A position paper for the ICSE 2001 Workshop on Advanced Separation of Concerns in Software Engineering

Analyzing Concerns Using Class Member Dependencies

Martin P. Robillard and Gail C. Murphy
Department of Computer Science
University of British Columbia
{mrobilla,murphy}@cs.ubc.ca

1 Introduction

Basic programming language elements, such as classes and methods, cannot always faithfully modularize all the concerns programmers might wish to express in a program [6]. Often, some concerns end up scattered amongst the various classes, and tangled within a particular class.

Various mechanisms have been proposed to help developers modularize concerns that cannot readily be encapsulated within classes (e.g., AspectJ™ [1], and Hyper/J™ [2]). These technologies typically allow concern modules to be integrated with a system at the declaration or use sites of class members. For instance, aspects in AspectJ can specify the introduction of a new field to a class (declaration site), or the addition of behavior before or after the body of a method gets executed (use site).

Gaining an understanding of how a particular concern is integrated with the base code and other concerns in terms of class members use can thus help in the application of these technologies [4]. We have started to investigate what benefits can be obtained from the analysis of dependencies between class members, in terms of concern specification and understanding.

Our first step was to establish a description of concerns, based on class members, that would provide a basis for reasoning about concerns. An example of a task requiring reasoning about concerns is deciding how to refactor code to achieve better separation of a concern, either using advanced composition technology, or basic object-orientation [4, 5].

In this paper, we propose a simple notation for describing concerns in object-oriented programs. This notation is based on the declaration and the use of class members. We then show how this description of concerns can provide data for three tasks pertaining to concerns: concern mining, concern understanding, and concern overlap analysis.

2 Describing Concerns

This work targets concerns that are expressible as features. Our goal was to devise a simple notation that could capture, at a high-level, how those features are implemented, and how they relate to the base code. The notation presented here uses class members and their uses as the links between a concern and the base code.

We discern between three levels of concern elements:

1. Use of classes.
2. Use of class members (i.e., fields, methods).
3. Class member behavior elements (i.e., use of fields and classes within method bodies).

The description of a concern consists of an enumeration of those elements, placed in the appropriate declarative context (e.g., fields and methods are declared within their corresponding class). Typically, an element is part of a concern if we can answer yes to the question: *is this element necessary for the concern implementation to be complete and correct?*

Use of classes is expressed by the `class-use` production rules in Figure 1. The rules specify that a concern either uses all of a class to implement its behavior, or only part of a class. In the latter cases, it is possible to express what parts are involved in the concern. The use of class members is expressed using the `member-use` production rules. The rules specify that a concern that uses part of a class can use any number of fields and methods. If the concern only uses part of a method to implement its behavior, it is possible to express what parts of the method are involved in the concerns. The specification of method behavior follows the `method-behavior-element` production rules. Essentially, at this point, we can specify that a field access, a method call, or an object creation are behavior related to a concern.

```
<class-use> ::= all-of  class <class-name>;
            ::= part-of class <class-name> { <member-use>* }

<member-use> ::= all-of field <field-name>;
             ::= all-of method <method-signature>;
             ::= part-of method <method-signature>
                                  { <method-behavior-element>* }

<method-behavior-element> ::= [reads | writes] field <field-name>;
                          ::= calls method <method-signature>;
                          ::= creates [object | array] <class name>;
```

Figure 1: Syntax for concern description.

For example, Figure 2 illustrates the specification of the ListCommands concern in an FTP server [3]. The figure specifies that:

- all of class `WildcardFilter` is used in implementing the ListCommands concern.
- part of class `FTPConnection` is used in implementing the concern. More precisely, all of the `parseDir`, `makefilePath`, and `doListCommand` methods, and part of the `doCommand`. The notation also supports a description of which part of the `doCommand` method is involved in implementing ListCommands, in this case, a call to the `doListCommand` method.

```
all-of class jftpd.WildcardFilter;

part-of class jftpd.FTPConnection
{
   all-of method parseDir(java.lang.String);
   all-of method makeFilePath(java.util.Vector);
   all-of method doListCommand(java.lang.String);

   part-of method doCommand(java.lang.String)
   {
      calls method jftpd.FTPConnection.doListCommand(java.lang.String);
   }
}
```

Figure 2: Example of concern specification.

