An Exploration of a Lightweight Means of Concern Separation

Martin P. Robillard and Gail C. Murphy
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC V6T 1Z4 Canada
{mrobilla,murphy}@cs.ubc.ca

March 28, 2000

A position paper submitted to the ECOOP'2000 workshop on aspects and dimensions of concerns.

1 Introduction

Different concerns which arise when developing a system, such as different features, often end up scattered and tangled in code bases. The scattering and tangling of concerns complicates the performance of many software engineering tasks. For example, the scattering of a feature across a set of source files complicates the task of making any change to that feature. To help reduce the scattering and tangling problems, several separation of concerns mechanisms have been proposed (e.g., Composition Filters [1], Aspect-oriented Programming[4], Hyperspaces [7]).

Separation of concerns mechanisms allow a developer to describe how a concern integrates (composes) with other concerns and other code. Different mechanisms provide different means of describing the composition, including different means of describing the *join points* of the composition—the points in the code where a concern is to be integrated. Since existing separation of concerns mechanisms are limited in the kinds of join points which can be specified (i.e., submethod join points are not possible), applying a separation of concerns mechanism to an existing system typically requires some restructuring of the existing code base.

In earlier work, we reported on the trade-offs of various restructurings of the `jFTPd` code base when separating previously identified concerns [5] with the Hyper/J™ tool developed at IBM T.J. Watson Research Center [6].[1] The `jFTPd` code base implements an FTP server in Java™ [2].

When trying to decide what to restructure, one can be overly influenced by the nature of the separation of concerns mechanism at hand. To consider in more depth the restructurings we had used for `jFTPd`, we revisited the code base, this time considering how concerns might be factored using only a lightweight approach. Our approach was to refactor the `jFTPd` code to explicitly represent the concerns directly in the object-oriented structure, to perform that refactoring respecting a set of design rules, and to use simple lexical tools, such as `grep`, to allow a concern to be factored out of the code base (Section 3 and 4). We are not implying our lightweight approach should replace or compete with separation of concerns mechanisms; in particular, our approach supports decomposition, but not composition, of concerns. However, comparing the costs and benefits of such a lightweight approach with separation of concerns mechanisms might help determine conditions

[1] `jFTPd` (Version 1.3) was written by Brian Nenninger.

under which separation of concerns mechanisms are warranted. In addition, the restructurings used in our lightweight approach are different than those used when we applied Hyper/J: each restructuring also has different trade-offs and carries different consequences (Section 5).

2 Lightweight Concern Restructuring in jFTPd

Table 1 summarizes the concerns identified in a previous study of the `jFTPd` code base [5]. The key concerns we consider in this investigation are in bold text. The restructuring of the code we describe in this paper was performed by the first author, who was involved neither with the initial concern mark-up experiment, nor with the Hyper/J restructuring experiment [6].

Table 1: Concerns identified in the code of jFTPd

Concern Name
User Interface
GUI
Debugging
Logging
Platform Specific
Windows Specific
Client Feedback
Client Interaction
Directory Commands
List Commands
Server File Manipulation
Connection Commands

2.1 Debug

The code relating to the Debugging concern consisted of a constant boolean attribute (`DEBUG`) declared and initialized in most of the classes, and of code which prints information to the console screen if the value of the `DEBUG` attribute is set to `true`. Most of the code identified as part of the debug concern consisted of lines similar to the following.

```
if (DEBUG) System.out.println(...);
```

Our restructuring consisted of defining a `Debug` class with a boolean attribute and methods such as `entering(String)`, `exiting(String)`, and `message(String)`. In every class, the `DEBUG` attribute was replaced by the declaration and instantiation of a `Debug` object named `aDebug`, and each `if` structure to print debug information was replaced by a single method call.

2.2 Logging

The Logging concern comprised pieces of code to perform the logging of various FTP operations. The code comprising the Logging concern was scattered as follows.

- Four attributes of the class `FTPHandler` were only used for logging operations.
- Some methods declared in `FTPHandler` were fully marked as relating to the Logging concern. These methods were only called in blocks of code also marked as part of the Logging concern.
- Two `else-if` clauses, part of a bigger `else-if` structure, in a method of `FTPHandler`.
- Two blocks of code marked as part of the Logging concern in two different methods of `FTPHandler`. These block of code were identical.
- One line in a different class, `FTPConnection`, called some of the logging functionality on an object of class `FTPHandler`.

This combination of code hinted at the fact that the Logging concern should actually have been an object. It was straightforward to encapsulate this concern with an object. A `FTPLog` class was created, and the four logging attributes were moved to it. Methods fully marked as part of the Logging concern were also moved to the `FTPLog` class, and were specified as private methods. After these movements, we were left with the blocks of code and the branches of the `else-if`. Both cases were implemented as small methods. Because of the high redundancy (and even duplication) of the scattered logging code, this resulted in the creation of only two methods. The branches of the `else-if` were put in a method of the `FTPLog` class, which was called in the `else` branch of the `else-if` structure.

