Separating Features in Source Code: An Exploratory Study

Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard
Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver BC Canada V6T 1Z4
1.604.822.5169
{murphy,alai,walker,mrobilla}@cs.ubc.ca

ABSTRACT
Most software system codebases are inflexible. Reconfiguring the software modules comprising a system to add or to delete a feature typically requires substantial effort. This lack of flexibility increases the costs of building variants of a system, amongst other problems.

New languages and tools that are being developed to provide additional support for separating concerns in a system show promise to help address this problem. However, applying these mechanisms requires determining how to enable a feature to be separated from the codebase. In this paper, we investigate this problem through an exploratory study conducted in the context of two existing systems: gnu.regexp and jFTPd. The study consisted of applying three different separation of concern mechanisms—Hyper/J™, AspectJ™, and a lightweight, lexically-based approach—to separate features in the two packages. In this paper, we report on the study, providing contributions in two areas. First, we characterize the effect different mechanisms have on the structure of the codebase. Second, we characterize the restructuring process required to perform the separations. These characterizations can help software engineering researchers elucidate the design space of using these mechanisms, tool developers design support to aid the separation process, and early adopters apply the techniques.

Keywords
design space, feature separation, aspect-oriented programming, hyperspaces

1 INTRODUCTION

Most software system codebases are inflexible. Reconfiguring the software modules in a codebase to add or to delete a feature typically requires substantial effort. Adding a new feature into a telecommunications system, for example, requires substantial effort simply in understanding how the feature interacts with potentially hundreds of other features [4]. This lack of flexibility increases the costs of building variants or versions of a system, and delays the time to market of products, amongst other problems.

New languages and tools that are being developed to provide additional support for separating concerns in a system show promise to help address this problem. These mechanisms provide a means to express which code relates to a concern and a means to compose (or decompose) desired concerns into (or out of) a system. Since these mechanisms are in their infancy, there is no definition of what constitutes a concern. Concerns may range from performance enhancements to features. In this paper, we focus on concerns comprised of code supporting user-relevant features. We use the terms concern and feature interchangeably.

Abstractly, separation of concern mechanisms sound ideal to use to improve the flexibility of software systems. Features can be encapsulated and then composed into a system as desired. Practically, there are many challenges involved in using these mechanisms [15, 2]. One challenge is determining what concerns are useful to encapsulate. Another challenge is understanding how to structure the base code to enable a concern to be separated.

In this paper, we focus on this second challenge, considering how various software structures support the separation of a concern. We investigate the structures possible by focusing on two primary ways in which concerns may interact in an object-oriented system: within methods, and between classes. For each of these interaction types, we apply three different separation of concern mechanisms: Hyper/J, a tool that supports the concept of hyperspaces [14, 11], AspectJ, a tool that supports the concept of aspect-oriented programming [7], and a lightweight, lexically-based approach [13] that considers what separation is possible without advanced tool support. The investigations were undertaken on two existing object-oriented systems: gnu.regexp and jFTPd.

We chose to investigate the impact of these mechanisms on structure in the context of existing systems because we believe that many concerns will not be determined a priori but instead will emerge as the codebase for a system matures. Studying existing systems allows us to consider how concerns are encoded, different means of restructuring the system to expose and extract a concern, and the process required to perform the separation.

No single structure in our study is obviously "right": each carries with it a different set of tradeoffs. Based on our expe-

Paper submitted to ICSE 2001. August 28, 2000.

riences we report on lessons we have learned about designing appropriate target structures for the base and separated code, for preparing a system to separate concerns, and for restructuring the code.

This paper makes two contributions:

- It provides an initial characterization of the process of restructuring a codebase to permit the separation of concerns.
- It provides an initial characterization of the effect of separating concerns on the structure of both the non-separated and the separated pieces of a codebase.

These characterizations can help software engineering researchers elucidate the structural design space for these new separation of concern mechanisms. They can also help early adopters apply the technologies by suggesting alternative ways of applying the mechanisms and by providing a means of selecting amongst alternatives. Tool developers may also benefit through guidance about the kind of design methods, design patterns, and tools needed to support the use of the mechanisms. Finally, the structural characterizations can help the developers of separation of concern mechanisms evolve their techniques.

Section 2 provides a brief overview of the code and concerns used in the study. Section 3 provides an overview of the three separation of concern mechanisms applied. Section 4 and Section 5 describe the study and provide a synthesis of the results. Section 6 discusses limitations of the study. Section 7 compares our study to others that have been conducted. Section 8 concludes the paper.

2 DETERMINING CONCERNS

The two systems we selected for our study were gnu.regexp and jFTPd.[1] These systems were chosen because they are of moderate size, they include a range of functionality, and their Java™ source is readily available.

To determine concerns in the two packages, two of the authors marked concerns in the source using the Feature Selection tool [8]. This tool parses a set of Java files and allows a user to highlight and tag segments of code as belonging to one or more user-defined concerns. All tagging is free-form: a user may choose any portion of the code to mark as belonging to a concern.

The markers selected concerns based on a variety of criteria. Some concerns encapsulated code implementing a standard, such as pieces of code supporting the FTP protocol in jFTPd. Other concerns encapsulated a configuration of the software package; for example, in gnu.regexp, different concerns were defined for different forms of input, such as character arrays and strings. Most concerns were selected based on the criterion that they represented portions of code—features—that a developer might want to change or remove. For instance, one concern selected in gnu.regexp captured code related to the matching of a regular expression over input spanning multiple lines. More detail about the concerns selected is available elsewhere [8].

