our concerns were related to specific user-level features such as specifying the output file, reversing the sort order, and displaying help information. Other concerns were related to internal aspects of the program a user would not be aware of, such as the use of assertions, buffers, temporary files, POSIX compliance, or signal handling.

To better understand the subjective nature of programmer identification of concern code, we compared our concerns to those of Carver and Griswold, who used the same implementation of `sort` in their work. One difference between the two sets of concerns was in the number of concerns found—they had 83 concerns compared to our 50. Table 3.1 shows all the concerns identified in the `sort` case study categorized by those concerns found by both us and Carver and Griswold, those found only by us, and those only found by Carver and Griswold.

There were 23 commonly identified concerns between the two parties—mostly user-level features. The majority of the additional concerns Carver and Griswold identified relate to more fine-grained details of concerns we had or to system-specific issues that we did not address, such as access to the system environment space and releasing the thread of execution to the operating system. We do not know how well Carver and Griswold knew the implementation language, but we assume they were more familiar with it than we were based on the presence of these more specific concerns.

Both	Us	Carver & Griswold	
<i>Alternate EOL</i>	<i>Buffers</i>	<i>1stKeyOptz</i>	<i>RecOrder</i>
<i>Assertions</i>	<i>Close File</i>	<i>Application</i>	<i>RevGlobal</i>
<i>Character Set</i>	<i>Command line</i>	<i>BadField</i>	<i>RevKey</i>
<i>Check</i>	<i>Error Handling</i>	<i>Blanks</i>	<i>SimpleCompare</i>
<i>Dictionary Order</i>	<i>Exit Status</i>	<i>BufStream</i>	<i>SingleSortOptz</i>
<i>General Numeric</i>	<i>File Input</i>	<i>ByteOrder</i>	<i>SortFiles</i>
<i>Large Files</i>	<i>File Output</i>	<i>CmndLine</i>	<i>SortLines</i>
<i>Locale</i>	<i>Files to Sort</i>	<i>ErrorExit</i>	<i>SortMode</i>
<i>LocalOptz</i>	<i>Free Memory</i>	<i>ErrorSrcv</i>	<i>SrcSpec</i>
<i>Month Order</i>	<i>Help</i>	<i>FieldCompare</i>	<i>StdError</i>
<i>Numeric Order</i>	<i>Ignore Leading Blanks</i>	<i>FieldMgmt</i>	<i>SysCfg</i>
<i>Output File</i>	<i>Ignore Non-printing</i>	<i>FieldOpts</i>	<i>SysCtrl</i>
<i>POSIX</i>	<i>Key</i>	<i>Fields</i>	<i>SysEnv</i>
<i>Program Name</i>	<i>Key Field</i>	<i>FieldSeq</i>	<i>SysIO</i>
<i>Race Condition</i>	<i>Lines</i>	<i>GlobalKey</i>	<i>SysMem</i>
<i>Signals</i>	<i>Memory Allocation</i>	<i>GlobalOpts</i>	<i>SysSrcv</i>
<i>Solaris</i>	<i>Merge</i>	<i>GnuOpts</i>	<i>TempCleanup</i>
<i>Stable</i>	<i>Open File</i>	<i>IgnoreCompare</i>	<i>TempMngr</i>
<i>Tab Separator</i>	<i>Reverse</i>	<i>IgnoreText</i>	<i>TempSpace</i>
<i>Temp Directory</i>	<i>!SA_INTERRUPT</i>	<i>Input</i>	<i>TextOrder</i>
<i>Unique</i>	<i>SA_INTERRUPT</i>	<i>LocaleSrcv</i>	<i>TransText</i>
<i>Usage Message</i>	<i>Sort</i>	<i>MergeFiles</i>	<i>TrimFields</i>
<i>Version</i>	<i>STDIN</i>	<i>MergeMode</i>	<i>TrimKey</i>
	<i>STDOUT</i>	<i>Modes</i>	<i>UpperCase</i>
	<i>Temp Files</i>	<i>MultiOrder</i>	<i>UpperMon</i>
	<i>To Uppercase</i>	<i>OldFields</i>	<i>UpperText</i>
	<i>Trailing Blanks</i>	<i>OutputBytes</i>	<i>WhiteSep</i>
		<i>OutputResult</i>	<i>WorkArea</i>
		<i>PrintOnly</i>	<i>WrapIO</i>
		<i>ProgID</i>	<i>WrapMem</i>

Table 3.1: Concerns identified by us and Carver and Griswold in GNU sort.

3.2 Concern Overlap

Concern overlap is a measure of how similar the code that two programmers believe is associated with a concern. In the `sort` case study, there were 18 concerns with 80% character overlap in what we and Carver and Griswold tagged and 7 concerns with 100% overlap, as can be seen in the character overlap column of Table 3.2. Twelve of these concerns with better than 80% character overlap are user-level features. The average concern overlap for characters was 82.52%. Interestingly, when we looked at line overlap, the number of concerns with 100% increased to 9, but the average line overlap dropped to 79.57%. Table 3.2 also shows the total number of characters and lines tagged with each of the concerns. The total number includes any character or line annotated with a particular concern by either programmer. Nine of the 18 concerns with better than 80% concern overlap have over 1000 total characters tagged, so half of the concerns with high overlap had a significant number of characters annotated.

Concern overlap is obviously effected by cases where one programmer annotated a fragment of code while the other did not. However, sometimes programmers do associate code in the same general area of the source code, but that associated code is not exactly the same. We observed cases where the two programmers had identified the same concern in the same location in the code, but had different interpretations on how to actually tag the manifestation of the concern. For an illustrative example of this situation, consider the *Unique* concern which removes duplicates from the input to be sorted so that each item is unique. The code in Figures 3.1(a) and 3.1(b) comes from `sort` and shows two different taggings of the same fragment. Underlining means that the code was associated with the *Unique* concern by that particular programmer. Part of the `if` statement's condition tests to see if `unique`, a global flag, is true. Because of this, we included the whole of the

Concern	Character Overlap	# Chars. Tagged	Line Overlap	# Lines Tagged
<i>Alternate EOL</i>	93.19%	470	100.00%	14
<i>Assertions</i>	42.26%	168	66.67%	6
<i>Character Set</i>	40.71%	737	48.94%	47
<i>Check</i>	90.08%	3985	93.60%	125
<i>Dictionary Order</i>	100.00%	378	100.00%	8
<i>General Numeric</i>	99.24%	1177	100.00%	45
<i>Large Files</i>	0.00%	801	0.00%	27
<i>Locale</i>	100.00%	109	100.00%	4
<i>LocalOptz</i>	100.00%	646	100.00%	22
<i>Month Order</i>	100.00%	1378	100.00%	66
<i>Numeric Order</i>	99.02%	4175	98.59%	213
<i>Output File</i>	90.64%	2222	83.33%	90
<i>POSIX</i>	63.12%	5719	67.46%	169
<i>Program Name</i>	40.00%	310	66.67%	9
<i>Race Condition</i>	96.61%	1592	98.25%	57
<i>Signals</i>	100.00%	1400	100.00%	49
<i>Solaris</i>	100.00%	123	100.00%	4
<i>Stable</i>	100.00%	364	100.00%	10
<i>Tab Separator</i>	98.29%	819	94.29%	35
<i>Temp Directory</i>	87.65%	834	88.46%	26
<i>Unique</i>	86.22%	2504	79.07%	86
<i>Usage Message</i>	84.39%	2569	78.18%	55
<i>Version</i>	86.67%	150	66.67%	3
Average	82.52%	1418.70	79.57%	50.87

Table 3.2: Concern overlap for characters and lines for concerns identified by us and Carver and Griswold in the sort case study.

<pre> if ((unique && cmp >= 0) (cmp > 0)) { sorted = 0; break; } </pre>	<pre> if ((unique && cmp >= 0) (cmp > 0)) { sorted = 0; break; } </pre>
(a) Our tagging.	(b) Carver and Griswold's tagging.

Figure 3.1: Two different taggings of an instance of the *Unique* concern.

if block in her *Unique* concern because it can will be executed if the unique flag is true. However, Carver and Griswold excluded the body of the if statement from their *Unique* concern presumably because the unique flag does not have to be true for the body of the if statement to be executed. Differences in interpretations of what code should be associated with a concern based on the context of that code in the program is a factor that reduced concern overlap.

3.3 Concern Abstraction

There was a significant difference in the number of concerns that we and Carver and Griswold identified in *sort*. This difference can be explained by the fact that the two parties tended to think about concerns at different levels of detail and abstraction. Not all of the concerns that only one programmer or the other found were unrelated. For example, Carver and Griswold created *meta-concerns* to group related concerns, and these meta-concerns had no associated code. For instance, Carver and Griswold had a *Modes* meta-concern to encompass *sort*'s three modes of operation: sorting files, merging files, and checking if files are already sorted. We had individual *Sort*, *Merge*, and *Check* concerns, but did not see the need to create a higher level concern such as *Modes* because there would not have been associated code with it.

In contrast, it was more often the case that Carver and Griswold used multiple concerns where

we used a single concern. They identified a *Month Order* concern that deals with sorting dates by month and an *UpperMon* concern that consists of a single line of code that translates month names to upper case. We had only a *Month Order* concern which included the code for converting month names to upper case, but we did not think such a small feature warranted a concern because its total associated code was only one line.

