The Structure of Features in Java Code: An Exploratory Investigation

Albert Lai and Gail C. Murphy
Dept. of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC V6T 1Z4 Canada
{alai,murphy}@cs.ubc.ca

September 15, 1999

A Position Paper Submitted to the OOPSLA '99 Workshop on Multi-dimensional Separation of Concerns

1 INTRODUCTION

Techniques to help software developers explicitly separate different concerns in their programs have been gaining increasing attention in the last few years. Three examples of such techniques are composition filters [ABV92], aspect-oriented programming [KLM$^+$97], and hyperspaces [TOHSJ99]. A central idea behind these techniques is that software systems would be easier to change if various design and programming decisions were modularized separately and simultaneously.

But what is a concern? And how do different concerns interact within a code base?

To gain some insight into these questions, we have conducted an exploratory investigation of concerns in two existing Java [GJS96] packages: gnu.regexp,[1] and jFTPd.[2] Each of the packages was marked for concerns by each author of this position paper. We then compared the concerns identified and analyzed how the concerns interacted with each other and across the existing structure of the Java package.

The results of this investigation provide:

- examples of concerns (in our terminology, features) that a software developer might want to explicitly separate,
- examples of the interactions that occur between features, and
- some insights into support that might help to identify and consolidate features.

2 FINDING FEATURES

2.1 FEATURE SELECTION TOOL

To make it easier to conduct this study, we developed a tool—`Feature Selector`—to help mark and analyze features in Java code. The `Feature Selector` tool parses a defined set of Java files and allows a user to highlight and tag segments of code as belonging to one or more user-defined features. The tool does not provide any automation mechanisms for marking code; all code is marked manually by the user.

Figure 2.1 illustrates the use of the tool. A user has defined several features (i.e., an error handling feature, a debugging feature, etc.), and has assigned each feature a different highlighting color. The files available for browsing and the features defined to date are visible in the left frame of the tool's interface. The user has determined that

[1] Written by Wes Biggs (Version 1.0.8)

[2] Written by Brian Nenninger (Version 1.3)

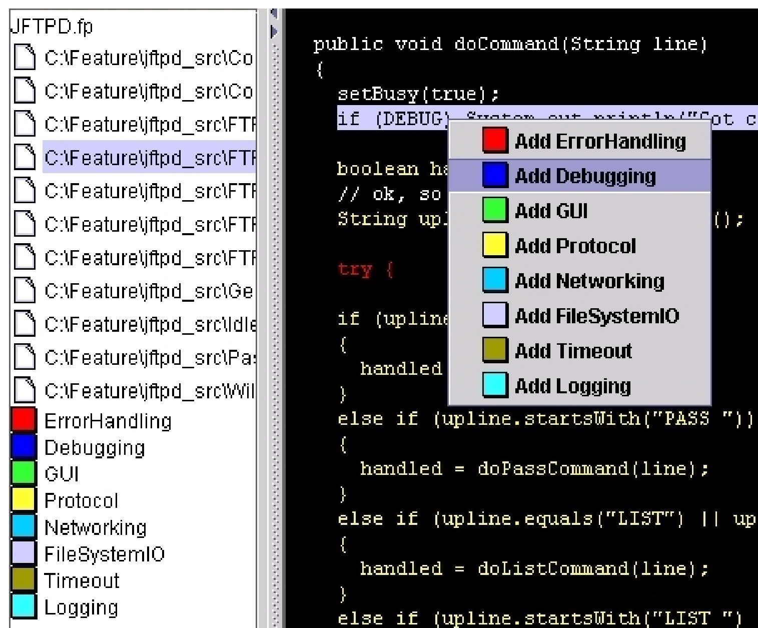

Figure 1: Feature Selection Tool

a segment of code visible in the main frame is related to debugging and has brought up a menu of the features currently defined. The user may select the debugging feature and the code will be highlighted to indicate the tagging. A segment of code may be highlighted as being part of more than one feature. The user also has access to a query facility that will summarize all code belonging to a particular feature or to a combination of features.

