Separating Concerns with Hyper/J™: An Experience Report[*]

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

May 1, 2000

A Position Paper for the ICSE 2000 Workshop on Multi-dimensional Separation of Concerns

Abstract

In earlier work, we conducted an exploratory investigation of concerns in two existing Java™ packages: jFTPd and gnu.regexp. Two separate developers marked concerns in the source for each package: these concerns were then compared and analyzed. In this paper, we describe the next step of our investigations: the use of the IBM Hyper/J™ tool to separate and configure the identified concerns. We describe the various kinds of hyperspaces, hypermodules, hyperslices, and concern mappings we used to describe our previously identified concerns. We also discuss code restructurings we used to enable the capturing and composition of concerns.

1 Introduction

A number of tools are appearing to help software developers explicitly separate different concerns in their programs. Two examples of such tools are AspectJ™,[1] [LK98] and Hyper/J™.[2] Each of these tools provides general mechanisms for separating concerns. In AspectJ, a developer specifies crosscuts and crosscut actions: the crosscuts define points in the program to which the actions apply. For example, a crosscut definition might identify all `get`ter methods of a subset of the classes in the program; an associated crosscut action might check a database to ensure the most up-to-date information is returned by the method. In Hyper/J, a developer chooses the most appropriate decomposition for a piece of a program and can then combine the pieces into a coherent system by stating a series of composition rules [TOHS99]. For example, a developer might separately describe two concerns of a system, such as the ability to make certain classes persistent and some functionality of the system, and then might combine these two concerns together.

Given the presence of mechanisms to help explicitly separate concerns, it is now possible to explore more concretely the kinds of concerns that each approach is best suited to capture. In this paper, we focus on the approach supported by the Hyper/J tool, applying the tool to separate concerns[3] we identified previously [LG99] in two existing Java™ [GJS96] packages: gnu.regexp,[4] and jFTPd.[5]

We begin with an overview of the range of concerns we tried to separate with Hyper/J (Section 2) and a brief

*This research was funded by an IBM University Parternship Program award and an NSERC research grant.

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

[2]`http://www.research.ibm.com/hyperspaces`

[3]To keep the terminology clear with respect to Hyper/J, we are using the term *concerns* to refer to those items described as *features* in our earlier work.

[4]Written by Wes Biggs (Version 1.0.8)

[5]Written by Brian Nenninger (Version 1.3)

overview of Hyper/J itself (Section 3). Then, we describe a variety of ways in which we captured the concerns in Hyper/J (Section 4). As part of this description, we discuss code restructurings we performed to enable the capturing and composition of concerns.

2 Concerns

The concerns we tried to separate with Hyper/J had been identified by two separate markers using a source code mark-up tool called the Feature Selection tool [LG99]. The 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 concerns.

The concerns were selected based on a variety of criteria. Some concerns encapsulated standards conformation, such as pieces of code supporting the FTP protocol in jFTPd. Other concerns encapsulated a configuration of the software package: 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 the code 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.

Table 1 repeats a table presented in our earlier work that summarizes the range of concerns selected in each package by each marker.

3 Hyper/J

Hyper/J is a tool developed at IBM T.J. Watson Research Center to support the "multi-dimensional separation and integration of concerns in standard Java software" [TO00, p. 1]. For a full description of Hyper/J, the reader is referred to the Hyper/J documentation. In this section, we provide a brief overview of the concepts in Hyper/J to facilitate the reading of this paper.

The Hyper/J tool permits a set of Java source files to be decomposed along multiple dimensions of concern simultaneously. For example, Java classes may be considered as classes in the Class dimension while particular methods in the classes may be considered operations in the Feature dimension. Each dimension may be partitioned into a set of concerns. For example, the Feature dimension in jFTPd can be partitioned into several concerns along the lines of those identified in Table 1.

The Hyper/J tool also supports the integration of different dimensions of concerns through the description of composition rules. For example, with jFTPd, a developer might describe how to integrate concerns of interest for a particular configuration of the tool, such as support for connection commands and directory commands, but might choose not to include a concern that supports listing commands.

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

- a *hyperspace* file that describes the Java class files that can be manipulated by Hyper/J,
- a *concern mapping* file that describes which pieces of the Java source map to each dimension of concern, and
- a *hypermodule* file that describes which dimensions of concern should be integrated (i.e., which hyperslices)[6] and how that integration should proceed.