Tables 3.3 and 3.4 display the concern abstractions that we have discerned based on the sets of concerns found by us and Carver and Griswold in `sort` and the code associated with those concerns. There are 20 concern groups that relate one or more of our concerns to one or more of Carver and Griswold's concerns. Of the 27 concerns that we had and Carver and Griswold did not, we were able to map all of them using concern abstraction. This means that in some way, Carver and Griswold incorporated all of our concerns in their concerns. On the other hand, of the 60 concerns that only Carver and Griswold had, 53 map to one or more of our concerns using concern abstraction.

It is possible for a concern to be a part of more than one concern group. For instance, Carver and Griswold's *UpperMon* concern was subsumed by both of our *Month Order* and *To Uppercase* concerns. *UpperMon* translates the names of months to uppercase. `sort` does this translation to improve month name recognition when putting dates in order (which is the purpose of the *Month Order* concern). Our *To Uppercase* concern handles all cases of translating characters to upper case, of which converting month names is one example in the program. It is significant that *UpperMon* is subsumed by two different concerns because it reveals the relationship between the *Month Order* and *To Uppercase* concerns.

Seven of Carver and Griswold's concerns did not relate to any of our concerns. These concerns are all at a level of abstraction above or outside specific functionalities within the program. *Application* is a meta-concern for the main program. *Modes* is another meta-concern that is described

Our Concerns	Carver & Griswold Concerns	Character Overlap	# Characters Tagged
<i>Buffers File Input Lines</i>	<i>BufStream Input SortLines SrcSpec</i>	69.74%	10559
<i>Close File Open File</i>	<i>WrapIO</i>	44.82%	1756
<i>Command Line</i>	<i>CmndLine GnuOpts GlobalOpts</i>	99.37%	7454
<i>Ignore Leading Blanks Trailing Blanks</i>	<i>Blanks TrimFields TrimKey</i>	64.34%	2947
<i>Ignore Non-printing</i>	<i>IgnoreCompare IgnoreText PrintOnly TextOrder</i>	82.12%	2729
<i>Error Handling</i>	<i>BadField ErrorExit ErrorSrcv StdError</i>	85.42%	7477
<i>File Output</i>	<i>OutputBytes OutputRslt</i>	85.15%	1475
<i>Files to Sort Sort</i>	<i>ByteOrder RecOrder SortFiles SortLines Sort Mode WorkArea</i>	67.42%	5157
<i>Files to Sort STDIN</i>	<i>SrcSpec</i>	30.03%	1572
<i>Merge</i>	<i>MergeFiles MergeMode</i>	97.87%	6335
<i>Month Order</i>	<i>Month Order UpperMon</i>	100.00%	1378

Table 3.3: Character concern overlap for groups of concerns in the sort case study.

Our Concerns	Carver & Griswold Concerns	Character Overlap	# Characters Tagged
<i>Key</i>	<i>1stKeyOptz FieldCompare FieldMgmt FieldOpts Fields FieldSeq GlobalKey MultiOrder RecOrder TextOrder</i>	63.45%	15400
<i>Key Field</i>	<i>Fields OldFields WhiteSep</i>	54.82%	11952
<i>Memory Allocation</i>	<i>SysMem WrapMem</i>	41.58%	1847
<i>Output File STDOUT</i>	<i>Output File SysIO</i>	85.02%	2444
<i>Reverse</i>	<i>RevGlobal RevKey</i>	97.48%	555
<i>Signals SA_INTERRUPT !SA_INTERRUPT</i>	<i>SigHand</i>	100.00%	1400
<i>Temp Files</i>	<i>TempCleanup TempMgr TempSpace</i>	61.00%	3049
<i>To Uppercase</i>	<i>IgnoreCompare TransText UpperCase UpperMon UpperText</i>	97.06%	2719
<i>Usage Message Help</i>	<i>Usage Message</i>	83.74%	2589

Table 3.4: Character concern overlap for groups of concerns in the sort case study.

above. *ProgID* consists of the comments at the top of the file with the author and copyright information. We did not have these three concerns because we did not consider possible concerns at higher levels than individual pieces of the program and its functionality. The four final concerns that did not map to any of our concerns deal with low-level system issues: *SysCfg* handles system configuration settings, *SysCtrl* manages receiving and releasing the thread of execution to the operating system, *SysEnv* provides access to the system environment space, and *SysSvc* is a meta-concern for general purpose virtual machine services. We did not have these four concerns because of our lack of familiarity with C and its standard libraries.

3.4 Spread

Since `sort` is implemented in a single file, we could not measure the spread of concerns across multiple files. However, we were able to measure spread in terms of the number of lines separating instances of a concern. Table 3.5 gives the raw data for the spread metric plus the number of instances of each concern. The data confirms that programmers are able to find the same concerns at the same locations in code since for the most part, the concerns with high overlap have low average spread. Figure 3.2 shows the percent character overlap for commonly identified concerns versus the average spread between instances of a concern for each programmer. Fifteen of the 23 commonly identified concerns had both a concern overlap greater than 80% and an average spread less than 600 lines. We make a distinction at 600 lines because a program of 600 lines should be reasonably simple to understand. Programmers seem to be able to more easily find the code associated with concerns that are less spread out in a program.

Concern	Us		Carver & Griswold	
	Avg. Spread	Instances	Avg. Spread	Instances
<i>Alternate EOL</i>	193.1	11	214.56	10
<i>Assertions</i>	614	3	N/A	1
<i>Character Set</i>	34.76	30	81.93	15
<i>Check</i>	297.5	7	297	7
<i>Dictionary Order</i>	513.67	4	514	4
<i>General Numeric</i>	233	9	219.67	10
<i>Large Files</i>	N/A	1	5.57	5
<i>Locale</i>	1607	2	1697	2
<i>LocalOptz</i>	1270	2	1270	2
<i>Month Order</i>	187.5	11	187.5	11
<i>Numeric Order</i>	216.5	9	216.63	9
<i>Output File</i>	459.25	5	204.78	10
<i>POSIX</i>	306.5	7	809.5	3
<i>Program Name</i>	400	5	399.25	5
<i>Race Condition</i>	N/A	1	342	2
<i>Signals</i>	565.67	4	565.67	4
<i>Solaris</i>	N/A	1	N/A	1
<i>Stable</i>	595	4	595	4
<i>Tab Separator</i>	442.5	5	442	5
<i>Temp Directory</i>	275	8	301	7
<i>Unique</i>	148.55	10	207	9
<i>Usage Message</i>	586.33	4	N/A	1
<i>Version</i>	856.5	3	1495	2

Table 3.5: Spread of commonly identified concerns in sort.

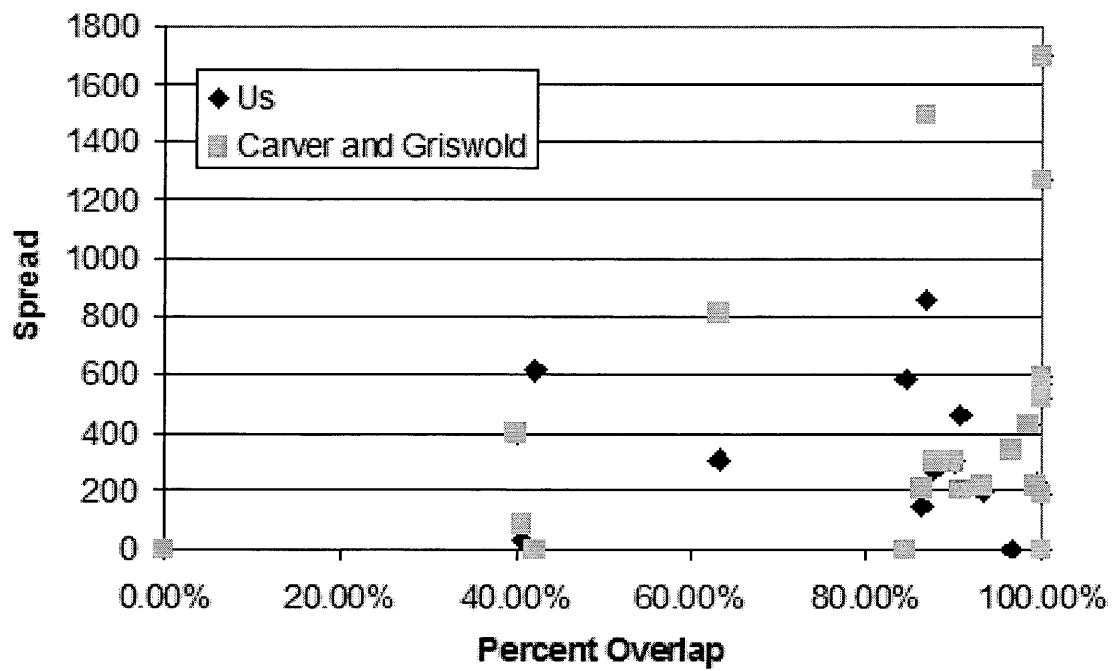


Figure 3.2: Concern overlap vs. average spread for commonly identified concerns in sort.