Since the `Feature Selector` tool parses the Java code into an abstract syntax tree representation, we are able to analyze the relationship of a set of features to the existing code structure.

2.2 FEATURE SELECTION PROCESS

Before marking any code, we reviewed both packages to gain an understanding of how the code works. This task was necessary to support feature selection.

Feature selection was based on various criteria. Some features were selected based on standards conformation. Examples of features selected using this criterion include the FTP protocol in jFTPd, and various regular expression syntaxes supported by gnu.regexp (e.g. Perl [Wal90], grep, and awk [AKW79]).

Other features encapsulated a configuration of the software package. As an example, the gnu.regexp package supports various forms of input, including among others, character arrays and strings. Each of these different forms of input was designated as a feature as it is conceivable that only a subset of the supported input data types may be necessary for any given program.

Another criterion used was portions of the code that a developer might want to change or remove. For instance, one feature selected in gnu.regexp captured code related to the matching of a regular expression over multiple lines in a given input.

Table 2.2 summarizes the wide range of features selected in each package by each marker.

Once a decision was made to identify a feature, we both found it difficult to mark the code related to that feature. There were two reasons why marking was a hard task. First, it was difficult to ensure feature self-consistency: that is, when marking, it was hard to ensure that an instance of marked code followed the same “theme” as other

marked code belonging to the same feature. For example, in some cases, it was possible to identify a central variable involved in providing a feature: following all uses of that variable was difficult without specific tool support. Second, it was difficult to ensure feature completeness: that is, it was difficult to ensure that all appropriate code was captured by a given feature. Multiple passes through the code were used to try to address these two issues.

3 FEATURES AND STRUCTURE

After marking the packages, we tried applying some simple metrics to determine whether we could characterize the features, and whether the metric values would point us towards interesting features to analyze further. We applied three metrics to each of the features: spread, tangle and density.[3]

$$spread(f) = \frac{\text{\# of files containing feature f}}{\text{total \# of files}}$$

$$tangle(f) = \frac{\text{\# of tokens marked with feature f and another}}{\text{\# of tokens marked with feature f}}$$

$$density(f) = \frac{\text{\# of tokens marked with feature f}}{\text{\# of tokens in files marked with feature f}}$$

3.1 Spread

Features with low spread intersect fewer files: we assumed that these features were already likely well modularized. As a result, we focused on features with high spread values.

One feature in both packages with a high spread was debugging. Debugging features consistently had a spread of over 50%. Since, in most cases, we believe that the debugging features can already be fairly well modularized using mechanisms such as AspectJ, we did not analyze these features further.

[3] The tangle and density metric relies on *tokens*. A token is defined by the Java grammar provided by JavaCC (http://www.metamata.com). For example, each keyword, variable, and comment is a token.

A feature with a high spread value (29%) that we did analyze was the input types feature for gnu.regexp. Further investigation quickly revealed that the high spread was due to the representation of each input type by a class. These classes made up a significant fraction of the total classes in the package.

The input error handling feature in gnu.regexp also has a high spread value (43%). This feature centered around the definition and use of a static variable which represented an error code. Typically, this error code is compared to the return value of a method. Thus, uses of this error code tend to occur right after calls to a particular method.

The timeout feature identified in jFTPd also had high spread (36%). Analysis revealed that this feature was poorly modularized. Knowledge of the timeout was spread through several classes when it could have been better encapsulated with the help of the Observer pattern [GHJV94].

3.2 Tangle

Since we were interested in feature interactions, we focused our attention on features with high tangle values. The features with the highest values were those that were encapsulated by other features. For example, one feature identified in jFTPd was GUI. This GUI feature was a subset of the User Interface feature. The non-GUI code in the User Interface feature managed the console. Other features shared similar relationships and thus both markers noticed a need for the concept of a sub-feature: a feature that is a subset of another feature.

3.3 Density

High values (over 90%) for density indicated that the corresponding feature was well modularized. Low values were harder to interpret. Low values could be caused by at least two different, contradicting situations. The first situation is where several files each have a small number of tokens marked with a feature. An example of this is the logging feature in jFTPd with a density value of 4%. The second situation is where a small number of tokens in a single large file are marked. The version information feature from gnu.regexp provides a good example. It had a density of 0.4%.