[1] Version 1.0.8 of gnu.regexp was written by Wes Biggs and comprises approximately 25 classes and over 3500 lines of source. This package provides regular expression support in Java. Version 1.3 of jFTPd was written by Brian Nenninger and comprises approximately 11 classes and just over 2900 lines of source. This package implements an FTP server in Java.

3 SEPARATION OF CONCERN MECHANISMS

We chose three different methods for separating the concerns identified in gnu.regexp and jFTPd. Two of the methods, hyperspaces [14] and aspect-oriented programming [7], are supported by special purpose tools, Hyper/J and AspectJ respectively. The third method is a lightweight, lexical means of separating some concerns which we have been investigating [13]. We included our lightweight approach because it provides a means of comparing the separation possible within the object-oriented paradigm. Although the lightweight approach is not a mechanism per se, for simplicity, we refer to all three approaches as mechanisms.

In this section, we provide a brief overview of each mechanism, deferring examples of each mechanism until Section 4.

Lightweight Concern Separation

Our lightweight lexical approach to concern separation (LSOC) is based on three simple ideas:

- refactor the code to capture concerns as classes,
- use a naming convention to refer to the concern classes in the base code, and
- use a simple lexical tool, such as `grep`, to factor a concern out of a codebase based on the naming convention.

This approach does not include any composition mechanism: a concern may be removed from a system but not added. As a result, it is not as flexible as the other two mechanisms included in the study.

Hyper/J

The Hyper/J[2] tool permits a set of Java source to be decomposed along multiple dimensions simultaneously. Each dimension may be partitioned into a set of concerns. For example, a Feature dimension in jFTPd may be partitioned into several concerns: some methods or classes may be designated as contributing to a feature that logs FTP server information, while others contribute to a feature that allows users to issue commands to connect to the server.

Hyper/J allows different dimensions of concern to be integrated using declarative composition rules. For example, a developer for jFTPd might describe how to compose the features related to issuing connection and directory commands while leaving out the functionality to support the listing of remote directories. The resultant system represents one member of a product family.

When using Hyper/J, a developer provides three inputs:

- a *hyperspace* file that describes the Java class files that can be manipulated,
- a *concern mapping* file that describes which parts of the codebase map to each dimension of concern, and

[2] See `www.research.ibm.com/hyperspace` for more information.

- a *hypermodule* file that describes which hyperslices[3]—dimensions of concern—should be integrated and how that integration should proceed.

Given these inputs, Hyper/J produces a new set of class files which include the specified behaviour.

AspectJ
AspectJ is an extension to the Java programming language.[4] AspectJ provides support for encapsulating concerns that crosscut a system's structure, such as exception handling, synchronization policies, or features.

Developers encapsulate a concern using the `aspect` construct. This construct is similar to the Java class construct: it provides a means of grouping together code related to a concern. Within an aspect, developers may use `advice` and `introduce` constructs. The `advice` construct groups together code that is to be executed at some defined point in the system's execution. The defined point may be `before`, `after`, or `around` the execution of a method, amongst other possibilities. `Before` and `after` advice is straightforward. The code associated with an `around` advice is given control before execution of the associated method; the code may or may not cause the actual method to be executed. The `introduction` construct provides a means of adding new behaviour to the classes participating in a concern.

The AspectJ language is supported by a compiler that, given the Java source for a system and a set of aspects, applies the aspects to the system and produces a set of source and class files for the combined product. For this study, we used Version 0.7b3 of the AspectJ tool.

4 STUDY

Concerns are coded into systems in a variety of ways. Sometimes, concerns are encapsulated in methods or classes. Other times, single methods include code associated with multiple concerns. In yet other cases, code contributing to a concern may be split across multiple methods of multiple classes.

To provide an initial analysis, we chose to focus on two common situations: the tangling of multiple concerns in a single method, and the tangling of multiple concerns in a small number of classes. The particular cases we considered are by no means exhaustive. We have not considered an exhaustive set of mechanisms for separating concerns nor all the ways in which the mechanisms selected can be applied. Despite these limitations, we believe the results of our study provide a start for elaborating the process of separating concerns in an existing system and the structural design space available when performing such a separation.

For each of the two situations of concern encoding, we selected representative examples from the gnu.regexp and jFTPd systems. We then applied each of the three mechanisms to separate the concerns. In this section, we describe the results of applying the mechanisms to the examples and provide a comparison of the results based on the effect of the separation on the structure of both the base and the separated code. We defer a synthesis of the results until Section 5.

[3] A hyperslice is a declaratively complete slice of the program with respect to a particular concern.

[4] See `www.aspectj.org` for more information.

```
int[] match( ... ) {
// Multiline matching
if (newline && (mymatch.offset > 0)
   && (input.charAt(index - 1) == '\n'))
      return next(input,index,eflags,mymatch);
// Not BOL match
if ((eflags & RE.REG_NOTBOL) > 0)
      return null;
...
```

Figure 1: Original `RETokenStart.match` Method

Concerns Tangled in a Method
This portion of the study considered methods involving multiple concerns. Specifically, we looked at the case where concerns are encoded as different branches of a, possibly nested, `if--then--else` construct. This kind of concern encoding occurred in both of the sample systems. For example, in the gnu.regexp package, a method responsible for part of the regular expression matching process, `RETokenStart.match`, checks for characteristics of the kind of match allowed, such as whether or not matches may be across lines, and then performs the appropriate kind of match depending upon which conditional test succeeds. Figure 1 depicts a portion of this method. The jFTPd package contains similar constructs. For instance, the `FTPConnection.doCommand` method, which parses and processes FTP requests entered by a user, employs a similar conditional structure.

