Understanding Concerns in Software: Insights Gained from Two Case Studies

Meghan Revelle, Tiffany Broadbent, and David Coppit
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23187-8795
{*meghan, tbroadbe*}*@cs.wm.edu, david@coppit.org*

Abstract

Much of the complexity of software arises from the interactions between disparate concerns. Even in well-designed software, some concerns can not always be encapsulated in a module. Research on separation of concerns seeks to address this problem, but we lack an understanding of how programmers conceptualize the notion of a concern and then identify that concern in code. In this work, we have conducted two exploratory case studies to better understand these issues. The case studies involved programmers identifying concerns and associated code in existing, unfamiliar software: GNU's `sort.c` *and the game Minesweeper. Based on our experiences with these two case studies, we have identified several types of concerns and have detailed 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.*

1. Introduction

A key problem with software is that it can become very complex, and much of this complexity can be derived from the interaction of concerns. Techniques for separation of concerns seek to cleanly disconnect concerns from source code in order to reduce complexity and increase comprehensibility [5, 10]. As Murphy and Lai note, many of the approaches for separation of concerns are still maturing, so there is no widely-accepted definition of what constitutes a concern [8]. There is an intuitive understanding of concerns, but a concrete definition is hard to come by [15].

Of the definitions provided by researchers, most are very broad and general. Robillard describes a concern as "any consideration ... about the implementation of a program" [11]. 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 [10]. For example, views can be used in the requirements phase to address only the criterion (concern) of interest [9]. Sutton [15] 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.

Lai and Murphy [7] as well as Turner et al. [17] have a more specific definition in which concerns are considered to be features—a functional property of a system that is visible to the user. However, it is clear that there are important non-feature concerns, such as performance. Aspect-oriented programming (AOP) [6] is a specific instance of separation of concerns that modularizes concerns that cross-cut a system's functionality, such as memory access patterns. AOP terms these units that are not a part of the system's functional decomposition *aspects*. Clearly, there is no consensus on the definition of a concern since researchers' definitions range from the vague "any consideration" to functional properties such as features to cross-cutting aspects.

We believe that this lack of consensus is due at least in part to our lack of understanding of how programmers think about concerns and identify them in source code. While we believe that flexibility in the notion of "concern" is useful, a clearer understanding of possible types of concerns would be a valuable guide for programmers and would help clarify 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. Once a programmer does decide on the presence of a concern, how does he or she identify the full manifestation of that concern in all of the source code? 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 programmers identified concerns and the code associ-

ated with them. In both case studies, we compared concern code found by two investigators, levels of abstraction, and concern spread in order to better understand the causes of similarity (or lack thereof) in concern identification to gain insight how different people come to similar conclusions about concern identification.

There are several contributions of this work. The first contribution is a classification of different types of concerns. The second contribution is insights gained regarding factors that contribute to consistent concern identification. The third contribution is a set of guidelines that can help programmers to more reliably and consistently identify concerns in existing source code.

In Section 2, we describe our two case studies. In Section 3, we discuss what we learned from these studies. Section 4 presents the guidelines we developed. Section 5 covers work related to ours, and Section 6 concludes.

2. Case Studies

In this section, we introduce Spotlight, the tool we used to associate concerns with source code. We also describe the two case studies that were the basis for developing our concern definition and guidelines.

2.1. Spotlight

Figure 1 shows a screenshot of Spotlight, an Eclipse [2] that we developed and used in both our case studies. Spotlight allows the programmer to annotate fragments of source code as belonging to a concern. We refer to this process as tagging code with a concern. A fragment of code may be annotated with more than one concern. To display the annotations or taggings, 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." Figure 1 also shows the context menu that is displayed when the user selects a fragment of code and right-clicks on it. This menu allows the user to easily edit the annotations for a block of code. 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. The user can also tell the tool to underline the particular characters in the code that are associated with a concern.