With this restructuring, all logging functionality in `FTPHandler` is performed through methods calls on the `FTPLog` object. The object is declared as an attribute of the `FTPHandler` class.

2.3 Client Feedback

The Client Feedback concern corresponds to the generation of textual messages to the clients of the FTP server, such as "500 No such directory".

The Client Feedback concern was scattered throughout the `FTPHandler` class. Basically, it consisted of an attribute—an output buffer—declared in `FTPHandler`, numerous one-line statements writing messages to this buffer, and a limited number of blocks of code performing slightly more sophisticated output operations involving lists. The scattering was not limited to `FTPHandler`: the class `FTPConnection` also contained a block of code marked as part of the Client Feedback concern. This block of code was independent from the buffer attribute.

Similarly to the cases of the Debug and Logging concerns, the Client Feedback concern was implemented in a new class, `FTPClientFeeback`, and all statements concerning client feedback operations were hidden in methods of this class. In `FTPHandler` and `FTPConnection`, all the parts of the code dealing with client feedback were reduced to a single method call.

2.4 GUI

The GUI concern related to the graphical display of windows indicating the status of the FTP server.

The case of the GUI concern was different than the concerns previously discussed. In the case of the GUI concern, the class `FTPHandler` declared two attributes marked as GUI, a boolean variable and a `FTPStatusWindow` object. The code in the `FTPStatusWindow` class was not fully marked as part of the GUI, but also contained some code that provided core functionality for the FTP server. This intermixing of core and GUI functionality made it impossible to encapsulate the GUI concern in a class as we did earlier. Numerous callback methods would have been necessary which would have made the code more complicated. For this reason, it was not possible to implement the GUI concern as an object.

3 Lightweight Strategy

The overall strategy we used was to encapsulate each concern in a class. Classes representing a concern were named to reflect the concern. As a first step towards this encapsulation, we moved all attributes marked as a particular concern into the concern class. In all the cases we analyzed, attributes that were marked as relating to a particular concern were only used in blocks of code that were marked as relating to the *same* concern. We specified these moved attributes to be private attributes of the concern class. The next step consisted of moving all the methods that were completely marked as part of a concern directly in the concern class. These methods were also scoped private.

Then, we needed to factor out concern-specific behavior that did not align with method boundaries. To achieve this, we took all blocks of code relating to a concern, and encapsulated the behavior in a method, using parameters to pass any information that was external to the concern. This step was not as difficult as expected because scattered blocks of code pertaining to the concerns analyzed tended to be cohesive. Often, the new method could be named using the comment above the block of code. Blocks of code were also duplicated, so encapsulating them in a method resulted in removing some code duplication.

The general result of this approach is that the data and behavior related to any particular concern becomes encapsulated in an class. Access to this concern in other classes consisted of calling a single method on an instance of the concern class.

For three out of the four concerns analyzed, it was possible to reduce the effects of scattering and tangling of concerns in `jFTPd` by judiciously using simple design rules.

Use a naming convention for concerns. Not only did we encode the name of the concern in the concern class, but we also named the objects representing the concerns judiciously, using the idea of information transparency [3]. We did this to make it possible to find join points. For example, we named our concerns using the name of their class prefixed by the letter 'a' (for attribute). The `FTPHandler` class thus contained the following declarations.

```
// Concerns
FTPClientFeedback aFTPClientFeedback;
Debug             aDebug;
FTPLog            aFTPLog;
```

Insert join points (method calls to concern objects) so that their removal does not break the syntax of the program. For example,

```
if (...)
   aClientFeedback.message();
```

should be

```
if (...)
{
   aClientFeedback.message();
}
```

so that removal of the line containing `aClientFeedback` does not cause a compile error. See Section 4 for further details on removing a concern.

Methods in concern objects should not have side effects on anything that is not part of the concern (i.e., the parameters to a method). This helps to ensure that the program will still behave correctly even if a concern is not included.

4 Benefits and Limitations

4.1 Benefits

Our lightweight approach allows us to achieve some the benefits of separation of concerns. By having all data and behavior relating to a concern encapsulated within a class, we reduce scattering. Changing the internals of the concerns can be achieved simply by looking at a particular class. Another benefit of the strategy is the reduction of code tangling. Since at each join point, the connection to the concern is expressed as a single method class, it is possible to remove all such points using a lexical tool such as `grep`.[2] With the join points of a concern removed, developers can look at a class without having to account for the interactions related to that concern. Better still, the class can also be compiled with some of the concerns removed, thus achieving an effective decomposition of concerns.

