4-10-2009

IVCon: A GUI-based Tool for Visualizing and Modularizing Crosscutting Concerns

Nalin Saigal
University of South Florida

IVCon: A GUI-based Tool for Visualizing and Modularizing Crosscutting Concerns

by

Nalin Saigal

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Jay Ligatti, Ph.D.
Adriana Iamnitchi, Ph.D.
Dewey Rundus, Ph.D.

Date of Approval:
April 10, 2009

Keywords: Software engineering, Code maintenance, Aspects, Code refactoring, Code isolation

ACKNOWLEDGEMENTS

This project is joint work with Dr. Jay Ligatti, and I am extremely grateful to him for his inspirational guidance through every stage of the project. This thesis derives from an earlier technical report titled “Defining and Visualizing Many-to-many Relationships Between Concerns and Code” [1]. I would also like to acknowledge the National Science Foundation as my research was supported by them under Grant No. CNS-0742736.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

IVCon: A GUI-based Tool for Visualizing and Modularizing Crosscutting Concerns

Nalin Saigal

ABSTRACT

Code modularization provides benefits throughout the software life cycle; however, the presence of crosscutting concerns (CCCs) in software hinders its complete modularization. This thesis describes IVCon, a GUI-based tool that provides a novel approach to modularization of CCCs. IVCon enables users to create, examine, and modify their code in two different views, the *woven view* and the *unwoven view.* The woven view displays program code in colors that indicate which CCCs various code segments implement. The unwoven view displays code in two panels, one showing the core of the program and the other showing all the code implementing each concern in an isolated module. IVCon aims to provide an easy-to-use interface for conveniently creating, examining, and modifying code in, and translating between, the woven and unwoven views.

CHAPTER 1

INTRODUCTION AND RELATED WORK

Code modularization provides many software engineering benefits; it makes code easier to write, understand, and maintain. Conventionally, software engineers try to separate code segments that are orthogonal in their functionality into distinct modules. However, in practice, software does not decompose neatly into modules with distinct, orthogonal functionality. For example, code that displays a popup window notifying users about a failed login attempt may be present in a login module, while (partially) implementing various other functional concerns such as security, GUI, and authentication; it may be equally reasonable for the window-popup code to be located in a security, GUI, or authentication module, and at various times it may be more convenient to write, view, or edit the window-popup code in the context of these other modules. Tarr et al. call this problem the "tyranny of the dominant decomposition" [2]. Although it is useful to modularize the same code segment in various ways throughout the software life cycle, current programming paradigms only allow modularization in fixed and limited ways (e.g., into functions and objects).

Conversely, functional concerns of software typically require many lines of code to implement, and that code is usually *scattered* throughout several modules. For example, code implementing a security concern may be scattered throughout login, logout, and network-socket modules. Thus, code segments implementing a functional concern may crosscut through other functional concerns of the program; such code segments implement *crosscutting concerns* (CCCs). Modularizing a CCC involves collecting and displaying in one place all the scattered code implementing that CCC. Isolating concern code in this way benefits programmers because it relieves them from having to browse through the whole program to find, study, or update a single software concern. For example, updating security code scattered throughout an application is much more difficult than updating an isolated security module.

```
JOptionPane.showMessageDialog(mainWindow.frame, "Welcome " + userName +
   ". Current time is " + format(System.currentTimeMillis()),"Welcome",
      JOptionPane.INFORMATION MESSAGE );
```

Figure 1.1. Sample code demonstrating the motivation behind token-level granularity

This thesis describes IVCon, a GUI-based tool that provides a novel approach to modularization of CCCs. IVCon users can switch back and forth between two equivalent views of their code, the *woven view* and the *unwoven view*. The woven view displays program code in colors that indicate which concerns various code segments implement (based on users' explicit assignments of those code segments to concerns). The unwoven view displays code in two panels, one showing the *core* of the program (i.e., all code not assigned to any concern) and the other showing all the modularized concerns, each displayed in isolation. Users can create, examine, and modify code in both views. The process of extracting concerns from the main code to produce the unwoven view is called *unweaving*, whereas the process of inlining concerns into the core program to produce the woven view is called *weaving*. Although we have implemented, and this paper discusses, IVCon in a Java environment, we believe the principles underlying IVCon apply to software in any language.