Chapter 4

Case Study #2: Minesweeper

For our second case study, we considered a Java implementation of the game Minesweeper that is 2776 lines contained in six classes. One class controls the logic of the game, and the remaining five classes deal with components of the graphical user interface such as the frame the game is played in, a pop-up dialog window, and a specialized panel that displays the time since the game started. In this game, the user is presented with a grid of cells, any one of which may contain a “mine.” When the user selects a cell, either no mine is present, a mine is present, or there is a digit indicating the number of adjacent cells that contain mines. The user can also “flag” cells which he or she believes contain a mine. The game ends when the user correctly identifies all of the cells not containing mines or clicks on a cell containing a mine. Figure 4.1 shows a screen shot of the game. For this case study, two of investigators (Investigators M and T) independently identified concerns in the Minesweeper source code. Investigator M had over three years of experience using the implementation language and had previously written graphical user interfaces in Java. Investigator T had over two years of experience with the implementation language but had never programmed graphical user interfaces in Java.

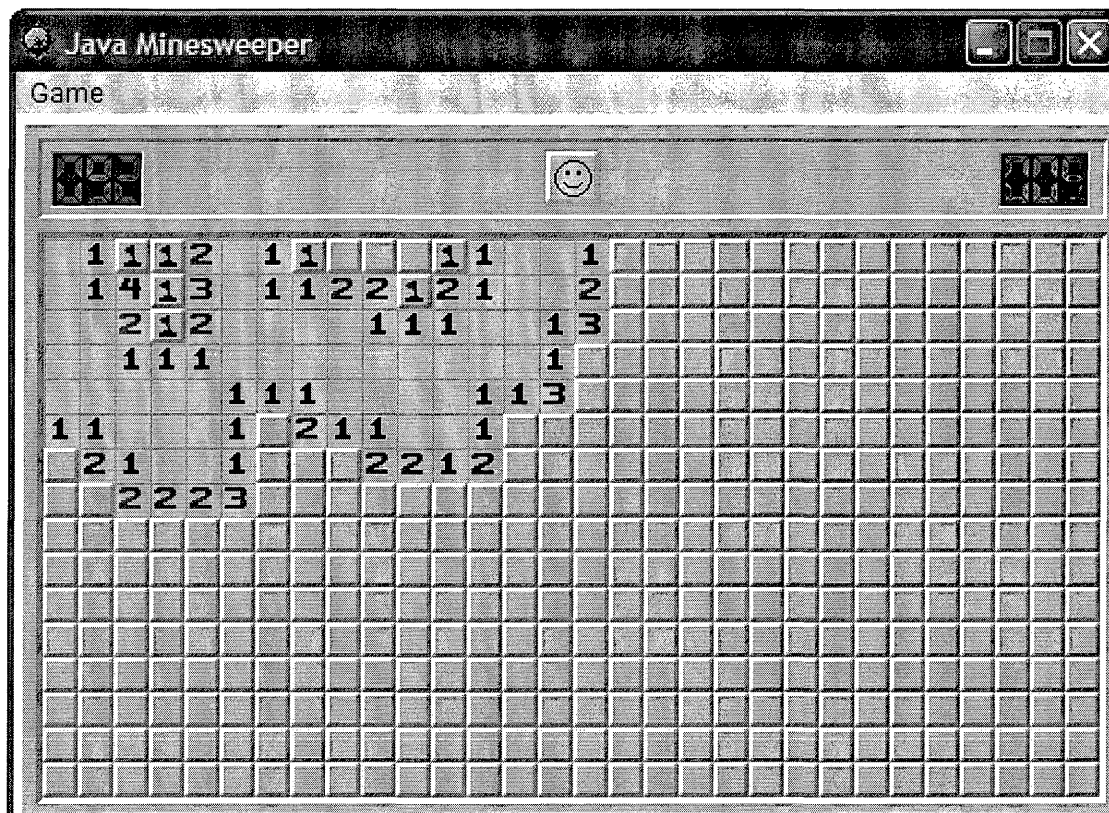


Figure 4.1: A screen shot of the Minesweeper game.

4.1 Concerns Identified

For the Minesweeper case study, we were able to do more analysis of the concerns identified than in the sort case study because we had access to both investigators instead of only one investigator and concern information as in the previous study. After locating concerns in code, the Minesweeper investigators were able to discuss their reasons for identifying certain concerns and for tagging fragments of code in certain ways. However, they did not communicate about the concerns they were finding while in the process of identification.

Investigator M found 30 concerns, and Investigator T found 26 concerns. We compared both sets

Concerns identified by both investigators	Concerns identified only Investigator M	Concerns identified only Investigator T
<i>Cell State</i>	<i>Accessors</i>	<i>Cell</i>
<i>Debug</i>	<i>Cell is mine</i>	<i>Color</i>
<i>Error Handling</i>	<i>Connected Mines</i>	<i>Constants</i>
<i>Flag Cell</i>	<i>Custom dialog</i>	<i>Import</i>
<i>Game Difficulty</i>	<i>Custom dialog visible</i>	<i>Keyboard</i>
<i>Game State</i>	<i>Custom game</i>	<i>Mines</i>
<i>Graphics</i>	<i>Custom input</i>	<i>Smiley</i>
<i>GUI</i>	<i>Easy game</i>	<i>Mouse</i>
<i>Images</i>	<i>Event Listener</i>	<i>Stdout</i>
<i>LED</i>	<i>Expert game</i>	<i>Thread</i>
<i>Menu</i>	<i>Exploded</i>	<i>User Input</i>
<i>Minefield</i>	<i>Game Window</i>	<i>Window</i>
<i>Timer</i>	<i>Intermediate game</i>	<i>XY Location</i>
	<i>Mark Cell</i>	
	<i>Mines cleared</i>	
	<i>Neighbor Mines</i>	
	<i>Window size</i>	

Table 4.1: Concerns identified by Investigators M and T in the Minesweeper program.

of concerns and found 13 out of the total 43 concerns were identified by both investigators. Table 4.1 summarizes the concerns found in this case study classifying them as identified by both investigators and those concerns only found by a single investigator. The concerns can be categorized into several different types. Fifteen of the concerns deal with the graphical user interface, 6 more focus on events and event listeners, 5 pertain to the level of difficulty of the game, 11 relate to cells, mines, or the grid of cells and mines, and 2 concerns are about debugging. The remaining 4 concerns are all relatively independent of the other concerns: accessor methods, error handling, constants, and import statements.

4.2 Identification Process

Each investigator's task was to identify concerns and their associated code in the six Minesweeper files: `Game.java`, `MineSweeperWindow.java` (MSW), `LED.java`, `CustomFieldDialog.java` (CFD), `LEDPanel.java`, and `Cell.java`. Investigator M approached her task by beginning in `Game.java` since this file deals with the logic of the Minesweeper game. Investigator M's strategy was to identify a single concern in `Game.java` and then look for that concern in the other five files. Investigator T began tagging in `Game.java` as well, primarily because this is the longest file in the Minesweeper suite, and thus she expected it to yield the most concerns. Investigator T identified concerns in `Game.java` and then proceeded to look for those concerns as well as new ones in each of the other files in succession, tagging an entire file before moving on to the next. From these two experiences, there appears to be a common starting point among programmers for concern identification but different methods for examining the code. We cannot yet say whether concentrating on tagging all the concerns in individual files or tagging all instances of a single concern across every file is a more efficient approach for finding concerns. It may just be a matter of programmer preference.

Once an initial tagging of the code was completed, both investigators felt the need to go back through the code to ensure they had found all the fragments of code that they thought belonged to a particular concern. To ensure the completeness of her annotations, Investigator T searched for keywords to find code related to a concern that she had missed during the first pass of taggings. In her initial review of the code, she had learned what variables, constants, and methods were associated with a particular concern. In her second review of the code, she simply searched for those keywords to see if she had overlooked associating instances of a concern where these keywords

were used. Investigator M also reviewed her taggings to see if she had missed any concern code but did not use the search functionality. Instead, she simply scrolled through the code. Scrolling may take more time and be less efficient because there is the possibility of overlooking important code. Regardless of the method used, this final step in the process of locating concerns in code is important because the identification of a concern may be easier in a later portion of the code but the programmer may not have recognized fragments that pertain to the newly found concern in previously reviewed code.

4.3 Concern Overlap

The common concerns found by each investigator tell us what concepts within a software system they find important. However, having identified a similar concern does not mean the investigators associated exactly similar fragments of code with those concerns. We now examine concern overlap in the Minesweeper case study to analyze to what extent the code programmers link with a concern is alike.