For the purposes of this study, we focused on separating concerns in the `RETokenStart.match` method. This method included three concerns: the multiline concern for handling matches across multiple lines, the "not beginning-of-line" (NotBOL) concern for ensuring particular matches occur at the start of a line, and the anchored concern for supporting matches of substrings. We wanted to be able to configure the system to support different combinations of matching functionality.

LSOC
Applying our lightweight separation of concerns approach resulted in the creation of three classes: one for each concern. Each of the classes introduced contained two static methods: `matchTest` and `matchResult`. The `matchTest` method encapsulates the conditional test from the original `if--then--else` structure to determine whether or not the concern applies. If it does apply, the `matchResult` method can be called to return the result of performing the match.

Given these classes, we modified the `match` method to use the classes encapsulating the concerns. For example, the portion of `match` dealing with the multiline concern was modified as follows.

```
int[] match( ... ) {
...
if ( Multiline.matchTest( ... ) )
   return Multiline.matchResult( this, ... );
```

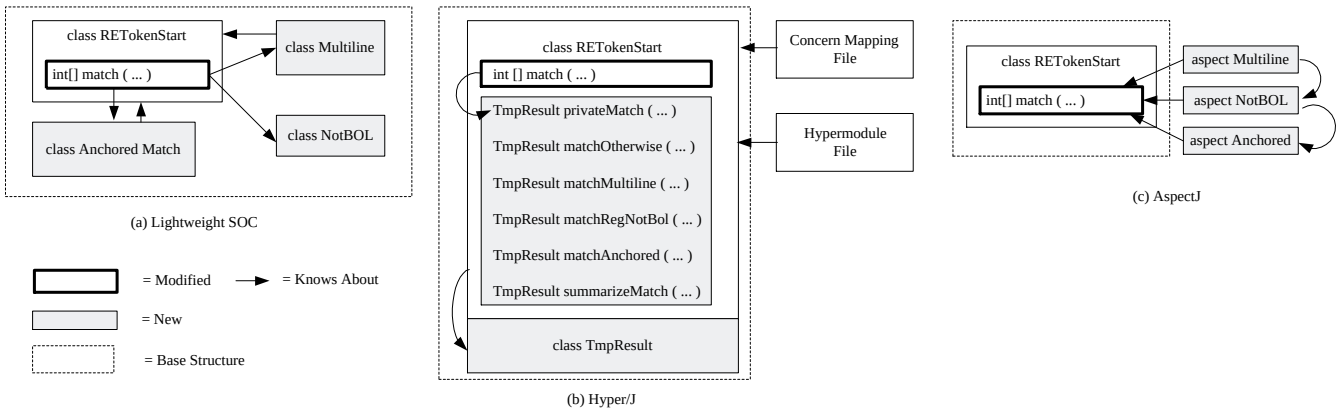

Figure 2: Method Restructurings

This restructuring permits the removal of the multiline concern with a tool such as `grep`. All lines including the lexical token `Multiline` simply need to be removed from the class.[5]

Figure 2a summarizes the restructuring performed to apply the lightweight separation of concerns approach. The figure shows which methods and classes were modified and added. The figure also depicts which pieces of the structure "know-about" other pieces of the structure. One piece of structure "knows-about" another if it names the other structural item. For instance, the `Multiline` class "knows-about" the `RETokenStart` class: an `RETokenStart` object is passed as a parameter to `matchResult` to provide access to necessary matching support. For simplicity, we have shown the "knows-about" relation only between pieces of structure that have been changed or added to separate concerns.

Hyper/J

With Hyper/J, the `RETokenStart.match` method was restructured into a number of new methods on the same class (Figure 2b). Three of the new methods—`matchMultiLine`, `matchRegNotBol` and `matchAnchored`—each encapsulate a concern. Another method, `privateMatch` encapsulates the default behaviour if no specific match routine applies. Relevant code from `match` was moved to these methods.

Each of these four methods has the same parameter list as the original `match` routine, but a different return type. Instead of returning an integer array similar to `match`, each routine returns an object of a newly introduced private class called `TmpResult`. The purpose of `TmpResult` is to bundle together the result of checking if a particular match applies with the actual result of performing that match.

The body of the modified `RETokenStart.match` method simply calls `privateMatch`, and then unbundles and returns the result embedded in the received `TmpResult` object.

With this restructuring, it is possible using Hyper/J to compose concern-specific functionality into `privateMatch`. Figure 3 outlines portions of the relevant Hyper/J files. The first line of the concern file states that by default all classes and methods are considered as part of the base—the Kernel—functionality of the system. Creating a system with only the Kernel functionality provides default matching behaviour only. The next two lines relate specific methods with specific concerns, which may or may not be chosen to be composed into `match` when a version of the system is configured.

As shown in Figure 3, composing concerns requires a few steps. First, to include a particular matching-related concern, we need to describe how the concern code should interact with the `privateMatch` method. In the example, we use an overall composition rule of `mergeByName` and then use `equate` statements to describe the correspondence between `privateMatch` and specific concern-related methods. Figure 3 shows the multiline and anchored match concerns being composed into the system. Furthermore, we need to express the ordering relationship between the concern-related methods by using `order` statements. Finally, we must describe how to compose the results of the merged methods. To accomplish this step, we introduce one more (static) method into the `RETokenStart` class, `summarizeMatch`, which receives an array of `TmpResult` objects from the composed methods. The method goes through the `TmpResult` objects in order and returns the integer array value from the first object with a test that succeeded.

AspectJ

Our AspectJ implementation involved the creation of an aspect for each concern (Figure 2c).