IVCon permits many-to-many relationships between concerns and code. That is, users can assign scattered code segments to the same concern but can also assign a single code segment to multiple concerns. We call code that has been assigned to multiple concerns *multi-concern code*. In addition, IVCon enforces *token-level granularity* in concern assignment; code assigned to a concern must begin at the beginning of a source-language token and end at the end of a source-language token. Allowing finer granularity in concern assignment (e.g., character-level granularity) would be inappropriate because tokens are the core semantic units of programming languages and of concerns implemented in those languages. On the other hand, requiring coarser granularity in concern assignment (e.g., line-level granularity) would be inappropriate as well. For example, consider the code in Figure 1.1. Token-level granularity enables assignment of just the `System.currentTimeMillis()` code segment to a `SystemCall` concern, while coarser concern-assignment granularities, such as line- or statement-level granularity, lack the precision needed for such a concern assignment. With token-level granularity, a user could even assign just the method name `currentTimeMillis` to the `SystemCall` concern. We consider token-level granularity to be the perfect balance for assigning code to concerns.

Concern assignments in IVCon can also be thought of as labels that document the functionality, or other grouping, of code segments. In this sense, IVCon serves as a code-documentation tool, with the benefit that software engineers can view and edit, in one module, all the code documented as being relevant to any concern.

1.1 Related Work

The most closely related body of research is in the domain of Aspect Oriented Programming [3]; like IVCon, AOP strives to ease the specification and manipulation of CCCs in software. AOP languages (AOPLs) such as AspectJ [4] and AspectC [5] define a new unit of modularization, the *aspect*, which is a combination of *advice* (code that implements a CCC) and *joinpoints* (points in a program's control flow where advice gets executed). A complete aspect-oriented program consists of a core program and aspects, and AOPL compilers typically weave advice from user-defined aspects into the core program at the joinpoints specified by those aspects. Roughly speaking, then, IVCon's unwoven view corresponds to an aspect-oriented view of a program, as code implementing CCCs appear in isolated, aspect-like modules. However, unlike standard AOP tools, IVCon (1) provides both woven and unwoven views of software, (2) allows multi-concern code, (3) enforces token-level granularity in concern code, and (4) applies a novel GUI design to aid concern visualization (e.g., as explained in Section 2.2, when a user clicks on concern code in the unwoven view, IVCon displays the core-program context in which that concern code appears). On the other hand, IVCon is able to provide some of these features only because it disallows joinpoints (called *regions* in IVCon) from being specified indirectly (i.e., as *pointcuts*), which normal AOPLs do allow.

1.1.1 Aspect-visualization Tools

Turning our attention to specific existing AOP tools related to IVCon, AspectBrowser also provides a GUI for viewing CCCs [6, 7]. In AspectBrowser, users specify CCCs in the form of regular expressions (over source-code characters) and assign a color to each CCC. Aspect-

Browser then searches for all instances of the regular expressions in the program and displays a high-level program map that uses concern colors to indicate which lines contain which CCCs. When a program line contains more than one CCC, AspectBrowser colors the corresponding map line red. Also, AspectBrowser users can zoom within the map to obtain a more detailed view and can click on colored lines to view that line's CCC and its context (i.e., the core code surrounding the CCC). Although AspectBrowser allows many-to-many relationships between concerns and code, it lacks support for viewing and editing concern code in isolation. Also, AspectBrowser allows character-level granularity in concern assignments.