The average concern overlap between concerns identified by the two investigators in the Minesweeper case study was 52.97%, which is lower than the average in the `sort` case study. Table 4.2 presents the character and line overlap for the thirteen commonly identified concerns. The Minesweeper case study did not yield any cases of 100% concern overlap, as the `sort` study did. Also, the average number of characters associated with each concern was higher in the Minesweeper case study. Unlike in the `sort` study where a high number of tagged characters did not necessarily mean a low concern overlap, the opposite seems to be true in general for the case of Minesweeper. We believe there was more overlap in the `sort` case study because `sort` is a more feature-oriented

Concern	Character Overlap	# Characters Tagged	Line Overlap	# Lines Tagged
<i>Cell State</i>	79.87%	4769	85.92%	206
<i>Debug</i>	40.12%	2074	61.76%	68
<i>Error Handling</i>	81.25%	1307	70.91%	55
<i>Flag Cell</i>	57.38%	1016	87.50%	64
<i>Game Difficulty</i>	30.00%	10934	25.64%	472
<i>Game State</i>	34.86%	1486	60.00%	65
<i>Graphics</i>	35.98%	12657	26.29%	464
<i>GUI</i>	15.00%	23678	13.58%	1016
<i>Images</i>	52.96%	9659	51.64%	275
<i>LED</i>	45.80%	11542	51.64%	548
<i>Menu</i>	88.87%	2687	92.05%	88
<i>Minefield</i>	47.65%	11023	45.70%	442
<i>Timer</i>	78.87%	970	83.78%	37
Average	52.97%	7215.54	58.19%	292.31

Table 4.2: Concern overlap for characters and lines for concerns identified by both investigators in the Minesweeper case study.

program. Because it has so many command line options, it is easier to identify code fragments that implement each individual feature.

Since the concern overlap metric can be skewed by minor differences in whitespace, we also looked at line overlap. Recall that we consider a concern to be present in a line if any portion of that line is tagged with that concern. The percent overlap between lines was generally an improvement over the percent overlap between characters in the Minesweeper case study, as shown in the right-most two columns of Table 4.2. The average overlap for lines was 58.19%. We attribute the cases where the line overlap for a concern was lower than the character overlap to one investigator tagging blank lines or lines of comments that the other did not. For the rest of this work, we mainly consider character overlap.

4.4 Concern Abstraction

While the two investigators only had 13 out of 43 total concerns in common, concern abstraction improves this ratio by combining related concerns. Table 4.3 shows the Minesweeper concern groups along with their concern overlap. Only one group had a concern overlap better than 80%, giving us further evidence that it is easier to identify concern code in feature-rich programs like `sort`. The groups were created based on concern abstraction, which is discussed in the next section. In one example, Investigator T had a *Mines* concern, while Investigator M had five concerns relating to mines: *Neighbor Mines*, *Connected mines*, *Cell is mine*, *Exploded*, and *Mines cleared*. We observed that when combined, these five concerns of Investigator M were equivalent to Investigator T's *Mines* concern and could conceptually be abstracted into a single concern. We followed a similar procedure for all of the concerns and created the concern groups and a concern abstraction hierarchy, shown in Figure 4.2.

Each object in Figure 4.2 represents a concern. The shape of the object indicates whether Investigator M, Investigator T, or both investigators identified the concern. A rectangle means Investigator M identified the concern, an ellipse means Investigator T found the concern, and a diamond means both investigators identified the concern. The number scale at the left of figure is the abstraction level of the concern. We identified nine different levels of abstraction, ranging from 1 to 9. Level 1 concerns are the lowest level of abstraction. These concerns are very specific and easy to identify in the source code. For example, all print statements are tagged with the *Stdout* concern. At the opposite end of the hierarchy, the *GUI* concern is placed at the highest level of abstraction with a ranking of 9. A higher ranking means that the concern is broader and more vague. These rankings were assigned subjectively.

Concern Group	Investigator M Concerns	Investigator T Concerns	Character Overlap	# Characters Tagged
<i>Cell</i>	<i>Cell State Flag Cell Mark Cell</i>	<i>Cell Cell State Flag Cell</i>	28.35%	2102
<i>Debug</i>	<i>Debug</i>	<i>Debug Stdout</i>	61.61%	1761
<i>Event Listener</i>	<i>Event Listener</i>	<i>Keyboard Mouse User Input Window XY Location</i>	72.27%	9775
<i>Game Difficulty</i>	<i>Game Difficulty Custom game Easy game Intermediate game Expert game</i>	<i>Game Difficulty</i>	96.54%	3321
<i>Game State</i>	<i>Game State</i>	<i>Game State Thread</i>	37.95%	1486
<i>Graphics</i>	<i>Graphics Images</i>	<i>Color Graphics Images Smiley</i>	42.58%	10797
<i>GUI</i>	<i>Custom dialog Custom dialog visible Custom input Game State Game Window Graphics GUI Images LED Menu Timer Window size</i>	<i>Color Game State Graphics GUI Images LED Menu Smiley Thread Timer</i>	83.37%	38498
<i>Mines</i>	<i>Cell is mine Connected mines Exploded Mines cleared Neighbor mines</i>	<i>Mines</i>	48.24%	3806
<i>Minefield</i>	<i>Cell is mine Cell State Connected mines Flag Cell Exploded Mark Cell Mines cleared Minefield Neighbor mines</i>	<i>Cell Cell State Flag Cell Minefield Mines</i>	62.15%	21549

Table 4.3: Character concern overlap for groups of concerns in the Minesweeper case study.

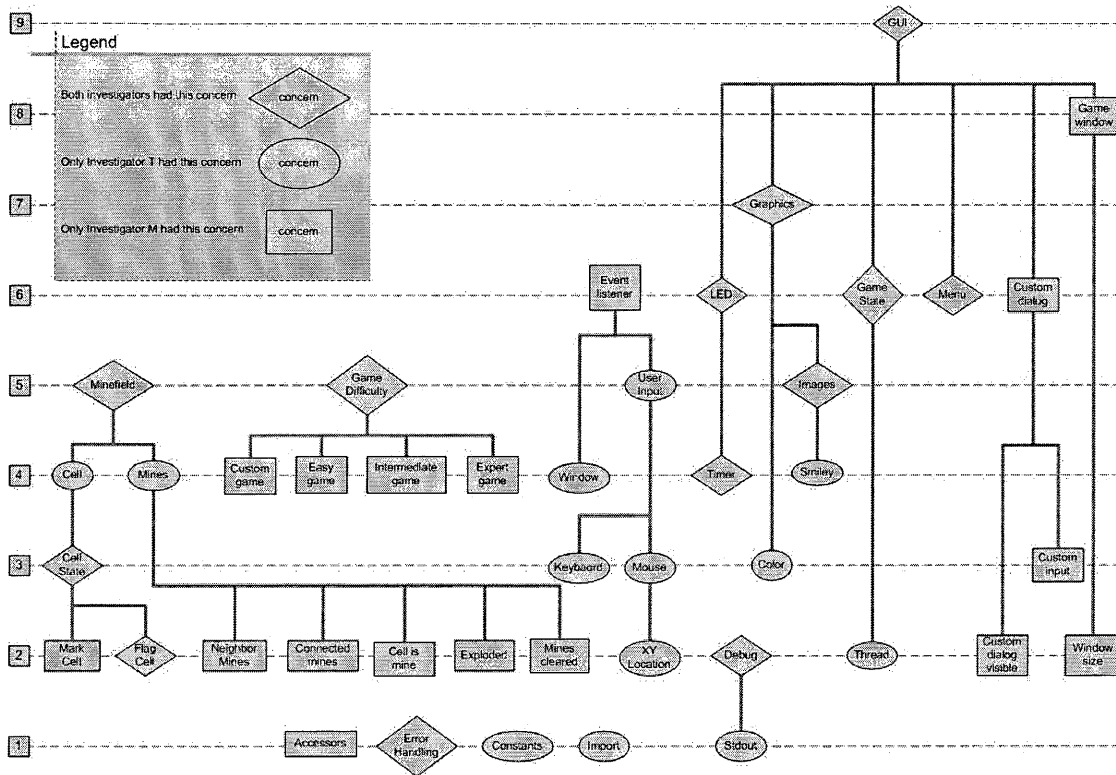


Figure 4.2: Hierarchy of concerns in the Minesweeper program.

If a concern has a line connecting it to concerns at a lower level of abstraction, we say that the concern subsumes those lower level concerns. This means that conceptually combining the lower level concerns should result in a concern equivalent to the higher level one. For example, the *Game Difficulty* concern found by both investigators subsumes the *Custom game*, *Easy game*, *Intermediate game*, and *Expert game* concerns, found by Investigator M. These concerns are the various levels of game difficulty, so their union should be equivalent to the *Game Difficulty* concern.