Each aspect included one `around` statement on the `RETokenStart.match` method. The code associated with one of these `around` statements is executed when an instance of `RETokenStart` is asked to execute the `match` routine. The code performs the conditional test associated with the matching concern to determine if the concern applies. If it does, code moved from the `match` method to perform the concern processing is executed, and the result returned. If the concern does not apply, the code continues with the normal method invocation of the `match` method.

[5] Particular formatting conventions must be followed for this approach to be successfully applied [13].

Concern Mapping File:

```
package gnu.regexp : Feature.Kernel
operation gnu.regexp.RETokenStart.matchMultiLine : Feature.MultilineHandling
operation gnu.regexp.RETokenStart.matchAnchored : Feature.MatchingRules
...
```

Hypermodule File:

```
...
  mergeByName;
  equate operation Feature.Kernel.privateMatch, Feature.MultilineHandling.matchMultiLine,
                   Feature.MatchingRules.matchAnchored;
  order action Feature.MultilineHandling.RETokenStart.matchMultiLine
    before action Feature.MatchingRules.RETokenStart.matchAnchored;
  set summary function for action DemoGnu.RETokenStart.privateMatch_matchMultiLine_matchAnchored
    to action DemoGnu.RETokenStart.summarizeMatch;
...
```

Figure 3: A Partial Hypermodule Specification for Separating Concerns in a Method

This continuation of the invocation will either cause another `around` associated with `match` to be run, such as the `around` for another match-related concern, or the original `match` routine itself.

The body of the original `RETokenStart.match` method was modified to perform only default matching functionality.

For the behaviour to correspond to the original system, we must be careful to apply the concerns in the appropriate order.

As shown below, we used the `dominates` keyword of AspectJ to indicate that one aspect, the `MultiLineAspect` is applied before the `NotBOLAspect`.

```
public aspect MultiLineAspect
   dominates NotBOLAspect
   of eachobject( instanceof(RETokenStart) )
```

Comparison
We compare the effect of the mechanisms on both the base and the separated code structure. In Figure 2, the base structure for each case is outlined in a dotted box: all other structural items shown are considered part of the separated structure.

Effect on Base Structure The structures resulting from our use of LSOC and Hyper/J reduce the cohesion of the `RETokenStart` class. From the viewpoint of a maintainer of the code, there is no apparent reason for the three concern classes introduced in the LSOC case. Similarly, a maintainer may find it difficult to understand why there are dead methods in the base structure using the Hyper/J approach: no calls appear in the base code to the concern-related and summarizing methods. In contrast, the AspectJ restructuring results in a clean base structure: there are no added pieces to the base structure, dead or alive.

Effect on Separated Code Structure Only the Hyper/J and AspectJ approaches separate code. Of these two, AspectJ provides a more complete separation: Figure 2c shows how the aspects are separated from the base structure and how "knows-about" is a relation from the separated- to the base-structure. In contrast, our application of Hyper/J resulted in the methods associated with separated concerns being located as part of the base structure. One advantage of this code locality is that it may be easier for developers maintaining the code to reason about how concerns fit into the system. In the AspectJ case, the aspects cannot be reasoned about in isolation of the `RETokenStart` class. One structural disadvantage of the AspectJ approach is that the specification of the ordering constraints amongst aspects couples the separated concerns. Coupling the aspects may make it more difficult to extend the system with new aspects.

Concerns Tangled Across Classes
This portion of the study considered concerns encoded as parts of methods and fields in different classes. For example, in the gnu.regexp package, a concern representing the kind of regular expression syntax to use is spread across two classes: `RE` and `RESyntax`. In jFTPd, the GUI concern, which encapsulates the display of the FTP server status, is split across the `StatusWindow`, `AboutBox` and `Handler` classes.[6]

For the purposes of the study, we focused on separating the GUI concern from the jFTPd system. In separating this concern, we wanted to support three different scenarios: running the system without the GUI functionality, running the system with the GUI, or incorporating into the system some other form of reporting status information. The original system allowed GUI functionality to be turned on or off using a run-time switch. We wanted to support the inclusion or exclusion of GUI functionality at system configuration-time. (We return to the issue of run-time versus configuration-time inclusion of a concern in Section 6.)

A perusal of the analyzed `StatusWindow`, `AboutBox`, and `Handler` classes indicated over 65% of the lines of code of the first class, all of the code comprising the second class, and approximately 1% of code in the last class was related

[6]We have dropped the `FTP` prefix used for jFTPd classes for simplicity.

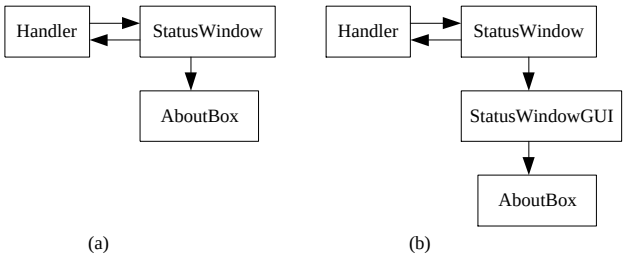

Figure 4: Class Interactions in GUI Concern

to the GUI concern. Furthermore, as shown in Figure 4a, the `StatusWindow` and `AboutBox` classes were isolated from the rest of the system by the `Handler` class This knowledge helped guide our application of each mechanism.

Our overall strategy to separate the concern was to split the `StatusWindow` class into two classes: a modified `StatusWindow` class which interacts with `Handler` but which does not display status information,[7] and a new `StatusWindowGUI` class which does display status information (Figure 4b). If a status information display was desired, the `StatusWindow` class would be configured to use the new `StatusWindowGUI` class. This strategy also involves modifying `Handler` to remove the existing code associated with the GUI concern. The `AboutBox` class is not affected since it included code only related to the GUI concern and because it is referenced only via the `StatusWindow` class.