Table 1: Features Selected in gnu.regexp and jFTPd

No.	Package	Marker	Feature	Description
1	gnu.regexp	#1	Version Information	Software version tags in code
2	gnu.regexp	#1	Input Data Types	Various forms of input, e.g., strings
3	gnu.regexp	#1	Error Handling	
4	gnu.regexp	#1	Debugging	
5	gnu.regexp	#1	String Substitution	Replacing strings within matches
6	gnu.regexp	#1	* various syntax flags *	Features were selected for each syntax supported
7	gnu.regexp	#1	REFilterInputStream	For a given input stream, replace all regexp with a specified string
8	gnu.regexp	#2	Input Error Handling	Handling of errors in input to match against
9	gnu.regexp	#2	Pattern Error Handling	Handling of errors in regexp pattern
10	gnu.regexp	#2	Multiline Match Support	Code supporting matches across lines
11	gnu.regexp	#2	Newline Handling	Code dealing with newlines
12	gnu.regexp	#2	Variable Substitution	Code supporting variable substitution during matching
13	gnu.regexp	#2	Matching Rules	Code controlling matching process
14	gnu.regexp	#2	RE Pattern Syntax	Code related to multiple regexp syntaxes
15	jFTPd	#1	Error Handling	
16	jFTPd	#1	Debugging	
17	jFTPd	#1	GUI	
18	jFTPd	#1	Protocol	FTP RFC commands and completion codes
19	jFTPd	#1	Networking	Underlying network connection code
20	jFTPd	#1	File System IO	Code dealing with the server filesystem
21	jFTPd	#1	Timeout	Code related to command timeouts
22	jFTPd	#1	Logging	Code related to logging server commands
23	jFTPd	#2	User Interface	
24	jFTPd	#2	GUI	
25	jFTPd	#2	Debugging	
26	jFTPd	#2	Logging	Code related to logging server commands
27	jFTPd	#2	Platform Specific	Code dealing with specific platforms
28	jFTPd	#2	Windows Specific	Code dealing with the Wintel platform
29	jFTPd	#2	Client Feedback	Responses to client program
30	jFTPd	#2	Client Interaction	Commands from client
31	jFTPd	#2	Directory Commands	ftp commands related to directories
32	jFTPd	#2	List Commands	ftp commands related to listing files
33	jFTPd	#2	Server File Manipulation	ftp commands configuring server
34	jFTPd	#2	Connection Commands	ftp commands connecting to server

3.4 Summary

Overall, the values we computed for these metrics did not show any obvious trends. One reason why the metrics were not very helpful might be that they were computed at too fine of a granularity (i.e., token-level rather than method-level). In general, given current technology to support separation, some means of determining high-spread, low-tangle features might be helpful in assessing whether a feature is worth separating.

4 DISCUSSION

Our analysis reveals that only 20 to 25% of the tokens in any given package were selected as part of any feature. This begs the question of what the remaining code represented. For the most part, the approximately 80% of the code left unmarked was code that was considered essential to the package. Specifically, if the code was not present, the package could not provide the most basic functionality you would expect from a regular expression package or an ftp daemon.

Though there were some similarities between the features we selected, the majority of them differed. For jFTPd, common features included debugging, logging and GUI. For the other features, one of us chose mostly behavioural features: protocol, networking, fileSystemIO, and timeout. The other chose a mixture of behavioural (i.e., directory commands, list commands, server file manipulation, and connection commands), interactional (i.e., user interface, client feedback, and client interaction), and code-based (i.e., platform specific and windows specific) features. For gnu.regexp, error handling features and features based on regular expression syntax were selected by both markers. For the remainder of the features, one marker selected more features based on configuration (input data types, string substitution, and regular expression filter input streams), while the other marker selected more on behavioural grounds (i.e., newline handling, matching rules, etc.).