As can be seen from the hierarchy, programmers think about concerns on different levels. Investigator M tended to be more detailed in her concern identification and think at lower levels of abstraction than Investigator T. Interestingly, there were some cases when an investigator would

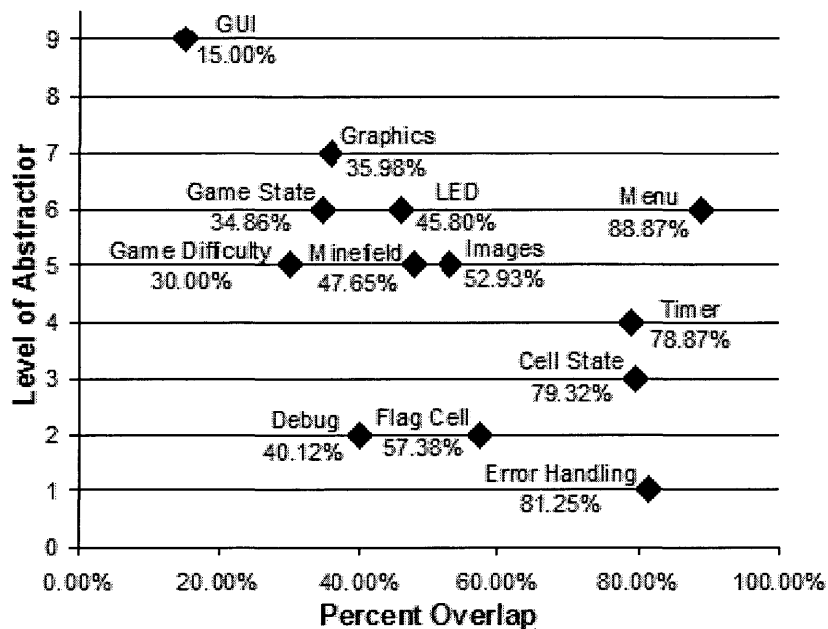


Figure 4.3: Concern overlap vs. abstraction level for concerns identified by both investigators in the Minesweeper study.

think on more than one level about related concerns. Again, take for example the *Game Difficulty* concern. Investigator M identified it as well as the four lower level concerns for the individual levels of play in the game. The four levels of game difficulty are all significant on their own, but Investigator M also recognized that they are relevant to the *Game Difficulty* concern and annotated instances of them as such. By also tagging the fragments of code related to the individual difficulties with the *Game Difficulty* concern serves as a form of documentation of the relationship between those two concerns.

4.5 Abstraction Level and Concern Overlap

There appears to be a correlation between the level of abstraction of a concern in the hierarchy and the percent overlap between two programmers' taggings of that concern, as shown in Figure 4.3. In

general, the concerns with a higher ranking have lower percent concern overlap, and concerns with a low abstraction level have a high concern overlap. The best example is the *GUI* concern, which is at the highest level of abstraction (9) and has the lowest percent overlap of any concern (15.00%). We hypothesize it is more difficult to determine the code associated with these broad, high level concerns. Similarly, the concerns at lower levels of abstraction tend to have higher percent overlap because the lower level concerns represent simpler, less abstract concepts. As examples of lower level concerns with high overlap, *Error Handling* is at level 1 and has 81.25% overlap, and *Cell State* is at level 3 and has 79.32% overlap.

4.6 Spread and Scatter

Since the Minesweeper case study consists of multiple files, each containing a single class, we were able to measure the spread of concerns among files as well as their scatter throughout the methods of a class. Figure 4.4 summarizes the correlation between the number of files in which a concern is present and percent overlap for the concerns found by both investigators. In general, there appears to be a downward trend so that the greater the spread of a concern, the smaller the percent overlap. For example, both Investigators M and T had a *Menu* concern, and both tagged code for it in only one file, giving $spread(Menu) = \frac{1}{6}$. The *Menu* concern has an 88.87% overlap in associated code. In contrast, the *Graphics* concern had a spread of $\frac{5}{6}$ and has only a 35.98% overlap.

Counting the number of files with instances of a concern does not accurately reflect scatter throughout the modules of a system. Therefore we also looked at the number of methods in which the commonly identified concerns appeared. Table 4.4 shows the difference between the number of methods the commonly identified concerns appeared in the two investigator's taggings along with

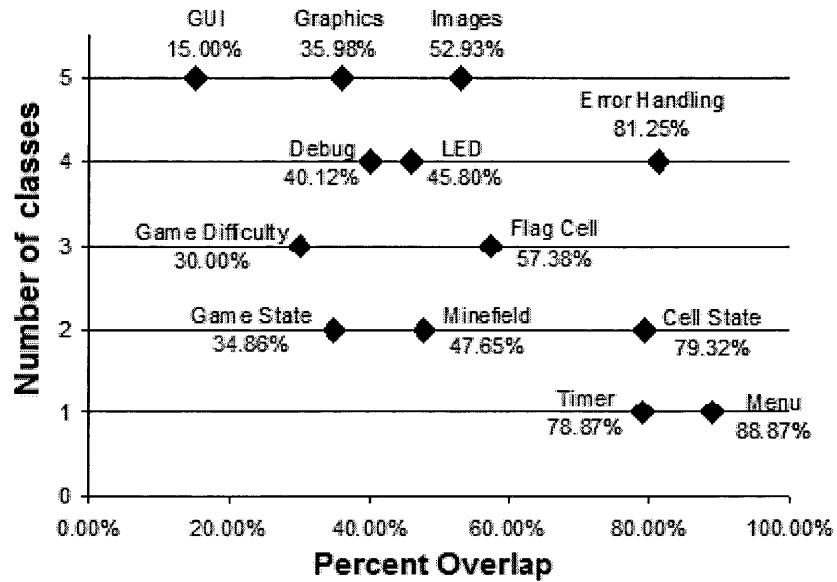


Figure 4.4: Concern overlap vs. number of classes for concerns found by both investigators in the Minesweeper study.

the overlap for the concerns. In general, those concerns that had a relatively low difference in spread had a higher concern overlap. The *Cell State*, *Error Handling*, and *Timer* concerns were identified in the same number of methods by both investigators and had around 80% overlap. At the opposite end of the spectrum, concerns like *Debug*, *Graphics*, and *GUI* had a large difference in the number of methods the investigators found them in and low concern overlap.

Character	Invest. M Scatter	Invest. T Scatter	Concern Overlap
<i>Cell State</i>	15	15	79.87%
<i>Debug</i>	20	4	40.12%
<i>Error Handling</i>	6	6	81.25%
<i>Flag Cell</i>	9	8	57.38%
<i>Game Difficulty</i>	6	15	30.00%
<i>Game State</i>	10	9	34.86%
<i>Graphics</i>	19	10	35.98%
<i>GUI</i>	17	34	15.00%
<i>Images</i>	12	12	52.96%
<i>LED</i>	8	22	45.80%
<i>Menu</i>	5	4	88.87%
<i>Minefield</i>	14	21	47.65%
<i>Timer</i>	5	5	78.87%

Table 4.4: Scatter of concerns among the methods of the Minesweeper classes.

Chapter 5

Generalizing the Case Studies

In this chapter, we discuss the insights gained from the two case studies described above. We deliberate on factors we believe lead to agreement, or even disagreement, between programmers when identifying concerns and their associated source code. We then present a categorization of types of concerns and the guidelines we developed to aid programmers in the identification of concerns in code.

5.1 Factors in Agreement Among Programmers

From our two case studies, we observed there are several factors that possibly contribute to agreement or disagreement among programmers as to what constitutes a concern and where one is located in source code. We list those factors here in order of significance.

Understanding of the program. We believe the extent to which a programmer understands what that program is doing and how a program does it is the most important factor that influences concern identification. In the `sort` case study for instance, we lacked many of Carver and Griswold's concerns primarily because we had a hard time comprehending how `sort` works in detail.

As a result, we did not include some of the more fine-grained concerns of Carver and Griswold such as *IstKeyOptz*, which precomputes the character position of the first key since it is used in every comparison. In the Minesweeper case study, Investigator T did not have a full understanding of the meaning of one of the constants. Recall that to look for code associated with a particular concern, Investigator T used Eclipse's search functionality. She searched based on a full or partial constant or variable name. To look for code related to the *Minefield* concern, she searched for "field." The *Game* class has two constants called `PIXELS_INNER_FIELD_SLOPE` and `PIXELS_FIELD_LEFTMOST` which she came across as results of her search. Investigator T tagged the declaration and uses of these constants without examining their purpose to determine if they actually were related to the *Minefield* concern or just happened to use the word "field."

Knowledge of the programming language. Related to understanding the program, another factor in agreement is knowledge of the language in which the program is written. We have already mentioned how in the *sort* case study, we were unfamiliar with the implementation language and standard libraries. In the Minesweeper case study, Investigator M had previous experience with graphical user interfaces in Java, but Investigator T did not. As a result, Investigator T did not know that classes such as *Frame* and *Canvas* are GUI components and did not tag their uses as such.

Concern abstraction. The fact that programmers think about concerns at different levels of abstraction means it might not be initially evident that a number of concerns identified by two programmers are in fact related. By using concern abstraction, we can discover the cases where we can map several concerns from either programmer to one or more concern of the other programmer. We have seen how even when the set of concerns that one programmer identified may seem very different from the set of concerns of another programmer, there is in fact a great deal of commonality. There were a total of 87 individual concerns identified by only one programmer in the *sort*

case study and 30 in the Minesweeper case study. After mapping those concerns using concern abstraction, those numbers drop to 7 and 3 concerns respectively that do not map to a concern of the other programmer.

Same concern, different ideas. A factor that contributes to disagreement between programmers is when two programmers identify the same concern but have a different idea of the meaning of that concern. For example, Carver and Griswold and us both had concerns dealing with parsing command line arguments. However, command line parsing is not one of the 23 commonly identified concerns. Carver and Griswold's *CmndLine* concern is a meta-concern for the logical but unrealized concern that has no associated code. We did think there was code within `sort` to associate with the *Command line* concern.