For each mechanism, the major challenge involved the handling of join points in arbitrary parts of methods.[8] Given space limitations, we focus our descriptions on the overall structures resulting from the use of each mechanism and on our approaches to handling mid-method join points.

LSOC
Figure 5a summarizes the structure resulting from our application of the LSOC approach. This structure differs from our desired structure in two ways. First, the `StatusWindowGUI` class references the `Handler` class because some GUI events, such as particular button pushes, require handling by the `Handler` class. Second, `StatusWindowGUI` references `StatusWindow` to gain access to information to be displayed in the GUI and to dispatch events that occur in the GUI.

For the most part, splitting the `StatusWindow` class into two was straightforward. For each method in the class, we grouped code related to the GUI concern and moved it into methods on the `StatusWindowGUI` class. Code was moved to a method of the same name when the entire method related to the GUI concern or when the code was entirely at the start or at the end of the method. In any other case, code from the original class was moved to a new method on the new class. In place of the moved code in `StatusWindow`, we then inserted calls to methods on a `StatusWindowGUI` object named `win`.

For example, a set of GUI-related statements at the beginning of the `StatusWindow.startServer` method was moved to a `StatusWindowGUI.initStartServer` method. A new method had to be introduced because the `startServer` method also included GUI code at the end of the method that had to be moved. We replaced the moved code in `StatusWindow` with the call, `win.initStartServer()`. This call can be thought of as an explicit join point to the GUI concern as it explicitly names the concern.

Hyper/J
With Hyper/J, we decided to separate the concerns in this example by using concern-specific hierarchies. Taking this approach involved creating concern-specific versions of the classes of interest. Our intent was to be able to create independent hierarchies in the style of subject-oriented programming [5] that could then be composed. Independent concern-specific hierarchies may make it easier to build and manage different pieces of a system.

Figure 5b depicts the two hierarchies created. The Kernel hierarchy included versions of the `Handler` and `StatusWindow` classes. Two other pieces of structure were added to the Kernel hierarchy: the `IHandler` and the `IStatusWindow` interfaces, each of which describes the functionality available in its associated class. These interfaces provide a means of referring to functionality from classes in the GUI hierarchy. The GUI hierarchy included concern-specific analogues of the two classes in the Kernel hierarchy, `Handler_GUI` and `StatusWindow_GUI`, as well as the `AboutBox` class.

The classes in the Kernel hierarchy were modified to remove any GUI-related references or code. GUI concern code that resided at the start or in the middle of a method was replaced by a call to a new empty method introduced on the class. GUI concern code that resided at the end of a method was removed because we intended to cause a similarly named method from the GUI hierarchy to run after the Kernel hierarchy method. All GUI-related code was moved to similarly named methods on the versions of the classes in the GUI hierarchy. This movement of code and introduction of new methods enables the classes and methods to be implicit join points between the two hierarchies: the join point may or may not be used.

The two hierarchies can be merged by name. In the hypermodules file, we simply equate the class analogues from each concern. Then, for instance, when a reference is made to a particular method in the `Handler`, Hyper/J will ensure that the analogous method in `Handler_GUI` is also executed.

AspectJ
When separating the concern from the `StatusWindow` class using AspectJ, we had to make a choice about whether to move all of the GUI-related code to aspects or whether to move it to a combination of classes and aspects. We chose the combination approach because we thought capturing the

[7] We retain the name of the original class even though the GUI concern is removed for clarity in comparison.

[8] A join point is a location in the source code, or a point in the execution of a system, at which code or behaviour described in a separated concern may be integrated with the non-separated source or execution. This term is used in slightly different ways in different publications related to separation of concern mechanisms.