2.2. Case Studies

Here we describe the two case studies we completed in order to gain a more precise understanding of concerns and

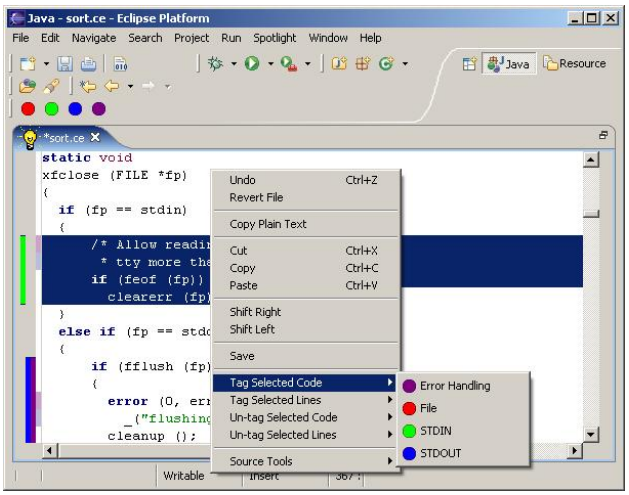

Figure 1. In the Spotlight editor, the programmer is using the context menu to annotate part of the code with a concern.

their location in source code. We present the programs we examined, discuss the concerns identified by two investigators, and compare the concerns and their code. We will provide a more in-depth analysis of the factors in concern identification in Section 3. These case studies are intentionally relatively small for several reasons. First, they are preliminary work necessary for future, more in-depth studies. Second, the investigators exhaustively identified concerns and associated code in the sample programs which is a very time-consuming task.

2.2.1. GNU `sort.c`. The GNU `textutils-1.22` implementation of `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. The command line flags of `-c` and `-m` 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 "phone directory order" or to ignore non-printing characters. For this case study, we compare concerns identified in `sort` by one of the authors (Investigator M) to concerns found by Carver and Griswold [1].

Investigator M identified 50 concerns in `sort.c`. Many 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 re-

lated to program characteristics such as the use of assertions, buffers, temporary files, POSIX compliance, or signal handling. Investigator M was not very familiar with the implementation language. She had used C previously but not extensively and did not have much knowledge of the standard libraries.

To better understand the subjective nature of programmer identification of concern code, we compared Investigator M's concerns to those of Carver and Griswold, who used the same implementation of `sort` in their work. One difference was in the number of concerns found—they had 83 concerns compared to Investigator M's 50. The majority of the additional concerns Carver and Griswold had relate to system-specific issues that Investigator M did not address, such as access of the system environment space and releasing the thread of execution to the operating system. However, there were 23 commonly identified concerns between the two parties—mostly user-level features.

A second difference was the level of abstraction. Not all of the concerns that only one programmer identified were unrelated. For example, Carver and Griswold created meta-concerns to group related concerns, and these meta-concerns had no associated code. For example, Carver and Griswold created a *Modes* concern to encompass `sort`'s three modes of operation: sorting files, merging files, and checking if files are already sorted. Investigator M had individual *Sort*, *Merge*, and *Check* concerns, but did and not see the need to create a higher level concern such as *Modes*. In contrast, in some cases Carver and Griswold used multiple concerns where Investigator M used a single concern. They identified a *Month Order* concern that deals with sorting dates by month and an *Upper Month* concern that translates month names to upper case letters. Investigator M had only a *Month Order* concern which included the code for converting month names to upper case, but she did not think such a small feature warranted a concern because its total associated code was only one line.

2.2.2. Minesweeper. For our second case study, we considered a Java implementation of the game Minesweeper that is approximately 3000 lines contained in six classes. One class controls the logic of the game, and the remaining five deal with the graphical user interface. 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 game ends when the user correctly identifies all of the cells not containing mines or clicks on a cell containing a mine. For this case study, two of the authors (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.

For the results of the Minesweeper case study, we were able do more analysis because the two investigators could discuss their reasons for identifying and tagging concerns in certain ways. Investigator M found 30 concerns while Investigator T found 26 concerns. We compared both sets of concerns and found 13 out of the total 43 concerns were identified by both investigators.

Each investigator began by tagging the six Minesweeper files: `Game.java`, `MinesweeperWindow.java`, `LED.java`, `CustomFieldDialog.java`, `LEDPanel.java`, and `Cell.java`. Investigator M approached the task of identifying concerns by beginning in `Game.java` since this file deals with the logic of the Minesweeper game. Investigator M's strategy was to identify a concern in `Game.java` and then look for that concern in the other five files. Investigator T began tagging in `Game.java`, primarily because this is the longest file the Minesweeper suite, and thus she expected it to yield the most concerns. Investigator T identified all concerns in `Game.java` and then proceeded to look for concerns 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.