As another example, both Investigator M and Investigator T had a *Minefield* concern in the Minesweeper study. However, there was a low percent overlap (47.65%) between their two taggings. Through discussion, we discovered that Investigator M considered the *Minefield* concern to only deal with data structures that represent the minefield in the program. Investigator T's *Minefield* concern was more inclusive because it included data structures and elements of the graphical user interface that pertained to the minefield in her *Minefield* concern.

Program context. Another factor that we found to contribute to disagreement among programmers was the context of a fragment of code in the source. We can best illustrate this point by example. We have already given an example from the `sort` case study where we tagged the body of an `if` block with the *Unique* concern but Carver and Griswold did not. In the Minesweeper study, there is a similar example. Both investigators had a *Flag Cell* concern, but they had conflicting views on how to tag the following code fragment: `if (currentCell.getState() != Cell.STATE_FLAGGED).` Investigator T tagged the condition with the *Flag Cell* concern. However, Investigator M did not tag

the condition with the *Flag Cell* concern because she thought since the condition was checking that the cell was not flagged, this code fragment should not be associated with a concern that deals with flagging cells.

Whitespace and comments. Minor differences such as tagging or not tagging whitespace or comments can lead to more or less concern overlap. There were instances in both case studies where one programmer would include a blank line in the code associated with a concern and the other would not, resulting in reduced concern overlap without causing any real change to the actual code that may or may not be associated with that concern.

5.2 Types of Concerns

Using what we learned from the two case studies, we developed a taxonomy of concern types in order to help programmers more consistently identify concerns in software. We interpret a concern as belonging to one or more of the following categories. With a better idea of the types of concerns that exist in source code, programmers should be able to more easily identify them.

Feature — Functionality a user of the program would be aware of. Examples include all the possible command line flags of `sort` and the different levels of game difficulty in Minesweeper.

Domain Independent Unit of Functionality — An aspect of the code that could appear in any type of program, such as assertions, debugging, and error handling. Both case studies had *Error Handling* concerns. These types of concerns are independent of the purpose of the program.

Input/Output — Anything dealing with input to or output from a program such as `stdin`, `stdout`, reading from or writing to a file or stream, and input received from a graphical user interface. `sort` had numerous input/output concerns: *Files to Sort*, *Close File*, *Output File*, *Open File*, *STDIN*, and

STDOUT to name a few. Some example concerns from the Minesweeper case study are *Keyboard*, *Mouse*, and *User input*.

Internal Program Characteristic — Something a user of a program would not necessarily be aware of, such as the use of buffers or temporaries, the steps taken to parse command line parameters, or optimizations implemented for better performance. The *sort* case study has *Buffers*, *Memory Allocation*, and *1stKeyOptz*. In the Minesweeper case study, examples would be *Thread* and *XY Location*.

Language Characteristic — Elements of a programming language such as constants, accessors, imported/included classes or interfaces, and comments. *sort* has *ProgID*. Example concerns from the Minesweeper case study are *Accessors*, *Constants*, and *Import*.

5.3 Guidelines

Based on the insights we just presented, we developed a set of guidelines that expound upon how to identify concerns and their associated code. Here we explain how the individual guidelines, which are presented in Figures 5.1 and 5.2, were developed. The first six guidelines address identifying concerns in a program. Guideline 1 says, “Before you begin tagging, review the file and look up any unfamiliar constructs of the language.” We hypothesize that many of the differences in the concerns identified by the two parties in each case study was due to a lack of knowledge on one programmer’s part. Carver and Griswold appear to have understood *sort* in much more detail than us. Many of the discrepancies between Investigators M and T were due to the fact that Investigator T had less experience with graphical user interfaces and thus was not able to understand the program as well as Investigator M. This guideline should help future programmers avoid the pitfalls of not being

familiar with such things as standard libraries or the classes used in a graphical user interface.

Guideline 2 states “Identify the main pieces of the program (features); they are concerns.” It was developed based on the fact that in both case studies, a majority of the commonly identified concerns were features. Since features are aspects of a program that would be easily recognizable to a user, they should be as easy to recognize by a programmer. We observed in both studies that these feature concerns, as well as other types of concerns, had user-defined types, global variables, or class attributes associated with them. Hence, the development of Guideline 3: “Constants, user-defined types, class attributes and imported classes are indicators of concerns.” Similarly, we noticed that it was often the case that an entire function or method was tagged with a concern, so we created our fourth guideline: “Entire functions usually relate to a concern or support a concern (except for main).” Functions and methods represent one concept or functionality within a program and are good indicators of possible concerns.

In discussing their sets of identified concerns, Investigators M and T discovered there were times when each was unsure what exactly a concern they had found should encompass. For example, Investigator T decided that the *Game Difficulty* concern should include the code specific to any one level of play, while Investigator M decided to create separate concerns for the different levels. Each investigator was consistent once she made her decision, otherwise it would have been very hard to understand the purpose of each concern. We developed Guideline 5 as a way to help programmers strive for consistency. As another example of the usefulness of consistency, consider Investigator M’s *Connected Mines* and *Neighbor Mines* concerns. After discussing her concerns with Investigator T, she realized that these two concerns are just two different names for the same thing. If Investigator M had a more well-defined idea of the meaning of each concern, she could have avoided the duplication.

Finally, Guideline 6 says “Look for domain independent concerns such as debugging and error handling.” We added it to the concern identification guidelines as a reminder to look for concerns that are not specifically tied to the functionality of the program. While many of the concerns in both case studies were features, each programmer did find a few concerns that were not. Examples include *POSIX* and *Temp Files* in the `sort` case study and *Constants* and *Debug* in the the Minesweeper study.

While the first six guidelines are meant to help programmers identify concerns, the final 11 guidelines can be used to help programmers locate concern code. Guidelines 7-17 were developed based on observations of trends and patterns seen in many cases of a fragment of code tagged in an identical or near-identical manner with a commonly identified or similar concern by both parties. Sometimes, the way in which the characters of the associated code were tagged differed slightly, so we created the concern identification guidelines to promote consistency.

Guideline 7, “Different levels of concerns can be tagged in the same code fragment,” was created with the concern hierarchy in mind. If a programmer recognizes that a fragment of code belongs to two different levels of concerns, *Game Difficulty* and *Easy game* for example, that fragment should be tagged with both concerns. By associating the code fragment with both concerns, the programmer is essentially documenting that there is a connection between the two concerns.

Guideline 8 is about tangled concerns and says, “Even though a code fragment is tagged with one concern, it can be tagged with another concern.” Tangling of concerns occurs when two or more concerns are implemented in the same component, thus they are tightly coupled. We developed this guideline based on the fact that both Investigator M and T noticed that while looking for associated concern code, they had a tendency to not consider code that had already been tagged with a different concern. Programmers should not fall into this habit of ignoring the possibility of tangling because

they might miss code that is important to a concern simply because it is already associated with another concern.

Guideline 9 states, “Use the search feature to find code that is possibly related to the current concern of interest, but take the time to figure out the context of the code before tagging it.” The guideline was spurred by the fact that when Investigator T used the search tool to identify concerns, the context of the instance was rarely examined, leading to many taggings of simply an instance of a variable that had little affect on the code or function in which it was contained. Recall the example from Section 5.1 where Investigator T searched for “field.”

Guidelines 10-16 promote consistency in tagging of code with concerns in hopes of increasing the concern overlap between two programmers. The guidelines specify how a programmer should tag conditional and iterative statements, declarations and uses of variables, parameters and arguments, and comments. Like the other guidelines, these seven guidelines are based on common conventions exhibited by the programmers in both studies. For example, we noticed the programmers almost always associated comments with the concern of the code directly succeeding them. Appendix A provides examples to illustrate the meaning and use of these guidelines.

Our final guideline suggests that “Most to all of the code in a file should be tagged.” Carver and Griswold associated every one of `sort`’s 2145 lines with at least one concern, while we left 120 lines untagged. Of these, 83 were blank lines, 22 were comments, and only 15 contained code. The untagged code included a few `#include` and `#define` statements as well as the declaration of `main` and a few of its variables. In the Minesweeper case study, Investigator M had 243 untagged lines of code, and Investigator T had 304 in a program of 2776 lines. Again as in the case with `sort`, the majority of the untagged lines were blank and the others were comments or class or method headers with very little actual code left not associated with a concern. It makes sense that almost all

of the code in each program was related with at least one concern. Presumably, every line of code is meaningful and useful, otherwise it would not be present in the program.

Concern Identification Guidelines

1. Before you begin tagging, review the file, and look up any unfamiliar constructs of the language.
2. Identify the main pieces of the program (features); they are concerns.
3. Constants, user-defined types, class attributes and imported classes are good indicators of concerns.
4. Entire functions usually relate to a concern or support a concern (except for main).
5. When you create a concern, decide what it encompasses. For example, if a program is created to check if the current date corresponds to a birthday of someone stored in a database, should a *birthday* concern encompass the Boolean value of whether the current date is someone's birthday, or should it relate to the String value representing the date of the person's birthday.
6. Look for domain independent concerns such as debugging and error handling.