As a simple example, consider the two classes shown in Figure 1. Class `A` has a method called `print` that simply prints the string "Hello" to standard output. Class `B` has a method called `print` that prints the string "there world" to standard output. We have been asked to create a system that prints "Hello there world" to standard output. Using Hyper/J, we can describe that class `A` belongs to the `Kernel` concern, while class `B` belongs to the `New` concern. These two methods can be composed to produce the desired behaviour.

Figure 2 shows the three input files necessary to Hyper/J to perform the desired decomposition and composition. The hyperspace file describes that we are interested in manipulating everything in the `aTest` package. The concern mapping file describes the mapping of classes to different concerns. Finally, the hypermodule file defines that the composed system will include functionality from both the `Kernel` and the `New` concerns, that functionality from each concern should be merged based on names,

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

No.	Package	Marker	Concern	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 *	Concerns 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

Table 1: Concerns Selected in gnu.regexp and jFTPd

```
package aTest;
class A {
   void print() {
      System.out.println("Hello");
   }

   static void main(String args[]) {
      A a = new A();
      a.print();
   }
}
```

```
package aTest;
class B {
   void print() {
      System.out.println(" there world");
   }
}
```

Figure 1: Two Example Classes

and that the `A` class corresponds to the `B` class in the merge. When processed by Hyper/J, this system produces the desired output.

4 Describing Concerns in Hyper/J

Most of the concerns we identified in the jFTPd and gnu.regexp packages are scattered and tangled through the code bases. Few concerns comprised entire methods or classes: instead, most concerns comprised a few lines of code within a method in one class, a few lines within a method in another class, a field in yet another class, and so on. Since existing separation of concern mechanisms, including Hyper/J, support separation along existing structural boundaries in the code base, such as method boundaries and fields, encapsulating and separating concerns often required the restructuring of code before Hyper/J could be applied.

To consider the range of possibilities of expressing concerns with Hyper/J, each author of this paper applied Hyper/J separately to one of the two packages. At this point, we have not yet exhaustively separated out and recomposed all concerns that we had identified previously. In this section, we describe the variety of approaches we used in applying Hyper/J to capture individual concerns. For each approach, we discuss the benefits and limitations of the approach.

4.1 Restructuring to New Methods on the Same Class

Some methods involve multiple concerns. For example, in the gnu.regexp package, one of the routines for performing a match (`RETokenAny.match`) checks for characteristics of the kind of match allowed, such as whether or not it is an anchored match, and performs the appropriate processing. This routine consists of an `if--then--else` sequence of code: check if the characteristic holds, and then perform the appropriate kind of match. The jFTPd package contains similar constructs, most notably in the `FTPConnection.doCommand` method, which parses each FTP request entered by the user and calls the appropriate method to handle the request. The different characteristics corresponded to different concerns. Using Hyper/J, we wanted to be able to separate and then mix-and-match in the desired concerns for a particular configuration of the system.

One way to accomplish the separation of the different concerns with Hyper/J is to restructure a method with tangled concerns into separate methods on the same class, each providing a different concern. For `RETokenStart.match` from gnu.regexp, this involved creating separate (private) methods for matching across multiple lines (`matchMultiLine`), matching when not at the beginning of a line (`matchRegNotBol`), and performing an anchored match (`matchAnchored`). Each of these methods had the same parameter list as the original `match` routine. However, the return type for each of these methods was altered. Instead of re-

Hyperspace File:
```
hyperspace Test
   composable class aTest.*;
```

Hypermodule File:
```
hypermodule Test
   hyperslices:
      Feature.Kernel;
      Feature.New;
   relationships:
      mergeByName;
      equate class Feature.Kernel.A,
                   Feature.New.B;
```

Concern Mapping File:
```
class A : Feature.Kernel
class B : Feature.New
```

Figure 2: Example Hyper/J Input Files

turning an integer array similar to `match`, each routine returned an object of a newly introduced private class called `TmpResult`. The purpose of `TmpResult` was to bundle the result of checking if a particular match applied and the actual result of performing that match. The introduction of this new return type was necessary to determine, when composing the methods, which test had succeeded, and thus, which value should be returned.

The body of the original `match` routine was altered to call a newly introduced (private) `privateMatch` routine. This routine was introduced for two reasons. First, `privateMatch` shared the same signature as the newly introduced specific match routines. Second, `privateMatch` comprised the default behaviour if no specific match routine applied. After calling `privateMatch`, the `match` routine unbundles the result from the temporary object and returns the provided integer array as the result of the `match`.