Once an initial tagging of the code was complete, both investigators felt the need to go back through the code to ensure they had found all the fragments that belonged to a concern. Investigator T searched for keywords to find code related to a concern that she had missed during the first pass of taggings. Investigator M reviewed her taggings to see if she missed any concern code but did not use the search functionality. Instead, she simply scrolled through the code. 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.

3. Analysis of Identified Concerns and Code

In this section, we explore the concerns found in the case studies and their associated source code in more depth. We explain concern overlap, the metric we developed to measure the similarity between code one programmer associates with a certain concern to the code another program-

Table 1. Concern overlap in `sort`.

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

mer associates with that same concern. We also discuss how our case studies led us to the observation that programmers think about concerns on several different levels of abstraction, and it is possible that one or more higher level concerns subsume several lower level concerns. Finally, we explore several correlations with concern overlap. The ultimate goal of these case studies was to gain insight into the ways programmers think about concerns in order to create guidelines on concern identification and how to annotate associated source code.

3.1. Concern Overlap

Since we were interested how programmers associate a concern with code, one of our colleagues implemented new features in the Spotlight tool for us to use in our analysis of concern code. These new features compute concern intersection and subtraction to aid us in comparing code tagged as part of one set of concerns to code tagged with another group of concerns. A concern group is a selection of concerns that are related. For example, one programmer may have an *Event Listener* concern while another programmer has two concerns, one for *Keyboard Events* and another for *Mouse Events*. The concern intersection or subtraction features allow a user to group the *Keyboard Events* and *Mouse Events* concerns into a single concern group and compare the combined taggings of both concerns to the annotated code of the *Event Listener* concern.

Table 2. Concern overlap in Minesweeper.

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

These new Spotlight features perform a character-by-character comparison of these concern groups 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 sets of concerns. 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:

$$concern\_overlap(c_1, c_2) = \frac{|c_1 \cap c_2|}{|c_1 \cup c_2|} * 100 \qquad (1)$$

In the `sort` case study, there were seven concerns with 100% overlap in what Investigator M and Carver and Griswold tagged, as can been seen in the character overlap column of Table 1. The average concern overlap was 82.52%. The overlap between concerns identified by the two investigators in the Minesweeper case study was on average much lower (52.93%) for the thirteen commonly identified concerns shown in Table 2. The Minesweeper case study did not yield any 100% concern overlaps, as the `sort` study did. We believe there was more overlap in the `sort` case study because `sort` is a more feature-oriented program, and it is easier to identify code fragments that implement each individual feature. We discuss some reasons for good and poor concern overlap in Section 4.

In addition to investigating concern overlap for individual concerns, we also explored concern overlap for concern groups. In the `sort` study, we were able to create 39 concern groups. For example, we combined Carver and Griswold's *Reverse Global* and *Reverse Key* concerns to compare them with Investigator M's *Reverse* concern. 25 out of the 39 concern groups had better than 80% overlap, with an average of 78.34%. In the Minesweeper study, there were 19 concern groups, and only three had an overlap better than 80%, giving us further evidence that it is easier to identify concern code in feature-rich programs like `sort`.

Realizing that the concern overlap metric could be skewed by minor differences such as one investigator tagging newlines and other whitespace while the other did not, we also looked at line overlap. 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 each case study, as shown in Tables 1 and 2. We attribute the cases where the line overlap was lower than the character overlap to one investigator tagging blank lines or lines of comments that other did not. For the rest of the paper, we consider only character overlap.

3.2. Concern Abstraction

After we completed tagging for the two case studies, we examined the two sets of concerns identified by both investigators in each study. We noticed that even though some concerns had different names, they were related. For instance, the *Month Order* and *Upper Month* example from the `sort` case study above where all of the *Upper Month* concern is included in the *Month Order* concern. We did a similar analysis for all the concerns identified by considering all combinations of concerns and deciding which were related and how. For the `sort` case study, our decision as to whether concerns were related was based on the names of the concerns and reviewing the taggings. In the Minesweeper case study, the two investigators were able to discuss their concerns to determine the relationships.

We found we could map all of Investigator M's concerns to one or more of Carver and Griswold's concerns and all but seven of Carver and Griswold's concerns to a group that conceptually matched one or more of Investigator M's concerns. We call this mapping of related groups of concerns *concern abstraction*. The seven remaining, unmapped concerns of Carver and Griswold are either empty meta-concerns (they have no associated code), are related to system services which Investigator M did not consider to be concerns, or in one case Carver and Griswold have a concern comprised entirely of the comments at the top of the file and Investigator M does not.