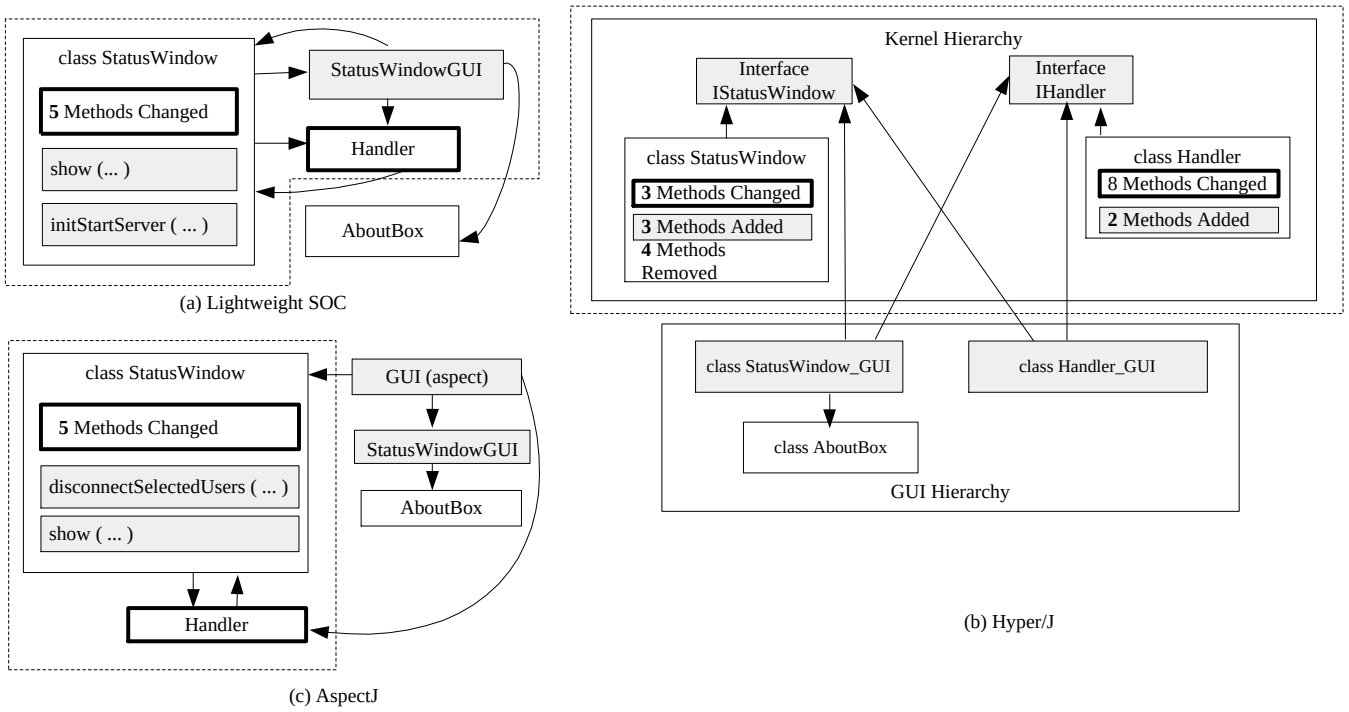

Figure 5: Class Restructurings

code as objects would be more explicit than embedding the code in aspects. Thus, as Figure 5c shows, we introduced both the `GUI` aspect and a new class, `StatusWindowGUI`.

This `StatusWindowGUI` class differs from the similarly named class introduced for the other two mechanisms in two ways. First, we chose to place some of the code from the start and end of methods in the aspect rather than creating new methods for this code in `StatusWindowGUI`. Second, the `StatusWindowGUI` class does not contain any references to either `StatusWindow` or `Handler`. Instead, these references are maintained in the aspect; the aspect is thus responsible for stitching the necessary pieces together.

The `GUI` aspect uses both `introduce` and `advice` statements to integrate concern code into the system. Seven of the ten `advice` statements are `before` or `after advice` statements. In the other three cases, new methods had to be introduced into the `Handler` class to provide join points in the middle of methods. For example, we had to add methods such as `notInteractive` in the middle of the `startService` method to overcome the lack of a run-time switch to indicate whether or not the system was running in interactive mode. The definition of the `notInteractive` method on the class handled the non-interactive case. The aspect overrides this method to handle the interactive case.

Comparison
As before, we compare the effect of the mechanisms on both the base and the separated code structure. Figure 5 indicates what we considered base and separated code in each case.

Effect on Base Structure For each of the approaches, the cohesion of each of the classes in the base structure was improved by separating the GUI-related code. One disadvantage of the Hyper/J and AspectJ approaches is that "odd" methods—methods with no apparent purpose—had to be introduced into some classes in order to support join points in the middle of methods. Comparing AspectJ to Hyper/J, the AspectJ restructuring required fewer "odd" methods because of the ability to have both `before` and `after advice` associated with methods.[9] From the viewpoint of the base structure, one advantage of the LSOC approach is that the join points are explicit in code and can be placed in the middle of a method, removing the need to add methods to provide join points.

Effect on Separated Code Structure The LSOC approach provides only limited separation. A new kind of status reporting could be introduced into the system by replacing the `StatusWindowGUI` class. However, unless the class were completely replaced, some new mechanism, such as the AbstractFactory pattern [3], would need to be added to the system in order to support the instantiation of a different class.

Both Hyper/J and AspectJ separate the GUI code more thoroughly from the base code. The use of separate hierarchies in Hyper/J makes it possible to consider the separated GUI code in the context of some structure. This structure may make it easier to understand the code. However, changes in the Kernel hierarchy may have to be reflected in the GUI hierarchy, potentially complicating maintenance activities. Since in the AspectJ case, we separated most of the code into classes rather than aspects, the end result may also be more straightforward to understand. Maintenance may be easier

[9]Hyper/J provides a similar bracketing operation but we chose not to use it in this example.

in the AspectJ case because the aspect, which contains the code connecting the base and separated structures, is short and straightforward.

5 STUDY RESULTS

The detailed descriptions in the previous section show that separation of concern mechanisms can be applied in several different ways, resulting in a variety of software structures. Not surprisingly, none of the structures produced are obviously "right", nor are any of the mechanisms obviously "perfect". In this section, we take a step back and consider the major lessons we learned from our study experiences in three areas: designing appropriate target structures, preparing a system to separate concerns, and restructuring the code.

Our discussion of each of these topics assumes that a developer has decided that separating concerns is an appropriate action.

Designing Appropriate Target Structures

Given a set of concerns to separate, particularly for an existing system, how can a developer design appropriate target structures for the base and separated code? Although it is too early to provide specific rules to guide this process, from our experiences, there are two questions we believe a developer should consider.

- How is the codebase to be managed and work to be distributed?
- What kinds of changes are expected in the base and concern code?

The management of the codebase and the distribution of work in an organization affects the kind and form of separation needed between the base and concern code. For example, if separate concerns are to be developed independently by different teams, it may be advantageous to take an approach similar to that of applying Hyper/J to jFTPd where we created separate class hierarchies for the base and the concern. Each team can then be responsible for a separate hierarchy. On the other hand, if only one team is involved with a codebase, it may be advantageous to represent the concern code as either "dead" code within the base structure, as we did when applying Hyper/J to gnu.regexp, or as aspects that have detailed knowledge about the base structure, as in the case of applying AspectJ to jFTPd. More separation, as in the separate class hierarchy approach, may make it easier to version and evolve base and concern code separately. Less separation, as in the "dead" code approach, may make it easier to reason about the base and concern code together.