The Visualiser [8] is a plugin to the Eclipse [9] platform that helps AspectJ programmers visualize joinpoints in their programs using a high-level program map. After assigning colors to existing aspects, the Visualiser performs static analysis to generate a program map similar to that of AspectBrowser. In the Visualiser map, colored lines represent the locations of joinpoints, and if multiple aspects share a joinpoint then the Visualiser splits that line into the colors of those aspects. For simplicity, IVCon does not build a high-level program map, though it does use colors to indicate which code implements which concerns. Also, although IVCon automatically performs the code modularization that Visualiser users must perform themselves by writing aspects in AspectJ, concerns that are implemented in AspectJ and used with the Visualiser can make use of AspectJ's rich joinpoint language. Due to its reliance on AspectJ, though, the Visualiser inherits AspectJ's limitations of one-to-many relationships between concerns and code (AspectJ does not support multi-concern code) and statement-level granularity in concern assignments.

Aspect-jEdit [10] plugs into the jEdit [11] text editor and, like IVCon, allows users to view and edit concern code in the context of the core program. Also like IVCon, Aspect-jEdit users assign code to concerns by highlighting code and explicitly assigning it to the single concern it implements, and users can assign syntactically different code segments to the same concern. Each aspect (i.e., concern) in Aspect-jEdit is associated with a color, and on assigning code to an aspect, that code's background color changes to match its aspect color. Aspect-jEdit users can hide one or more aspects and view aspects in isolation. Aspect-jEdit implements a one-to-many relationship between concerns and code (it does not support multi-concern code) and assumes line-level granularity in concern assignment. Also, Aspect-jEdit displays syntactically equal advice multiple times in an aspect, as opposed to IVCon's use of subconcerns (described in Section 2.2); consequently, Aspect-jEdit users cannot modify all identical concern code at once in a central concern module (cf. Section 2.2).

1.1.2 Aspect-oriented Programming Languages

Hyper/J is a Java-based AOPL that introduces the concept of a *hyperspace*, an imaginary space consisting of multiple dimensions of concerns [12]. Each dimension, or axis, in the hyperspace groups concerns, while each coordinate on an axis corresponds to a single concern. Hence, a code segment's position in the hyperspace indicates which concerns it implements (one concern per axis in the hyperspace). To build software with Hyper/J, users create a set of text files, and by writing these files appropriately, users can specify the set of features to include in a program. Hyper/J allows users to define a many-to-many relationship between concerns and code; however, concerns in Hyper/J are defined coarsely at the granularity of declarations (e.g., methods, functions, variables, and classes). If all declarations in a program are assigned to at least one concern, then a Hyper/J user can view any concern c in that program in isolation, but doing so involves modifying a textual concern-mapping file to specify that only c should be displayed (i.e., included in the program).

The C4 toolkit provides an aspect-oriented approach to system-level programming [13]. As with IVCon, C4 users can examine programs in, and translate between, two code views. In C4, these views are called the AspectC view (in which users define aspects in AspectC) and the C4 view (in which users view advice inlined into the program code); these are analogous to IVCon's unwoven and woven views. The C4 toolkit does not provide a GUI for visualizing concern code, and because C4 takes advantage of AspectC's aspect-definition language, it inherits AspectC's lack of support for multi-concern code and AspectC's statement-level granularity in concern code.

Finally, we note that although IVCon is related to AOPLs due to its emphasis on modularizing and refactoring concern code, IVCon could not be considered an AOPL tool according to Filman and Friedman's definition of AOPLs [14]. Filman and Friedman specify two necessary properties of AOPLs, *obliviousness* and *quantification*. IVCon's woven view is not oblivious because it requires a programmer to document concerns directly in the body of the code. Also, IVCon does not satisfy the quantification condition because it does not let users define joinpoints (i.e., IVCon *regions*) as conditions on the program's control flow. Allowing IVCon users to define joinpoints as conditions on control flow could lead to ambiguous order of execution when weaving concern code into the core; disambiguating concern-code execution ordering would require some mechanism for specifying concern-code precedence, which would complicate IVCon's design. Nonetheless, IVCon's lack of obliviousness and quantification does not necessarily prevent it from being used as a basis for standard AOP technologies, given that previous work has shown how to provably build an (oblivious and quantified) AOPL on top of an unoblivious and unquantified aspect language [15].