For more detailed examples of concern abstraction, we turn to the Minesweeper study. 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 a concern abstraction hierarchy, shown in Figure 2.

Each object in Figure 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 included 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. We assigned numbers subjectively so that a higher ranking means that the concern is broader and more vague.

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 all 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 code 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 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 them as such.

3.3. 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 taggings, as shown in Figure 3. In

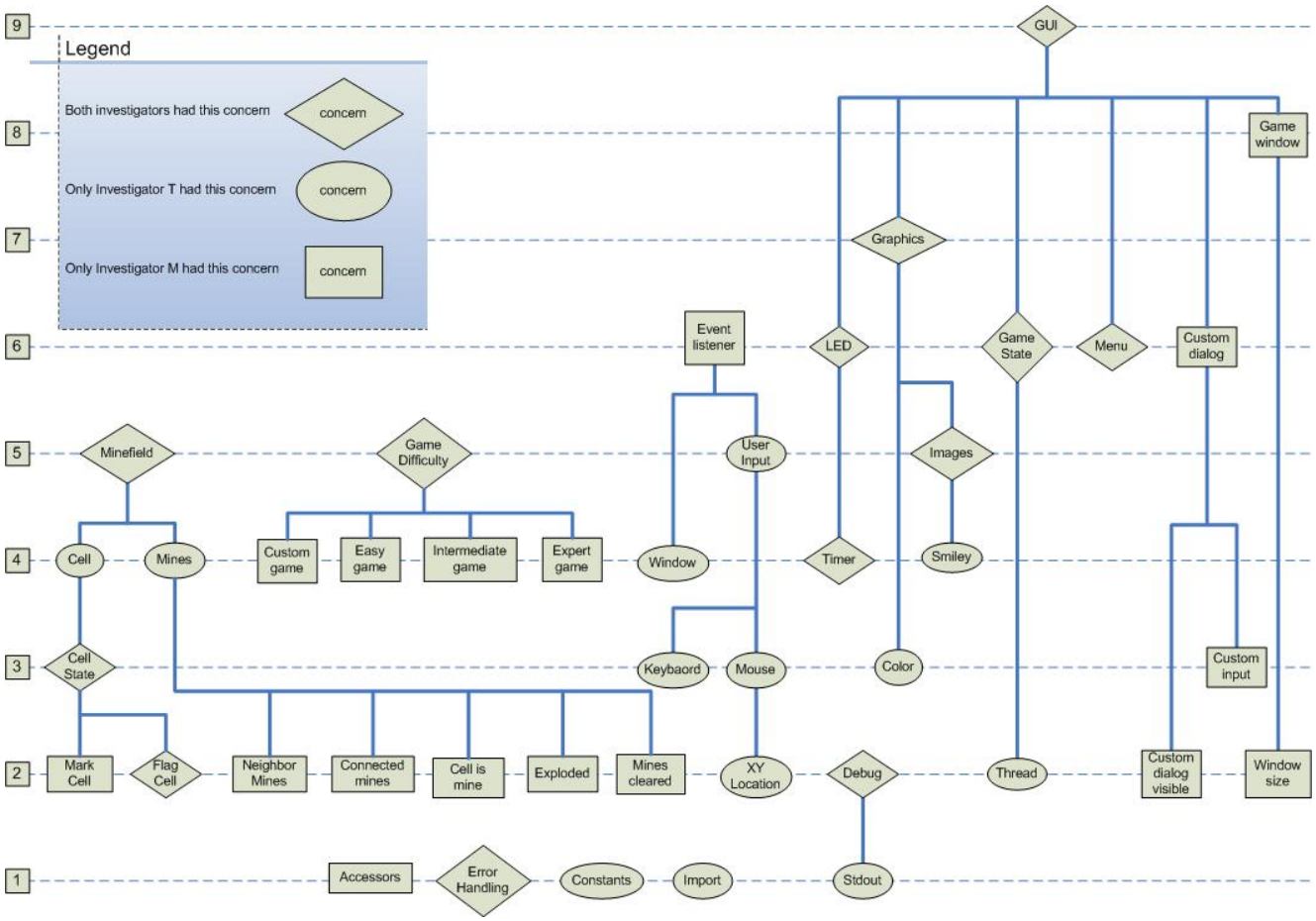

Figure 2. Hierarchy of concerns in the Minesweeper program.

general, the concerns with a higher ranking have lower 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. Similarity, the concerns at lower levels of abstraction tend to have higher percent overlap. *Error Handling* is at level 1 and has 81.25% overlap and *Cell State* is at level 3 and has 79.32% overlap.