Figure 5.1: Concern Identification Guidelines

Concern Tagging Guidelines

7. Different levels of concerns can be tagged in the same code fragment.
8. Even though a code fragment is tagged with one concern, it can be tagged with another concern.
9. Use the search feature to find code that is possibly related to the current concern of interest, but take the time to figure out the context of the code before tagging it.
10. If a function is tagged with a concern, the calls to it should also be tagged.
11. If the whole body of an `if` or `switch` statement is tagged, tag the `if` or `switch` as well as the beginning and ending braces.
12. If the whole body of a loop is tagged, tag the loop conditionals as well as the beginning and ending braces.
13. Make sure to tag both the declaration and use of variables associated with a concern.
14. When a variable is an argument or parameter to a function, tag only the argument or parameter and associated type.
15. Tag the whole expression when it affects a concern variable. When a concern variable is used on the right side of an assignment statement, tag only the use of that variable.
16. Whitespace, new lines and comments should be included when tagging concerns.
17. Most to all of the code in a file should be tagged.

Figure 5.2: Concern Tagging Guidelines

Chapter 6

Related Work

This chapter explores work that is related to the research presented in this thesis. We discuss areas in which our work could be put to beneficial use. We also examine work that has been done in the areas of identifying concerns and finding concern code.

The work most closely related to ours is an exploratory study by Lai and Murphy to investigate how different concerns interact [10]. They used a tool similar to Spotlight called Feature Selector to mark and analyze concerns in Java source code. In their work, they state some criteria for how they decided something was a feature (their word for a concern). Their criteria included standards conformation for the FTP and regular expression programs they examined, input/output, and parts of the code a programmer might want to change or remove. Our work has gone further in this direction to explore other types of concerns. They also remark that it was difficult to determine what code to relate to a concern and how to be consistent. Our work in developing guidelines can help provide that needed consistency.

One area where our work could have a significant impact is aspect-oriented programming. AOP facilitates the modularization of concerns that techniques such as object-oriented programming can-

not. These concerns that AOP deals with are cross-cutting; they are tangled and scattered across the modules of a system. These cross-cutting concerns are encapsulated in an *aspect* that is integrated with the rest of the system with the help of a specialized compiler called a *weaver*. AOP does not specify how to determine what within a program should be made an aspect. Our work may aid programmers using AOP in identifying candidates for aspects.

Software plans [2] is a tool-based approach for separation of concerns. Using a plug-in to the Eclipse integrated development environment, programmers can create concerns and associate fragments of source code with those concerns, much like we did in our case studies. Programmers can then elect to abstract away concerns by making them hidden or irrelevant, essentially removing their associated code from view. Our work on identifying concerns and related code fits nicely with the software plans approach. Programmers could use the insights and guidelines of our work to create useful and effective software plans.

6.1 Identifying Concerns

We have only considered the identification of concerns in existing systems. However, the identification of concerns is not limited to the implementation or maintenance phases of the software life cycle. Rashid et al. [15] propose a model for aspect-oriented requirements engineering. Their approach uses viewpoints [5] and focuses on cross-cutting concerns that increase the complexity of code and the difficulty of understanding and maintaining it. They argue that considering aspects at the requirements engineering level can provide better support for separation of concerns and lead to software that is more adaptable. The model calls for the identification of concerns from the requirements and then their specifications. Next, the concerns are related to the requirements, and

then the candidate aspects are identified. Candidate aspects are those concerns that crosscut several requirements. The candidate aspects are then specified to make them more concrete and to reveal interactions between them. Finally, candidate aspects' influences on the later stages of development is specified.

6.2 Finding Concern Code

In our work, we make a distinction between a concern as a concept and the code that implements that concern. In this section, we discuss some existing techniques for finding concern code that do not address how to identify the related concern.

Robillard and Murphy [19] extended the Eclipse platform to include an algorithm to automatically infer concern code from transcripts of the source code a programmer viewed while investigating a concern. This technique is intended to reduce the amount of time and effort needed to complete a program evolution task. While a programmer investigates existing source code in order to implement a change, a transcript records all the code that is visible to the programmer. Once the change is completed, the programmer can view a list of descriptions of the concern that the inference algorithm believes are relevant based on what the code the programmer looked at. A concern description consists of the methods that were visible during the investigation. The programmer can then select the most appropriate description, give it a name, and save it in a concern database. When a new programmer needs to make a change to the same concern in the source code, he or she can consult the concern database to get the saved concern description, saving the effort of investigating the code on their own.

Robillard and Murphy's tool for locating concerns differs from ours in several ways. First,

the unit of granularity in their approach is a method declaration, while with Spotlight, we allow individual characters to be associated with a concern, so our approach is much more fine-grained. In our work, we observed that fragments of code within functions or methods were associated with concerns. Second, their algorithm can only infer concern code from an investigation transcript, which can lead to false positives if a programmer examines code unrelated to the concern. Our manual approach to tagging concern code gives the programmer complete control over the code they wish to associate with a concern.

Robillard and Murphy [20] also developed a plug-in for the Eclipse platform called Feature Analysis and Exploration Tool (FEAT). A concern in FEAT is any fragment of a program consisting of classes, methods, or fields of interest to the programmer. FEAT allows the user to interactively build concern graphs [18] by exploring program structure and program element relationships and iteratively expanding the body of code associated with a concern. Their work is similar to ours in that they have tool support to locate concern code, but their approach is automated. However, it again lacks the granularity of our manual approach to finding concern code because FEAT only allows the inclusion of classes, methods, and fields.

Robillard et al. [17] did a study to examine the connections between program investigation behavior and success at a software modification task. Program investigation behavior is the way in which a programmer navigated through source code in pursuit of some goal, in this case implementing a change. The success of the programmer at completing the modification task was measured against the number of sub-tasks the programmer implemented that met the requirements and did not contain faults. They presented five programmers with the same change task—modifying the auto-save feature of jEdit, a text editor written in Java. Their results indicate that a more methodical investigation of source code is more effective than a systematic line-by-line investigation and that

programmers should have a plan when performing an investigation. This work focuses on investigating source code for program maintenance and not concern identification. However, the change task and sub-tasks could be considered possible concerns. Their observations on effective program investigation behaviors for change tasks easily could be applied to the task of locating code associated with a concern.

Information transparency [6] identifies scattered but related sections of code using inference and search mechanisms. If a programmer needs to make a change to the source code regarding a specific concern, he or she can use information transparency to lexically (based on naming conventions) and syntactically (based on characteristics such as loop structure) find the code pertaining to the change to be made. The fact that Investigator T searched for concern code using names of variables supports the information transparency approach. Aspect mining [22] is a method of advanced separation of concerns that automatically identifies cross-cutting concerns in software systems. Approaches for finding code related to a concern can be text-based (i.e. pattern matching) or type-based [7]. Information transparency or aspect mining could have potentially reduced the amount of time it took the investigators to locate and annotate concern code in our case studies by reducing the amount of code they had to consider.

Program slicing [24] attempts to reduce the complexity of code by selecting only those lines of code that have an effect on a particular variable. This approach could be used to locate code associated with a concern, but the results would most likely be undesirable. Program slices can be very large and include almost the entire program, while most code associated with a single concern may be a relatively small fragment of the source code. Also, it is not always the case that a program variable correlates to a single concern; a variable may relate to multiple concerns in a program.

Chapter 7

Evaluation and Conclusion

In this thesis, we have presented the results of two case studies that provide some insight into how programmers think about concerns and the factors that contribute to consistent identification of concerns among programmers. While there is no “right” or “wrong” way to identify concerns and their associated code, we believe that the guidelines we have developed based on observed similarities and trends can ease the difficulty of identifying concerns and improve the consistency of the concerns and code found by individual programmers. Clearly, experimental validation of these guidelines is an important area of future work.

Our results indicate that programmers think at different levels of abstraction for different concerns. We hope that our guidelines can help create some consistency in this regard. With more agreement on what constitutes a concern, programmers can potentially communicate more effectively because they will be thinking at the same or closer levels of abstraction. However, it is clear that this is an interesting issue that deserves further study.

The two case studies we have performed involve existing code that was unfamiliar to the programmers. Similar studies involving code developed by the programmers identifying the concerns

would complement the research we have presented here. Clearly, the impact of code unfamiliarity would be greatly reduced, and other unknown factors may also arise.

There are several threats to the validity of our work. First, our two case studies were relatively small in terms of the the number of participants and the small size of the code base. The small scale is intentional for several reasons. First, the case studies are preliminary work necessary for future, more in-depth studies involving more participants to see if the generalizations and guidelines we developed are valid. Second, the investigators in our case studies exhaustively identified concerns and associated code in the sample programs which is a very time-consuming task. We needed 10 hours to understand, identify, and tag concerns in `sort`. We do not know how much time Carver and Griswold needed. For the Minesweeper case study, Investigator M took 6.5 hours, and Investigator T took 8 hours. If the systems used in the case studies had been much larger and complex, it would have taken the programmers significantly more time to completely and thoroughly identify all concerns and their associated code. In practice, programmers would not be as meticulous as the investigators were in trying to identify all instances of all concerns.

Another possible threat to our case studies was communication or collusion between the investigators to identify the same concerns. In the `sort` case study, we were not given Carver and Griswold's concern information until after we had finished identifying concerns on our own. In the Minesweeper case study, we eliminated this danger by strictly disallowing any discussion between the two investigators on the topic of the case study until after they had both completed their identifications.