In our experience, separation of concern mechanisms do not magically remove the need during software design to choose structures that accommodate anticipated changes. The selected concerns themselves likely relate to anticipated changes. However, it is not enough to simply identify the concerns, we must also choose structures to represent those concerns. Concerns might be represented using existing structural constructs, such as object-oriented classes, or as structural constructs unique to a particular mechanism, such as aspects.

Although we cannot provide any specifics about how to represent concerns, some general guidelines have emerged from our experiences. One guideline is to use the "knows-about" relationship as a means of ensuring that concern code is separate from the base code. For example, in general, it is not an advantage to have base code refer to separated code as arose when we applied the LSOC approach to the gnu.regexp package. This guideline also arose in another study in which AspectJ was used to build a web-based system [6].

A second guideline is to use the "knows-about" relationship to determine the degree of coupling between separated code representing different concerns. For instance, when we applied AspectJ to the gnu.regexp package, some of the aspects had to "know-about" each other to express ordering constraints. Such knowledge may require changes to existing aspects when a new aspect is added to the system or may become extraneous if an aspect is removed from the codebase. Longitudinal studies of codebases using separation of concern techniques are needed to validate and expand these guidelines.

Preparing for Separating Concerns

A concern may be easier to separate, and a better overall structure for the code may result, if advance work to prepare the software system is undertaken. Given our experiences, we suggest the following two steps be taken.

First, analyze and, as possible, refactor the codebase to encapsulate concerns in entire methods and classes. This refactoring must balance the advantages of encapsulating a concern within the regular object-oriented structure of the program with qualities of the codebase, such as its readability and maintainability. For instance, capturing the multiline matching concern in gnu.regexp as a class as we did when applying the LSOC mechanism was probably not an overall advantage because it decreased the cohesion of the `RETokenStart` class. In contrast, splitting the `StatusWindow` class when applying the LSOC mechanism to jFTPd was an overall benefit because it increased the cohesion of classes in the system.

Second, for concerns tangled in methods after the refactoring, analyze the methods and try to group together as many statements as possible associated with each concern. Grouping the statements may require analysis to understand which statements may be reordered. Try to move groups of statements related to concerns to the beginning and ends of methods since most mechanisms support such points as join points. The intent of this analysis and code rearrangement is to minimize the need for mid-method join points. Such join points should be avoided since, for several mechanisms, require the introduction of "odd" methods, which exist solely to provide access to the desired join points. These "odd" methods can also affect code quality. As separation of concern mechanisms evolve, they may introduce a means of identifying mid-method join points, for instance as call sites, removing the need to add new methods to provide mid-method join points.

Restructuring the Code
After the code has been prepared to separate a concern, the actual separation must still be performed. In our study, we performed all code restructurings manually. Although manual restructuring is obviously possible, it is time-consuming and error-prone. Automated support would ease the difficulty of restructuring the codebase and separating a concern. Automated support could help in several ways:

- to move fields from one class to another or to a new piece of structure, such as an aspect,
- to move methods from one class to another or to a new piece of structure, such as an aspect,
- to form new methods on the same class from a grouped set of statements, and
- to modify inheritance and interface associations.

These restructurings are all simple and can be supported by tools (e.g., [10]). However, a constraint that restructurings be meaning-preserving enforced by much of the work on automated restructuring needs to be relaxed for these tools to be helpful when restructuring to separate concerns. For instance, when restructuring parts of the jFTPd codebase into two separate hierarchies, it was not possible for the meaning of the system to be preserved: we were trying to remove functionality from the system! Moreover, for some mechanisms, restructuring tools need to be specialized to be able to move code into new pieces of structure, such as aspects.

Given that we found similarities between how concerns were encoded between the two packages we studied, such as the use of `if--then--else` constructs, it is also likely that restructuring patterns could be developed for particular concern encoding forms.

6 DISCUSSION

As we described earlier, our study format had many limitations. In this section, we consider various issues impacting the validity of our study.

Concerns Considered
Our study considered only a small number of concerns. We identified code related to these concerns largely based on a reading of the source code of the two packages. We believe this approach was reasonable to support the lessons reported because we focused on the structure of the encoding of the concerns in the code rather than on the semantics of the code. Similar concern structure has been reported elsewhere [2]. As a result, it is not crucial that the concerns considered, such as the matching concerns in gnu.regexp, would truly be of interest to separate. Furthermore, since we focused on fairly fine-grained structure—conditional statements and small numbers of classes—it was not as critical that we ensure we had identified all code related to a concern.

Application of Mechanisms
For each of the two cases, we applied each separation of concern mechanism in only one way. For each application, we have argued why we think our use of the mechanism was reasonable. However, we do not claim that our application of any of the mechanisms was optimal. For instance, our application of the Hyper/J tool to the jFTPd package involved the refactoring of code into separate hierarchies. This refactoring was not necessary to apply the tool. One advantage claimed for Hyper/J is its ability to remodularize code without changing it by associating methods and classes with particular dimensions and recombining those dimensions in different ways [12]. We could have applied Hyper/J in more of this fashion to jFTPd. However, such remodularizations, are dependent upon the appropriate join points being available in the code. As can be seen from our use of all the mechanisms, appropriate join points were not generally available. We had to restructure code to provide suitable points for composition. As a result, although our particular uses of the mechanisms can be criticized, and our detailed criticisms of the resulting structures should be considered within that context, we believe the overall lessons we reported have value.