However, while there seems to be a general correlation, the correlation seems to be weak. The *Debug* and *Menu* concerns, for instance, seem to be outliers for a general inverse correspondence between percent overlap and level of abstraction.

3.4. Spread

Finally, we borrow Lai and Murphy's spread metric [7], replacing "feature f" in their definition with "concern c" and "files" with "classes"

$$spread(c) = \frac{\text{\# of classes containing concern c}}{\text{total \# of classes}} \quad (2)$$

This metric is useful when used in conjunction with our concern overlap metric because there is a correlation between the number of files that contain a concern and the amount of overlap between commonly identified concerns. In general, 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 of the Minesweeper files, 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 there was only a 35.9% overlap. Figure 4 summarizes the correlation between the number of files in which a concern is present and percent overlap.

As with abstraction, there seems to be a weak correlation between spread and percent overlap. The percent over-

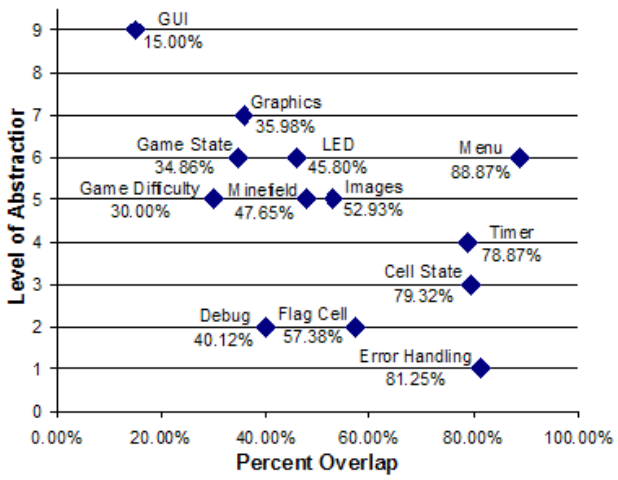

Figure 3. Concern overlap vs. abstraction level for concerns identified by both investigators.

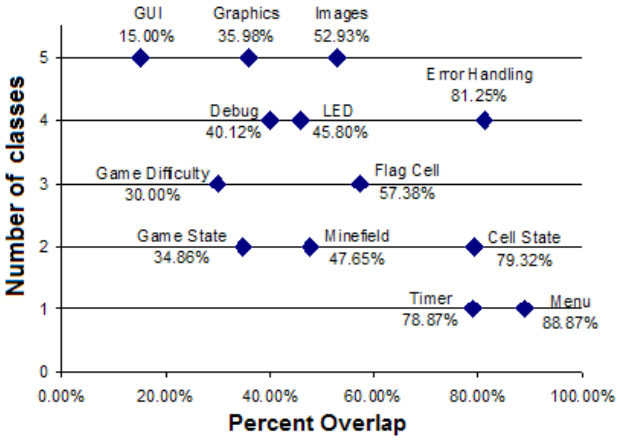

Figure 4. Concern overlap vs. number of classes for concerns found by both investigators.

lap of the *Game State* and *Error Handling* concerns, in particular, is not inversely proportional to the spread.

4. Generalizing the Case Studies

In this section, we discuss the insights gained from the case studies mentioned above. We discuss factors we believe lead to agreement, or even disagreement, between programmers when identifying concerns and associated source code. We then present the types of concerns we encountered in our case studies and the guidelines we developed.

4.1. Factors in Agreement Among Programmers

From our two case studies, we observed there are several factors that possibly contribute to agreement 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 a program is doing and how it does it is the most important factor that influences concern identification. In the `sort` case study for instance, Investigator M lacks many of Carver and Griswold's concerns primarily because she had a hard time comprehending how `sort` works in detail. As a result, she missed some of the more fine-grained concerns of Carver and Griswold.
- **Knowledge of the programming language.** Related to understanding the program, another factor is knowledge of the language in which the program is written. For example 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 on different levels of abstraction means it is not initially evident that two programmers have found the same concern. By using concern abstraction, we can discover the cases where we can map several concerns from either programmer to a single concern.
- **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, both Investigator M and Investigator T had a *Minefield* concern in the Minesweeper study. However, there was a low percentage (47.65%) of overlap 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 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. In the Minesweeper study, both investigators had a *Flag Cell* concern, but they had conflicting views on how to tag the following code fragment: `if(currentCell.getState()`

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. Concern Identification Guidelines

`!= 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.