The selection and analysis of these features clarifies and identifies a number of issues surrounding separable features in a source code base.

Sub-method Join Points Not surprisingly, when analyzing existing code for features, a need for sub-method join points arises.[4] Some of these sub-method join points might be identifiable based on patterns in the code. For example, when discussing the spread metric earlier, we described that the input error handling feature tended to occur right after calls to a particular method. The join points needed to separate (or combine) the input error handling feature could be relatively easily named by specifying this pattern.

Other sub-method join points are more difficult to name. One particular case that arose during our analysis related to the features encapsulating the various regular expression syntaxes supported in the gnu.regexp package. The syntax flags used to affect how regular expressions are compiled are used in a single "while" loop in the regular expression class constructor. This loop seems to have been written this way to deliberately trade modularity for performance. Ideally, the syntaxes supported by a regular expression package should be configurable without a loss in performance. However, the join points for syntax flags lie in the conditional parts of "if" statements. Succinctly stating which "if" construct should serve as a join point for a particular syntax flag appears difficult.

Required Features As previously mentioned, gnu.regexp supports various input data types. Support for each of these data types should be a user configurable option. Thus each of the data types could be considered a feature (or sub-feature). However, at least one of these features must be present for the package to function correctly. Any tool that is built to help a developer separate and compose features needs to track which features are required.

Feature Interaction Ideally, features should be encapsulated into reusable modules. However, various feature interactions may hinder proper separation. Two such interactions are overlap and order.

Overlap refers to one or more statements in the code being associated with more than one feature. (Our tangle metric measured overlap.) In the case of sub-features (discussed in Section 3.2), overlap is expected and should

[4] A join point refers to a point in the code where a feature interacts with the base code or with another feature.

be manageable in a straightforward way. Other forms of overlap indicate a more subtle form of feature interaction. For example, in jFTPd, overlap occurred between a portion of code associated with an error handling feature and code associated with the dispatching of commands. In this case, the error handling code that overlapped with the command dispatching formed a required part of the command dispatching feature. Recognizing and describing such interactions may become an important piece of managing feature interactions.

Order refers to whether or not there is a control- or data-flow relationship between code in the same method related to two different features. In the code we marked, two or more features are present in same method in approximately half of all code marked by any feature. In the cases where two or more features are present, the number of features present usually does not exceed three. However, both gnu.regexp and jFTPd have substantial blocks of code where more than seven features are present. In gnu.regexp, the regular expression class constructor interacts with approximately fifteen features. In jFTPd, a method for handling command lines interacts with eight features.

In most cases that we analyzed, when two or more features are present in the same method, the features are not partially ordered. This is good news for those building separation of concerns mechanisms because it may simplify the combination of separated features. In those cases where two features were partially ordered, very few were ordered by data flow; the majority were ordered by control flow. For example, the error handling features in both gnu.regexp and jFTPd tended to be partially ordered by control flow. This is not such good news since it may be more difficult to describe control-relationships between features than data-relationships.

5 SUMMARY

Much recent work has focused on mechanisms for explicitly separating concerns in code. However, we still lack a generally-accepted, precise definition of the concept of a concern. To investigate the range of meaning of concerns in source code, we undertook a small study focused on identifying concerns in existing Java packages. The results of this study suggest that concerns can encapsulate a variety of concepts such as standards, behaviours, user interactions and configurations. An initial analysis into the structure of the concerns identified suggests some issues that may arise as separation of concerns mechanisms evolve, namely a need for sub-concerns and a need for specifying required concerns. Our analysis also confirms that the task of modularizing and recombining features is non-trivial given the interactions that exist between features.

References

[ABV92] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *Proceedings of ECOOP '92*, pages 372–395, 1992.

[AKW79] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. Awk – a pattern scanning and processing language. *Software—Practice and Experience*, 9(4):267–280, 1979.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, June 1997.

[TOHSJ99] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton Jr. n degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of ICSE 21*, pages 107–119. IEEE Computer Society, 1999.

[Wal90] L. Wall. *Programming Perl*. O'Reilly & Associates, 1990.