1.1.3 Feature-oriented Programming

CIDE (Colored Integrated Development Environment) [16, 17] is a GUI-based tool for feature-oriented programming that was developed concurrently with IVCon. CIDE has many similarities with both Hyper/J and IVCon: as with Hyper/J, CIDE users can build programs by selecting the sets of features (which are analogous to CCCs in Hyper/J and IVCon) to include in those programs; as with IVCon, CIDE users can highlight code to assign it to features being implemented, can define many-to-many relationships between features and code, can assign colors to features, and can view code colored to reflect the features being implemented. However, there are at least four important, high-level differences between CIDE and IVCon:

1. CIDE displays code assigned to a feature f as black text on the background color of f. For code that implements multiple features, CIDE displays a background color equal to the chromatic blending of the colors of all features being implemented. This design relies on a user's ability to decompose any displayed background color b into the feature colors that combined to produce b, a challenging task when many feature colors exist (some of which may even be similar to combinations of other feature colors). IVCon attempts to avoid this problem by displaying all multi-concern code in a distinctive but uniform manner;

users determine exactly which concerns multi-concern code implements by looking in a concerns-at-current-position panel (as described in Section 2.1).

2. The granularity of feature assignment in CIDE is coarser than the granularity of concern assignment in IVCon. CIDE allows users to assign concerns at the grammatical level of nodes in abstract syntax trees (ASTs), rather than at the lexical level of tokens.

3. Because CIDE requires that any set of features can be composed to create a legal program, CIDE users can only assign code segments to features when those segments are optional according to the language's syntax. IVCon lacks this composeability requirement, so it is much less restrictive; concern code in IVCon must only be lexically valid.

4. By specifying a program to contain exactly one feature, CIDE users can view that one feature isolated from the others (but not isolated from the core program code). However, as with Aspect-jEdit, CIDE displays syntactically equal code implementing the same feature multiple times, as opposed to IVCon's use of subconcerns (cf. Section 2.2). Hence, CIDE users cannot modify all identical feature code at once in a central module.

The most significant of these differences are numbers 3 and 4, which arise out of the subtly different objectives of CIDE and IVCon: CIDE (and related technologies such as Software Plans [18]) focuses on constructing software as a set of features, while IVCon focuses on viewing and modifying CCCs in isolation (as well as in the regular, woven view of the code).

1.2 Contributions

As a concern-visualization and -management tool, IVCon contributes features beyond those of existing tools. Most of the existing research on concern management restricts users to defining one-to-many relationships between concerns and code, while other tools do allow many-to-many relationships but have line-, statement-, or character-level granularity in concern assignment and lack IVCon's support for conveniently modularizing and isolating concern code. Hence, this paper contributes a tool design and implementation that does all of the following:

1. Allows many-to-many relationships between concerns and code

2. Enforces token-level granularity in concern assignment
3. Enables programmers to conveniently translate between woven and unwoven code views
4. Provides a GUI for visualizing and modularizing concerns in software

By providing all these features, IVCon users can flexibly create, examine, and update code in, and translate between, woven and unwoven views of software.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 describes IVCon's GUI, Chapter 3 discusses our implementation, Chapter 4 evaluates the performance of our implementation, and Chapter 5 concludes.