With this restructuring, it is possible to describe to Hyper/J which methods participated in which concerns and how to compose the concerns into a desired `match` routine. Specifically, the concern mapping file included lines as shown in Figure 3. The first line ensures that the `match` and the `privateMatch` methods are both considered part of the `Kernel` feature. The next lines separate two of the newly introduced methods into specific features which may or may not be chosen to be composed into `match`.

Composing the concerns in a hypermodule requires a few steps (see Figure 3). The overall composition rule in use was `mergeByName`. Thus, to ensure that the methods for specific match handling were considered as part of `privateMatch`, it was necessary to use an `equate` statement. Furthermore, because there was an ordering to the testing of the match characteristics, it was necessary to use `order` statements to express the relationship between the methods. Finally, we needed to introduce one more (static) method on the `RETokenStart` class to handle the return values from the composed method (`summarizeMatch`) and we set the summary function for the composed method to this newly added method. The `summarizeMatch` method simply receives an array of `TmpResult` objects from the composed methods, goes through them in order and returns the integer array set in the first object that recorded a successful test.

The example shown in Figure 3 focuses on one method in one class. The approach is easily extended to multiple methods on multiple classes.

Benefits There are four benefits to this approach.

1. The composition rules—a combination of `equate`, `order` and `set summary`—needed to describe the merging of the split methods are easy to state and easy to understand.

2. The restructuring is straightforward to apply: the bodies of existing methods need to be altered and new

Concern Mapping File:

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

Hypermodule File:

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

Figure 3: (Approach #1) Part of Hypermodule Specification for Restructuring Methods in Class

methods need to be added to the existing class. Tool support can help with these restructurings [Opd92].

3. Only the code in the class with the method being untangled was affected. Thus, there is little impact to how the system is built.[7]

4. Because the impact of the restructuring is one class, it is easy to reason about the change. The developer need only analyze the one class.

Limitations There are three limitations to this approach.

1. When the order of the composition of the separated methods matters (as it did in the example above), it is difficult to capture the ordering constraints. The composition rules we used in Hyper/J make the ordering explicit, but the ordering information needs to be known to developers who are going to write such Hyper/J specifications.

2. The composition of the separated methods and the determination of the final result by a summary function means that the separated methods cannot have side-effects. Whereas in the original program, a method would only be executed if the test succeeded, in the composed program, all methods will be executed and the results then checked.

3. The class may be harder to comprehend. The separated methods must have the same parameter list as the original method, even if some of those methods do not require all of the parameters. It may be harder to understand the class because more methods have been introduced whether the reason for separating the methods is not clear.

[7] The reader may have noted that the restructuring above requires that some functionality be composed into `privateMatch` or the system may not work correctly.

4.2 Restructuring Each Class to Multiple Classes

A second way to accomplish the separation of the different concerns with Hyper/J is to subdivide each class, and hence each method therein, into concern-specific classes. For example, in marker #2's feature identification for jFTPd, the original package contained an `FTPStatusWindow` class composed of Kernel, Debugging, GUI, and User Interface features, and an `FTPAboutBox` class composed of GUI and User Interface features. After decomposition, there were hierarchies for the GUI and User Interface features that contained feature-specific analogues to `FTPStatusWindow` and `FTPAboutBox`, while the hierarchies for the Kernel and Debugging features contained feature-specific analogues to `FTPStatusWindow` but not to `FTPAboutBox`. If concerns were truly independent, then there could be completely separate hierarchies of concern-specific classes implementing each concern without reference to the other hierarchies. The presence of such independent hierarchies, i.e., subjects, could reap the potential benefits of subject-oriented programming [HO93].

While the declarative completeness guaranteed by hyperslices together with their ability to select individual methods and fields from classes may not require the explicit decomposition of each class into concern-specific analogues, this restructuring lent a clarity to what comprised each hyperslice.

However, since Hyper/J operates on class files, the concern-specific classes have to be compilable and, hence, declaratively complete before Hyper/J can manipulate them. If concerns were independent, this would not be problematic; when interdependences do occur, declarative completeness must be manually ensured by adding appropriate stub methods to concern-specific classes when they refer to methods that are only present in other concerns. Sometimes this is a reasonable and satisfactory course of action, but when the stub method has no semantics that follows from those of the concern itself, the conceptual cohesiveness of the concern (and thus, of the offending concern-specific class) is flawed.