3 Analyzing Concerns

We now turn to evaluating how the concern specification notation described in the previous section can be used to elicit information that can help reasoning about concerns. To that effect, we have investigated three specific tasks:

- **Concern mining.** The discovery of the set of elements potentially implementing a concern, based on a subset of those elements (which we call a *seed*).
- **Concern understanding.** The viewing of the elements used in the implementation of a concern, to understand and help validate a concern description.
- **Concern overlap analysis.** The determination of which part of a concern potentially interacts with parts of other concerns.

To carry out these tasks, we have developed a series of tools that operate on concern description files:

- **Concern Detector.** Reads in a concern description file, analyzes the bytecode of all the classes of a set of packages determined by the user, and produces a concern description file containing the uses of all the elements in the original file that are detectable statically without type inference.
- **Concern Viewer.** Reads in a concern description file and presents it graphically, as a hierarchical structure. The user can then click on any concern element to perform global dependency analysis on this element (Figure 3).
- **Concern Intersection.** Reads in two concern description files and produces the intersection of the two files, in terms of common elements.

As a preliminary experiment, these tasks have been performed on the jFTPd codebase identified in [3].

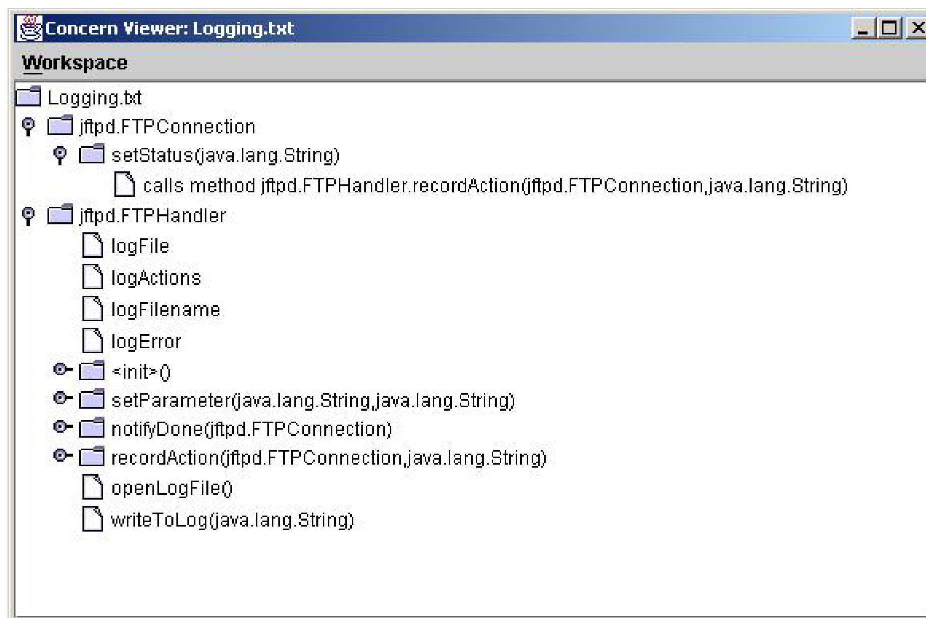

Figure 3: The Concern Viewer tool.

Concern Mining

Concern mining is a task performed to rapidly discover what elements might be involved in implementing a concern, and how these elements link with the base code. It consists in briefly browsing the code to determine seeds—program elements that are used in partially implementing a concern. These seeds are then used as input to the Concern Detector tool, which computes all their detectable[1] uses. Users of the tool can then determine if the elements using the seeds are also part of the concern.

For example, to elicit information about how the graphical user interface (GUI) is implemented in jFTPd , we looked at the classes of jFTPd and saw that two of them (`FTPAboutBox` and `FTPStatusWindow`) extended the windowing class `java.awt.Frame`. Running Concern Detector on these two class names allowed us to quickly discover that the uses of the two seed classes were contained by the `FTPHandler` class. In particular:

- The only detectable use of class `FTPAboutBox` is its instantiation in the constructor of `FTPStatusWindow`.
- The class `FTPStatusWindow` is only used by the class `FTPHandler`: it is instantiated in method `setupDone`, passed as argument to method `setFrame` where it is stored in field `win`, and accessed by the methods `notifyDone` and `recordAction`.

Concern Understanding

Concern understanding is a task performed to understand how a concern is implemented based on existing concern description information, such as lexical concern identification (e.g., highlighted program text). As opposed to concern mining, which involves discovering how a concern is implemented based on a few elements, the goal of concern understanding is to help validate a more complete view of a concern.