4.2. Types of Concerns

Using what we learned from the two case studies, we developed a taxonomy of concern types in hopes of creating more consistent concern identification among programmers. We interpret a concern to belong to one or more of the following categories.

- **Feature.** Something a user of the program would be aware of.
- **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.
- **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.
- **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.
- **Language Characteristic.** Elements of a programming language such as constants, accessors, imported/included classes or interfaces, and comments.

Concern Tagging Guidelines

7. Different levels of concerns can be tagged in the same code fragment.
8. Tangled concerns—even though a code fragment is tagged with one concern, it can be tagged with another concern.
9. Use the search feature to find things, but take the time to figure out the context of the code before tagging.
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 line 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 6. Concern Tagging Guidelines

With a better idea of the types of concerns that exist in source code, programmers should be able to more easily identify them.

4.3. Guidelines

Next, we developed a set of guidelines that expound upon how to identify concerns and their associated code. Here we give some insight into how the individual guidelines, which are presented in Figures 5 and 6, were developed. The first six guidelines address identifying concerns in a program. 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 supported the creation of Guideline 1. Guideline 2 was developed based on the common concerns of Investigators M and T, *GUI*, *Game Difficulty*, *Minefield*, *Cell state*, *Timer*, *LED*, *Menu*, and *Game state*. All are feature-related concerns and subsume most of the other concerns in the

hierarchy. Many of the differences between taggings, especially the higher level concerns, were due to inconsistent identification. For example, take the *GUI* concern. In some cases, *GUI* was used to tag an area of code, and in a subsequent area of similar code a concern *GUI* subsumes in the hierarchy was used. This led to the creation of Guideline 5. We added Guideline 6 as a reminder to look for concerns that are not specifically tied to the functionality of the program.

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 because in many cases an area of code would be tagged with a similar or equivalent concern by both investigators but the way in which the characters of the code were tagged differed. Guideline 9 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. To promote consistency in tagging, and thus more potential overlap in concern identification, Guidelines 10-16 were developed.

5. Related Work

Robillard and Murphy [13] 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. Their 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 and gives us the accuracy needed for our study. Second, their algorithm can only infer concern code from an investigation transcript, which can lead to false positives. 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 [14] 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 [12] 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.

Information transparency [3] 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. Aspect mining [16] 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 [4]. 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 [18] 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 is 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.

Lai and Murphy did an exploratory study to investigate how different concerns interact [7]. 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 farther 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 provide that needed consistency.

6. Discussion and Future Work

In this paper, 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 can ease the difficulty of identifying concerns and improve their consistency. 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, or perhaps code that is undergoing development, 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.

The size of the case study systems is an obvious threat to the validity of our study. One area of future work is to repeat the experiments using larger systems, assuming that one has the resources to exhaustively identify the code associated with the concerns of the system.

Acknowledgments

We would like to thank Lee Carver and Bill Griswold for making their GNU `sort` information available. We thank Justin Manweiler for implementing the new features in Spotlight we needed. We also thank the anonymous reviewers and Elisa Baniassad for their helpful comments.

References

[1] Lee Carver and William G. Griswold. Sorting out concerns. In *OOPSLA '99 Workshop on Multi-Dimensional Separation of Concerns*, November 1999.

[2] Eclipse.org. The Eclipse homepage. URL: `http://www.eclipse.org/`.

[3] William G. Griswold. Coping with software change using information transparency. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[4] 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.

[5] 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.

[6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97: Proceedings of the European Conference on Object-Oriented Programming*, pages 220–42. Springer-Verlag, 9–13 June 1997.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Martin P. Robillard. *Representing Concerns in Source Code*. PhD thesis, University of British Columbia, November 2003.

[12] 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.

[13] 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.

[14] 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.

[15] Jr. Stanley M. Sutton and Isabelle Rouvellou. Modeling of software concerns in cosmos. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 127–133. ACM Press, 2002.

[16] Tom Tourwe and Kim Mens. Mining aspectual views using formal concept analysis. In *Proceedings. Source Code Analysis and Manipulation Workshop*, September 2004.

[17] 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.

[18] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–7, 1984.