Furthermore, when a value-returning stub method is to be used in a concern-specific class, it either has to return a default value or throw an exception indicating that a stub method has been invoked (and should not have been). The latter effect is analogous to the error message produced by Hyper/J when an abstract method[8] automatically introduced for declarative completeness is neither merged nor overridden by some concrete method.[9] The significant difference is that the presence of an exception throwing stub can escape notice until run-time and until the appropriate execution path is traversed, whereas a method automatically introduced by Hyper/J will cause a loading or verification error if still present in the composed class. The use of only a select subset of concerns within a program is questionable when a concern that is being left out of a composition can be the only one providing an implementation appropriate for replacing these abstract or stub methods.

Consider the example in Figure 4 from jFTPd. The `handler` field was identified as the Kernel feature, as were the first two statements of the `stopServer` method. The third statement was identified as the User Interface feature. To eliminate the resulting dependence on `handler` within the User Interface feature, a new method, `serverOn()`, was added to this class that then delegated to `handler`. The Kernel-specific class then received the implementation of `serverOn()` that performed the delegation, while the User Interface feature received a stub method (see Figure 5). This stub method could either return a default value, such as `true`, or throw an exception to indicate that a stub method was actually invoked. The default value choice is not ideal, however, since one does not always want the same value to be returned, for example.

Benefits There are two benefits to this approach.

1. It might be easier to manage and version separate hierarchies semi-independently via the true benefits of subject-oriented composition [OKK^{+}96], which is effectively a restricted form of the mechanism provided by Hyper/J [TOHS99, p. 116].

2. The structure of hyperslices, in the absence of analysis tool support, is clearer as no doubt can exist as to which concern a given piece of code belongs.

[8] A method possessing no invokable body.

[9] Merging or overriding an abstract method with a concrete method means that there is one method body to be executed when the composed method is invoked—the lack of a body for the abstract method is ignored.

```
public class FTPStatusWindow extends Frame {
  FTPHandler handler;

  public void stopServer() {
    handler.stopService();
    disconnectAllUsers();
    setOnOffLabel(handler.serverOn());
  }

  ...
}
```

Figure 4: (Approach #2) An Example of Field Access Tangling Between Concerns

Limitations There are three limitations to this approach.

1. The lack of independence of concerns causes difficulty in actually constructing separate hierarchies. For example, stub methods need to be manually inserted.
2. The separated classes need to be individually compilable whereas, in using Hyper/J directly to pull out methods and fields from the original class, the original class had to be compilable, a less stringent requirement.
3. Throwing exceptions from stub methods can create systems that will unexpectedly fail at run-time only when particular execution paths are traversed. Analysis tool support could indicate the presence of such stub methods within the composed classes.

4.3 Restructuring Asymmetric Relationships using Bracket

A third approach to separation of concerns with Hyper/J resulted from noticing asymmetric relationships between concerns. For example, the User Interface concern in jFTPd primarily consists of `FTPStatusWindow` and its interactions with `FTPHandler`. The join points within `FTPHandler` are responsible for creating and sending events to an instance of `FTPStatusWindow`. `FTPHandler` provides the core functionality of handling FTP connections while `FTPStatusWindow` only provides user interface functionality. Although it was possible to capture the join points by an `equate` relationship, an `equate` inadequately expresses the asymmetry between the concerns. The `bracket` relationship seems more appropriate for expressing this asymmetry.[10]

Using the bracket relationship, developers can associate the execution of a target instance method with the execution of other related instance methods. For instance, a developer can state that one method should run after another method. At compile time, each instance method has a notion of a self or `this` object. Unfortunately, the bracket relationship does not provide a way for a bracket'ing method to refer to the bracket'ed object.

For example, consider the method `setupDone` before the User Interface concern was separated, as shown in Figure 6. The `setupDone` method passes `this` to the constructor for `FTPStatusWindow`. Ideally the bracket after relationship could be used to insert the call to `FTPStatusWindow`'s constructor, but it is not possible for `FTPStatusWindow` to refer to the target object (an instance of `FTPHandler`) there.