A final threat to our work is the use of bad metrics. Our concern overlap metrics may be sensitive to the inclusion of whitespace in the manifestation of a concern. Blank lines and other whitespace do not change the meaning of a program, but they can skew our metrics if one programmer prefers

tagging whitespace and another does not. We would need to do more analysis to determine the impact of whitespace on concern overlap.

This work has several contributions to the field of software engineering. As far as we know, our case studies are the first work to compare concerns and concern code identified by different programmers. The data from these case studies on the presence and relationships among concerns in code is our first contribution. Our second contribution is a categorization of the different types of concerns that the investigators found in real systems. Another contribution is exploration of the factors that are responsible for consistent concern identification among programmers. Finally, our last contribution is the set of guidelines on how to identify and locate concerns in source code.

Appendix A

Illustrations of Guideline Usage

In this appendix, we give some illustrative examples of the use of our taggings guidelines to make their meanings more clear. We do not present examples for all of the concern identification guidelines because they are tips on how to identify the presence of a concern. We also exclude examples for a number of the concern tagging guidelines because we feel they are self-explanatory.

A.1 Guideline 10

```
private static void bDayMsg(Student s){
    System.out.println("*****");
    System.out.println("It's " + s.firstName + " " + s.lastName() + "'s birthday!!!");
    System.out.println("*****");
}

private static void tokenize(String line){
    ...
    Student s = new Student(lastName, firstName, userName, dob);

    if (isBDay(s)){
        bDayMsg(s);
        birthday = true;
    }
}
```

Figure A.1: Illustration of the use of Guideline 10.

Figure A.1 shows the proper use of Guideline 10, “If a function is tagged with a concern, the calls to it should also be tagged.” Since the programmer tagged the definition of the `bdayMsg()` method, the call to it in `tokenize()` is also tagged with the same concern.

A.2 Guideline 11

<pre> if (!birthday){ System.out.println("No birthdays today."); } </pre>	<pre> if (!birthday){ System.out.println("No birthdays today."); } </pre>
(a) Improper usage.	(b) Proper usage.

Figure A.2: Illustration of the use of Guideline 11.

Guideline 11, which states, “If the whole body of an `if` or `switch` statement is tagged, tag the `if` or `switch` as well as the beginning and ending braces.” Figure A.2 improper and proper application of this guideline.

A.3 Guideline 12

<pre> while (b.ready()){ studentInfo = b.readLine(); if (!studentInfo.equals("")) tokenize(studentInfo); } </pre>	<pre> while (b.ready()){ studentInfo = b.readLine(); if (!studentInfo.equals("")) tokenize(studentInfo); } </pre>
(a) Improper usage.	(b) Proper usage.

Figure A.3: Illustration of the use of Guideline 12.

Guideline 12 is very similar to Guideline 11, except it deals with iterative statements. Figure A.3 shows the incorrect and correct way to tag a loop according to the guidelines.

A.4 Guideline 13

```

| String studentInfo;
| birthday = false;
| b.readLine();
| b.readLine();
| while (b.ready()){
|     studentInfo = b.readLine();

```

Figure A.4: Illustration of the use of Guideline 13.

Figure A.4 demonstrates Guideline 13, “Make sure to tag both the declaration and use of variables associated with a concern.” Since the declaration of the variable `studentInfo` has been tagged with the concern, its use in the `while` loop should also be tagged.

A.5 Guideline 14

```

| private static void read(BufferedReader b) throws IOException{
|     String studentInfo;
|     birthday = false;
|     b.readLine();
|     b.readLine();

```

Figure A.5: Illustration of the use of Guideline 14.

Guideline 14 states, “When a variable is an argument or parameter to a function, tag only the argument or parameter and associated type.” Figure A.5 shows an example of the meaning of this guideline. The parameter `b` to the `read()` method and its uses within the method are all tagged with the concern for consistency.

A.6 Guideline 15

```
|| studentInfo = b.readLine();
```

(a) Improper usage.

```
|| studentInfo = b.readLine();
```

(b) Proper usage.

Figure A.6: Illustration of use of Guideline 15.

Figure A.6 depicts Guideline 15, which says “Tag the whole expression when it affects a concern variable. When a concern variable is used on the right side of an assignment statement, tag only the use of that variable.” In this case, `b` is the concern variable. Since it is being used on the right side of the expression, it only its use and not the full line should be tagged with the concern.

A.7 Guideline 16

```
|| // A class holding a student's first and last name,  
|| // user name and date of birth  
|| private static class Student(  
|| private String lastName;  
|| private String firstName;  
|| private String userName;  
|| private String dob;
```

Figure A.7: Illustration of the use of Guideline 16.

Guideline 16 specifies that “Whitespace, new lines and comments should be included when tagging concerns.” Figure A.7 illustrates the inclusion of comments with the code tagged with a concern.

Bibliography

- [1] LEE CARVER AND WILLIAM G. GRISWOLD. Sorting out concerns. In *OOPSLA '99 Workshop on Multi-Dimensional Separation of Concerns*, November 1999.
- [2] DAVID COPPIT AND BENJAMIN COX. Software plans for separation of concerns. In *Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Lancaster, UK, 22 March 2004.
- [3] EDSGER W. DIJKSTRA. *A Discipline of Programming*. Prentice Hall, 1976.
- [4] ECLIPSE.ORG. The Eclipse homepage. URL: <http://www.eclipse.org/>.
- [5] BASHAR NUSEIBEH ANTHONY FINKELSTEIN. Viewpoints: A vehicle for method and tool integration. In *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*, Montreal, Canada, 1992.
- [6] WILLIAM G. GRISWOLD. Coping with software change using information transparency. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.
- [7] JAN HANNEMANN AND GREGOR KICZALES. Overcoming the prevalent decomposition in legacy code. In *ICSE 2001 Workshop on Advanced Separation of Concerns in Software Engineering*, 15 May 2001.
- [8] IEEE. Ieee recommended practice for architectural description of software-intensive systems. *IEEE Std. 1471-2000*, September 2000.
- [9] GREGOR KICZALES, JOHN LAMPING, ANURAG MENDHEKAR, CHRIS MAEDA, CRISTINA VIDEIRA LOPES, JEAN-MARC LOINGTIER, AND JOHN IRWIN. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.
- [10] ALBERT LAI AND GAIL C. MURPHY. The structure of features in java code: An exploratory investigation. In *OOPSLA '99 Workshop on Multi-Dimensional Separation of Concerns*, November 1999.
- [11] GAIL C. MURPHY, ALBERT LAI, ROBERT J. WALKER, AND MARTIN P. ROBILLARD. Separating features in source code: An exploratory study. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 275–85, Toronto, Canada, 12–19 May 2001. IEEE.

- [12] BASHAR NUSEIBEH, JEFF KRAMER, AND ANTHONY FINKELSTEIN. A framework for expressing the relationships between multiple views in requirements specification. *Software Engineering*, 20(10):760–773, 1994.
- [13] H. OSSHER AND P. TARR. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.
- [14] D. L. PARNAS. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–8, December 1972.
- [15] AWAIS RASHID, PETER SAWYER, ANA MOREIRA, AND JOO ARAJO. Early aspects: a model for aspect-oriented requirements engineering. In *IEEE Joint Conference on Requirements Engineering*, Essen, Germany, September 2002.
- [16] MARTIN P. ROBILLARD. *Representing Concerns in Source Code*. PhD thesis, University of British Columbia, November 2003.
- [17] MARTIN P. ROBILLARD, WESLEY COELHO, AND GAIL C. MURPHY. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, December 2004.
- [18] MARTIN P. ROBILLARD AND GAIL C. MURPHY. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, pages 406–417, 19–25 May 2002.
- [19] MARTIN P. ROBILLARD AND GAIL C. MURPHY. Automatically inferring concern code from program investigation activities. In *Proceedings of 18th International Conference on Automated Software Engineering*, pages 225–234, 06–10 October 2003.
- [20] MARTIN P. ROBILLARD AND GAIL C. MURPHY. Feat a tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, 3–10 May 2003. IEEE.
- [21] STANLEY M. SUTTON, JR. AND ISABELLE ROUVELLOU. Modeling of software concerns in cosmos. In *AOISD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 127–133. ACM Press, 2002.
- [22] TOM TOURWE AND KIM MENS. Mining aspectual views using formal concept analysis. In *Proceedings. Source Code Analysis and Manipulation Workshop*, September 2004.
- [23] C. REID TURNER, ALFONSO FUGGETTA, LUIGI LAVAZZA, AND ALEXANDER L. WOLF. Feature engineering. In *Proceedings of the 9th international workshop on software specification and design*, pages 162–164, 1998.
- [24] MARK WEISER. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–7, 1984.

VITA

Meghan Revelle

Meghan Kathleen Revelle was born in Columbia, Maryland on January 21, 1981. She graduated from Atholton High School in 1999 and then went to Mary Washington College (MWC) where she studied computer science. During the summer of 2002, Meghan was a Research Experiences for Undergraduates Fellow at Duke University. In 2003, she earned her B.S. in Computer Science from MWC. She entered the department of computer science at the College of William and Mary in 2003 and is continuing to pursue her Ph.D.