For this task, we have converted a slightly modified version of the concerns identified by marker 2 in previous work on feature selection [3] into our notation. Code that could not be expressed in terms of class member use (e.g., arithmetic operations) was simply indicated as a comment in the corresponding method description in the concern description file. Viewing the concerns in Concern Viewer, and performing dependency analysis with Concern Detector on various selected elements allowed us to elicit useful information about how the concerns were implemented. For example:

- Four fields of class `FTPConnection`, although not lexically marked as part of the ClientInteraction concern, were only used within blocks of code lexically marked as part of the ClientInteraction concern. We can thus include them as part of the ClientInteraction concern.
- The class `WildcardFilter`, although entirely marked as part of the ClientInteraction concern, is not used by any of the methods marked as part of the ClientInteraction concern. This suggests that the `WildcardFilter` class is not used in implementing the ClientInteraction concern, and should probably be removed from its description.
- Some blocks of code from the ClientInteraction concern included closing the FTP connection with the user (i.e., contained a call to `java.net.Socket.close()`), whereas some other ones did not. This pointed to an inconsistency in determining whether closing an FTP connection was or was not part of the ClientInteraction concern.

Concern Overlap Analysis

Concern overlap analysis is the task of determining how two or more concerns overlap. In our experiment, we have run the Concern Intersection tool on all the possible pairwise combinations of the 11 concern descriptions we had for the jFTPd code base. Of the $\binom{11}{2} = 55$ overlap possibilities, we detected 18 cases of potential concern overlap. For example, we automatically detected that the ListCommands concern had potential overlap with the ClientInteraction, DirectoryCommands, and PlatformSpecific concerns, whereas the ConnectionCommands concern only had potential overlap with the ClientInteraction concern.

[1] Statically detectable without type inference. This analysis is not fully conservative because it does not include uses of elements through polymorphism or reflection.

Furthermore, our overlap analysis allows to view the class members that overlap. For example, the ConnectionCommands and ClientInteraction concerns both use a call to method `doPortCommand` in the code of method `doCommand` of class `FTPConnection` (Figure 4).

```
part-of class jftpd.FTPConnection
{
   part-of method doCommand(java.lang.String)
   {
      calls method jftpd.FTPConnection.doPortCommand(java.lang.String);
      calls method jftpd.FTPConnection.doPasvCommand(java.lang.String);
   }
   part-of method doPortCommand(java.lang.String)
   {
      ...
   }
   part-of method doPasvCommand(java.lang.String)
   {
      ...
   }
}
```

Figure 4: Overlap between the ClientInteraction and ConnectionCommands concerns

Conclusions and Future Work

Analyzing concerns in terms of the class members that serve to implement them supports simple and fast analyses. However, this approach also has unequivocal limits. First, program code can only be specified as part of a concern if it can be expressed in terms of a dependency to a class member. Second, dependencies are not always very explicit about how code implements a concern: complex issues of protocol can arise. To make the best use of the concern notation presented in this paper, we found it necessary to view concern notation files in parallel with the corresponding code. Finally, there are false positives, and false negatives, which the user must cope with. False positives arise when, for example, uses of a particular class member are not limited to the implementation of a concern. False negatives arise when a dependency is not detected because a class member is accessed through polymorphism or reflection.

The question stemming from this preliminary experiment is whether we can operate within these limits to gather useful information about how concerns are implemented. A first step towards answering this question will be to enhance our lexical concern description tool, Feature Selector [3], to be able to automatically generate the type of descriptions presented in this paper, and to develop filtering mechanisms to deal with false positives. Then it will be possible to realistically tackle larger code bases implementing more sophisticated concerns.

Acknowledgments

Thanks to Albert Lai for his help with the Feature Selector tool, and for his insightful comments on this paper.

References

[1] AspectJ. `http://www.aspectj.org`.

[2] Hyper/J: Multi-dimensional separation of concerns for Java.
`http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm`.

[3] Albert Lai and Gail C. Murphy. The structure of features in Java code: An exploratory investigation. Position paper for the OOPLSA'99 Workshop on Multi-dimensional Separation of Concerns in Object-oriented Systems, November 1999.

[4] 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*. ACM, May 2001. To appear.

[5] Martin P. Robillard and Gail C. Murphy. An exploration of a lightweight means of concern separation. Position paper for the ECOOP 2000 Workshop on Aspects and Dimensions of Concerns, June 2000.

[6] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–109. ACM, May 1999.