The main benefit of this lightweight approach is that the relative reduction in scattering and tangling described above can be achieved with simple tools. Furthermore, in the case of the concerns analyzed in `jFTPd`, the approach resulted in a simple design and implementation, perhaps one from which all separation of concerns mechanisms should be applied. (We return to this point in Section 5).

[2] The `grep` tool, used with the option `-v`, filters from the input all lines matching a specific pattern.

4.2 Limitations

Although the lightweight approach is simple and straightforward to apply, there are some limitations.

- The approach cannot deal with overlapping concerns. Even if some statements were marked as belonging to multiple concerns, they were ultimately integrated in a single concern class (the one that seemed the most relevant).
- While it is possible to decompose concerns, it is not possible to compose them. As a result, join points are inevitably scattered.
- Depending on how the code is organized, the concerns can end up with a set of small methods that do not make much sense without the context in which they are used.
- The restructuring can result in a loss of efficiency of the code. For example, if the branches of an `else-if` structure are factored in different concern objects, the resulting join points will have to be a series of `if` structures, which involves more test operations at run-time.

Perhaps the most stringent limitation is that some code structures simply do not allow for direct refactoring of concern code into methods. For example, any block of code containing a return statement cannot easily be encapsulated in a method. As another example, there can be no dependence from the concern code to the class using the concern. In our experience with the GUI concern, this situation resulted in very complex callback schemes that clearly degraded the overall structure of the code.

5 Other Restructuring Approaches

As outlined in Section 1, the restructuring described above is not our first encounter with `jFTPd`. In earlier work [6], we investigated three different restructurings of `jFTPd` when using Hyper/J to separate the previously identified concerns in the code base. Although space does not permit a full description of the restructurings nor a full analysis of the trade-offs between the different approaches, we briefly describe and compare the approaches.

To simplify the discussion, each restructuring approach is numbered. We will use `Restructuring #1` to refer to the restructuring used with our lightweight approach described above. In `Restructuring #2`, concerns found within a method on a class were restructured into multiple methods on the same class. In `Restructuring #3`, concerns were factored into separate class hierarchies, with common code necessary for a concern appearing in each relevant hierarchy.[3] Finally, `Restructuring #4` refers to a restructuring in which concerns were separated into classes, similar to `Restructuring #1`, in such a way that a `before` and `after` method approach could be used to recombine concerns. This constraint meant that sometimes call sites of restructured concerns had to be restructured to pass new parameters.

The most significant difference between `Restructuring #1` and the other three restructurings is that it is harder to state join points with `Restructuring #1`. This difficulty arises because the join points do not lie on method boundaries, rather calls to an encapsulated concern appear in the middle of methods.

The first, third and fourth restructurings are similar in that they encapsulate concerns. `Restructuring #2` separates concerns into methods but does not encapsulate those methods within a class. However, `Restructuring #2` is likely easier to automate for this very reason.

No restructuring clearly dominates any other for making concerns easy to manipulate at a reasonable cost for performing the restructuring. Moreover, each restructuring in some way limits the reexpression of concerns. For instance, the restructurings which

[3] In other words, each class hierarchy for each concern was declaratively complete.

separate the concerns into different classes may reduce the ability of a user to express a new concern which overlaps with an existing concern. Although `Restructuring #2` makes less of a commitment with respect to what a concern is and which code it involves, the naming of different methods as part of different concerns may still make manipulation of some new cross-cutting concern difficult.

6 Summary

Different levels of sophistication in separation of concerns mechanisms provide different levels of returns. We have described a lightweight approach to separating some concerns which can help decompose, but not compose, concerns in a system. Investigating this approach on the `jFTPd` code base led us to consider a restructuring of the code. We have also found restructuring necessary when applying other separation of concerns mechanisms, namely Hyper/J, to an existing code base. Each restructuring approach has different costs and benefits. A more thorough understanding of the trade-offs between different restructuring approaches is needed to understand how to best apply separation of concerns mechanisms to existing systems.

References

[1] 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.

[2] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley Longman, Inc., 1996.

[3] William G. Griswold. Coping with software change using information transparency. Technical Report CS98-585, Department of Computer Science and Engineering, University of California, San Diego, April 1998. Revised August 1998.

[4] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[5] Albert Lai and Gail C. Murphy. The structure of features in Java code: An exploratory investigation. Position paper for the OOPSLA'99 workshop on multi-dimensional separation of concerns in object-oriented systems. October 1999.

[6] Albert Lai, Gail C. Murphy, and Robert J. Walker. Separating concerns with Hyper/J™: An experience report. Position paper for the ICSE 2000 workshop on multi-dimensional separation of concerns.

[7] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, May 1999.