Another possible criticism of our study format is that we did not globally redesign the two systems prior to trying to separate concerns. For example, we did not analyze the existing object-oriented structure and redesign it according to state-of-the-art design principles and guidelines, such as design patterns [3]. We did not undertake a global redesign for two reasons. First, for a system of any size for which a developer may want to separate a concern, it is not generally possible within time and cost limitations to perform a redesign. Second, our experiences marking the concerns in the code indicated that the designs of each system were reasonable: classes were chosen to hide design decisions and the classes encapsulated important data.

Run-time Feature Selection
Originally, the GUI functionality in jFTPd could be selected at run-time. In separating this functionality, we restructured and separated the code to support the configuration-time, rather than run-time, selection of the functionality. This change was reasonable because a developer may wish to create variants of the system in which particular functionality, such as the GUI, is not included in the system, but rather is purchased as an add-on unit. Furthermore, users may desire variants of the system that do not include such functionality in the executable so as to reduce the memory and CPU requirements of applications.

Overlapping Concerns
In each of the two sample systems, we considered independent concerns. In gnu.regexp, the three concerns were all related to matching but these concerns did not interact with each other. In jFTPd, we considered only the GUI concern. These are simple cases. Other concerns we marked in the two systems often overlapped: one piece of code was assigned to two or more concerns. The presence of overlapping concerns raises many questions for a developer attempting to restructure the codebase and separate the concerns. Should multiple copies of the code be executed if all concerns are composed into a system? Does the overlapping indicate a relationship between concerns, such that one concern must be included if another concern is included? Should the code only be executed if all concerns are composed into the system? Further study is needed to investigate these questions.

7 RELATED WORK

Since mechanisms for separation of concerns in the style considered in this paper are in their infancy, there have been few studies on the use and affect of the mechanisms on a system's code.

Walker, Baniassad, and Murphy have reported on the results of conducting two exploratory experiments about the impact of aspect-oriented programming [15]. Kersten and Murphy have reported on the experience of building a web-based system with aspect-oriented programming [6]. These studies touch on the structural impact of aspect-oriented programming. The experiments provided some preliminary evidence that separating concerns more distinctly from the base code may provide benefits. The web system case study reported on the advantages of limiting the "knows-about" relation between the aspects and base code to be from the aspects to the base code.

Lippert and Lopes have reported their experiences using AspectJ to simplify and unify parts of the exception handling structure in an existing system [9]. They focused on reengineering the exception handling where suitable join points were already available for the aspects. Carver and Griswold have described their experiences identifying concerns in the GNU sort program [2]. Carver has also reported on experiences applying Hyper/J to separate the identified concerns [1]. This body of work describes some of the difficulties isolating concerns and possible extensions to Hyper/J that would make separation easier.

This paper differs from these earlier efforts in considering more than one mechanism for separating a concern, and in focusing on the restructuring process needed to separate concerns in an existing system.

8 SUMMARY

In this paper, we have reported on a study in which various separation of concern mechanisms for object-oriented systems were applied to two common scenarios: separating concerns tangled within a method, and separating concerns tangled between classes. This exploratory study demonstrates the wide range of software structures that can result from applying these mechanisms. In addition to analyzing the tradeoffs between these different structures, we have provided a set of initial guidelines to help others choose an appropriate target structure, prepare their codebase for separating concerns, and perform the necessary restructurings. This analysis and set of guidelines can help software engineering researchers develop tools and techniques to aid the application of the technology,es and early adopters apply the technology.

ACKNOWLEDGEMENTS

We would like to thank Harold Ossher and Peri Tarr for their help with our applications of the Hyper/J tool, and Gregor Kiczales for comments on a draft of this paper. This work was supported in part by an NSERC research grant, an NSERC post-graduate fellowship, and an IBM University Partnership Program award.

REFERENCES

[1] L. Carver. A practical hyperspace application: Lessons from the option-processing task. Position Paper for Multi-Dimensional Separation of Concerns Workshop, ICSE, 2000.

[2] L. Carver and W. Griswold. Sorting out concerns. Position Paper for Multi-Dimensional Separation of Concerns Workshop, OOPSLA, 1999.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[4] N. Griffeth and Y.-J. Lin. Extending telecommunications systems: The feature-interaction problem. *Computer*, 26(8):14–18, 1993.

[5] W. Harrison and H. Ossher. Subject-oriented programming: A critique of pure objects. In *Proc. of OOPSLA*, pages 411–428, 1993.

[6] M. Kersten and G. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In *Proc. of OOPSLA*, pages 340–352, 1999.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of ECOOP'97*, pages 220–242, 1997.

[8] A. Lai and G. Murphy. The structure of features in java code: An exploratory investigation. Position Paper for Multi-Dimensional Separation of Concerns Workshop, OOPSLA, 1999.

[9] M. Lippert and C. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proc. of ICSE*, pages 418–427, 2000.

[10] W. Opdyke. *Refactoring Object-oriented frameworks*. PhD thesis, University of Illinois, 1992.

[11] H. Ossher and P. Tarr. Hyper/J: Multi-dimensional separation of concerns for Java. In *Proc. of ICSE*, pages 734–737, 2000.

[12] H. Ossher and P. Tarr. On the need for on-demand remodularization. Position Paper for Aspects and Dimensions of Concern Workshop, ECOOP, 2000.

[13] M. Robillard and G. Murphy. An exploration of a lightweight means of concern separation. Position Paper for Aspects and Dimensions of Concern Workshop, ECOOP, 2000.

[14] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proc. of ICSE*, pages 107–119, 1999.

[15] R. Walker, E. Baniassad, and G. Murphy. An initial assessment of aspect-oriented programming. In *Proc. of ICSE*, pages 120–130, 1999.