Figure 7 shows the solution to this problem. This solution provides a clean separation of the User Interface feature from the `FTPHandler` code. However, it requires the parameter list to `setupDone` be modified to include a parameter for the `ConnectionHandler` object so that the bracket'ing method can access the necessary information. This change requires the modifica-

[10]After restructuring the User Interface concerns, we noticed that the `bracket` relationship seemed similar to the `advice crosscut` in AspectJ.

```
public class FTPStatusWindow/*Kernel*/ extends Frame {
  FTPHandler handler;

  boolean serverOn() {
    return handler.serverOn();
  }

  public void stopServer() {
    handler.stopService();
    disconnectAllUsers();
  }

  ...
}

public class FTPStatusWindow/*UserInterface*/ extends Frame {
  boolean serverOn() {
    return true;
    /* Alternatively: throw new RuntimeException("Stub method called"); */
  }

  public void stopServer() {
    setOnOffLabel(serverOn());
  }

  ...
}
```

Figure 5: (Approach #2) Untangling Field Access Produces Unsatisfying Results

tion of all call sites for for `setupDone`. Fortunately, there was only one such site. Additionally, from the perspective of maintaining this code, it may not be immediately obvious to a developer why `setupDone` requires the `ConnectionHandler` argument.

The code in Figure 7 represents a first attempt to separate the User Interface concern. During this attempt, no code was added to instantiate `UIUsage`. This was a developer error, but it resulted in working code—apparently "magically"; somehow, the `win` field obtained a reference to a particular `FTPStatusWindow` instance and retained it between the calls to `newWindow` and `connectionsChanged`. Stated another way, even though there seemed to be no object instantiation that contained `win`, both `newWindow` and `connectionsChanged` referenced the same `FTPStatusWindow` instance. In retrospect, this was a surprising result.

Examination of the `FTPHandler` class generated by Hyper/J quickly revealed how this was possible: Hyper/J made `FTPHandler` a subclass of `UIUsage`. Thus, the existing instance of `FTPHandler` provided the context for `win`. In light of this discovery, it now seems possible for bracket'ing methods to refer to the bracket'ed object through `this`. However, it raises the problem of multiple brackets from different classes on one class. The `bracket`'ed class must subclass each of the `bracket`'ing classes. This is particularly a problem in

```
public class FTPHandler implements ConnectionHandler {
  public void setupDone() {
    ...
    startService();
    ...
    (new FTPStatusWindow(this)).show();
  }
  ...
}
```

Figure 6: (Approach #3) Reference to `this`

Java where multiple inheritance is not allowed.

Benefits There are two benefits to this approach.

1. Similar restructuring has been applied in systems using AspectJ. Developers with previous AspectJ experience can apply similar decomposition strategies with Hyper/J.

2. The `bracket` relationship can take optional arguments `$ClassName` and `$OperationName` to provide some context information for structuring that is not available in other composition relationships.

Limitations There are two limitations to this approach.

1. The bracket relationship is only suitable for join points which lie on method boundaries. For example, if the join point for some concerns is within a conditional statement, `bracket` can not be used without some restructuring.

2. To provide context for `bracket`'ing methods, the `bracket`'ed method's arguments may be modified. The modifications may seem unnecessary outside of the context of the `bracket` relationship. An example of this is the method `setupDone` in Figure 7.

5 Discussion

In addition to those raised above, we found a few more general issues in our investigations.

A Separable If–Then–Else Construct Long, nested `if--then--else` constructs occurred within both packages in which individual `else-if` blocks were part of different concerns. To solve this in a general manner that worried about side-effects and ordering, each one of these blocks would have to be moved within its own method, and these resultant methods would have to be merged in the same order in which the corresponding blocks occurred in the original method. Each of these methods would require all the values of local variables and parameters in the original method to be encapsulated in a “state” object that would be passed to each, in turn, along with a boolean indicating whether a successful branch had been found yet or if the current test with its attendant side effects should proceed. The values within the “state” object would be altered just as the original local variables and parameters were. Finally, a boolean would be returned to indicate whether this test had succeeded or not. Fortunately, this general solution was not needed in the actual cases found in the packages.

Faking Mid-Method Join Points Hyper/J has no direct means of merging one concern-specific method into the midst of another, rather than neatly at its start or end. For example, consider the method in Figure 8; here the first and third statements are in the Directory Commands feature, while the second is in the Debugging feature. In order to provide a “hook” onto which Hyper/J can merge the debugging statement, it is necessary to make a new method, move either the preceding or following expression, statement, or block to this method, and call the new method in place of the moved code. In Figure 9, we chose to move the constructor call to `File` within the new method `FileHook`. If it were possible to perform merges with call sites, we could merge the debugging statement with this particular constructor call to `File` and avoid the artificiality of creating and calling the `FileHook`s from each feature, as well as the artificiality of the structure of the Debugging-specific `FileHook`.

Overlapping Features In the jFTPd package, it frequently happened that marker #2 assigned a given piece of source code to multiple concerns. At first glance, this seemed to indicate that these pieces of code should, thus, belong to all concerns marking that code; rather than repeating a piece of code in each concern and then merging the pieces to run sequentially, one could be selected to override the others. It often happened that a piece of code would be assigned to one feature while a subset of it was additionally assigned to another; this is problematic as the code that belonged to the first feature but not to the second was often necessary for the correct functioning of the subset belonging to both features, such as code initializing variables. It seemed as though the second concern was therefore dependent on the first, i.e., the first *had* to be present if the second were to be present and the resulting system were to be functional. In cases where the dependences were in one direction in one part of the system, but in the other in a different part of the system, it seemed that the both concerns had to be present if either one was. For example, Figure 10 shows a snippet of code the entirety of which was identified with the Client Interaction feature, and the last `else-if` block of which was additionally identified with the List Commands feature. Since the List Commands portion depends on the correct initialization of the `handled` and `upline` variables, including this portion of code in a composed system without including the initialization portion would not yield a functional system.

However, it is possible that the situation is more interesting than this: code segments that are shared by multiple concerns, perhaps, should be included in a composition only when *all* of those concerns are included in that composition, and not the rest of the time. This possibility was not realized until after our investigation, so it is not clear whether leaving out such shared code would allow each concern to be independently functional when not all concerns participating in sharing it are present in a composed system.

6 Summary

In this position paper, we have outlined a variety of techniques we have tried for encapsulating and composing concerns in two existing Java packages: jFTPd and gnu.regexp. Each of these techniques involves different trade-offs. We plan to continue applying and analyzing Hyper/J to these packages to develop a better understanding of which techniques are best suited to help separate concerns in an existing system. We believe this work will also help develop insights into appropriate design strategies for building new systems with Hyper/J.

References

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

[HO93] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proc. of OOPSLA*, pages 411–428, 1993.

[LG99] A. Lai and Murphy. G.C. The structure of features in Java code: An exploratory investigation. 1999 OOPSLA Workshop on Multi-dimensional Separation of Concerns in Object-Oriented Systems, October 1999.

[LK98] C. Lopes and G. Kiczales. Recent developments in AspectJ.™ In *ECOOP '98 Workshop Reader*, pages 398–401. Springer-Verlag, 1998.

[OKK+96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.

[Opd92] W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign , IL , USA, 1992.

[TO00] Peri Tarr and Harold Ossher. *Hyper/J User and Installation Manual*. IBM Research, 2000.

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. of the 1999 Int'l Conf. on Soft. Eng.*, pages 107–119, Los Angeles, USA, 16–22 May 1999.

```
public class FTPd {
  public static void main(String args[]) {
    ...
    handler.setupDone(handler);
  }
}

public class FTPHandler implements ConnectionHandler {
  public void setupDone(ConnectionHandler h) {
    ...
    startService();
    ...
  }
  ...
}

public class UIUsage {
  FTPStatusWindow win = null;

  public newWindow(ConnectionHandler h) {
    win = new FTPStatusWindow(h);
    win.show();
  }

  public void connectionsChanged() {
    if (win != null)
      win.connectionsChanged();
  }
  ...
}
```

Figure 7: (Approach #3) Solution Permitting Reference to `this`

```
public boolean doCwdCommand(String line) {
  ...
  String newPath = makeFilePath(newDirPath);
  if(DEBUG) System.out.println(newPath);
  File f = new File(newPath);
  ...
}
```

Figure 8: A Concern-Specific Statement in the Middle of a Method

```
public boolean doCwdCommand/*DirectoryCommands*/(String line) {
  ...
  String newPath = makeFilePath(newDirPath);
  File f = FileHook(newPath);
  ...
}

public File FileHook/*DirectoryCommands*/(String path) {
  return new File(path);
}

public File FileHook/*Debugging*/(String path) {
  if(DEBUG) System.out.println(path);
  return null;
}
```

Figure 9: The Artificial Solution to Obtain a Mid-Method Join Point

```
public void doCommand(String line) {
  ...
  boolean handled = false;
  // ok, so this isn't efficient
  String upline = line.toUpperCase();

  try {
    if(upline.startsWith("USER ")) {
      handled = doUserCommand(line);
    }
    else if(upline.startsWith("PASS ")) {
      handled = doPassCommand(line);
    }
    // The next else-if block is in List Commands
    else if(upline.equals("LIST") || upline.equals("NLST")) {
      handled = doListCommand(line);
    }
    ...
  }
  ...
}
```

Figure 10: Multiple Concerns Marking a Single Piece of Code