CHAPTER 2

USER INTERFACE

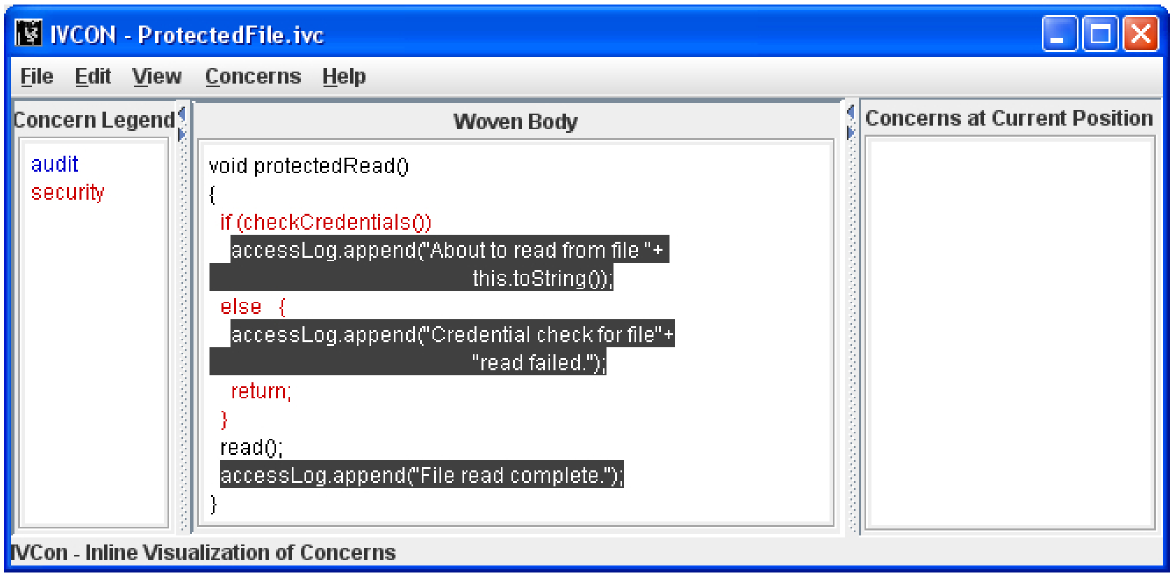
IVCon displays code in two different but equivalent forms: the woven view (Figure 2.1) and the unwoven view (Figure 2.2). Users can translate their code between the two views simply by selecting the `weave` or `unweave` menu options, or by pressing <ctrl-w> or <ctrl-u>.

2.1 Woven View

In the woven view, shown in Figure 2.1, users can write code as they normally would in a standard text editor or development environment. In addition, users can define concerns, associate a color with each concern, and highlight and explicitly assign code segments to concerns (by right-clicking highlighted code and selecting the concern to which to assign it). Upon assigning a code segment to a concern, IVCon recolors the assigned code to match the concern it implements. IVCon displays code assigned to multiple concerns in white text on a dark background, though users are free to change this *multi-concern background* color.

By highlighting a contiguous code segment and assigning it to a concern, a user defines a *region* in the code. Every region starts at the beginning of code assigned to a concern and extends as far as the code does that implements that concern. Multiple concerns can share the same region if code implementing those concerns begins and ends at exactly the same positions in the program file. Although IVCon requires a user-specified name for every unique region a user defines, it always provides a default name for regions (based on the name of the concern to which the region is being assigned). If a previously defined region gets assigned to a new concern, IVCon simply reuses the existing region name. Specifying names for regions helps users understand where subconcerns are implemented, as discussed in Section 2.2.

Figure 2.1 shows the woven-view window divided into three panels: *concerns legend*, *woven body*, and *concerns at current position*.


IVCON - ProtectedFile.ivc
File Edit View Concerns Help
Concern Legend
audit
security
Woven Body
Concerns at Current Position
IVCon - Inline Visualization of Concerns

```
void protectedRead()
{
 if (checkCredentials())
   accessLog.append("About to read from file "+
                              this.toString());
 else  {
   accessLog.append("Credential check for file"+
                              "read failed.");
   return;
 }
 read();
 accessLog.append("File read complete.");
}
```

Figure 2.1. IVCon's woven view of the same code shown in Figure 2.2

1. The concerns-legend panel lists all the user-defined concerns in the current file. IVCon displays the name of each concern in the color associated with that concern.

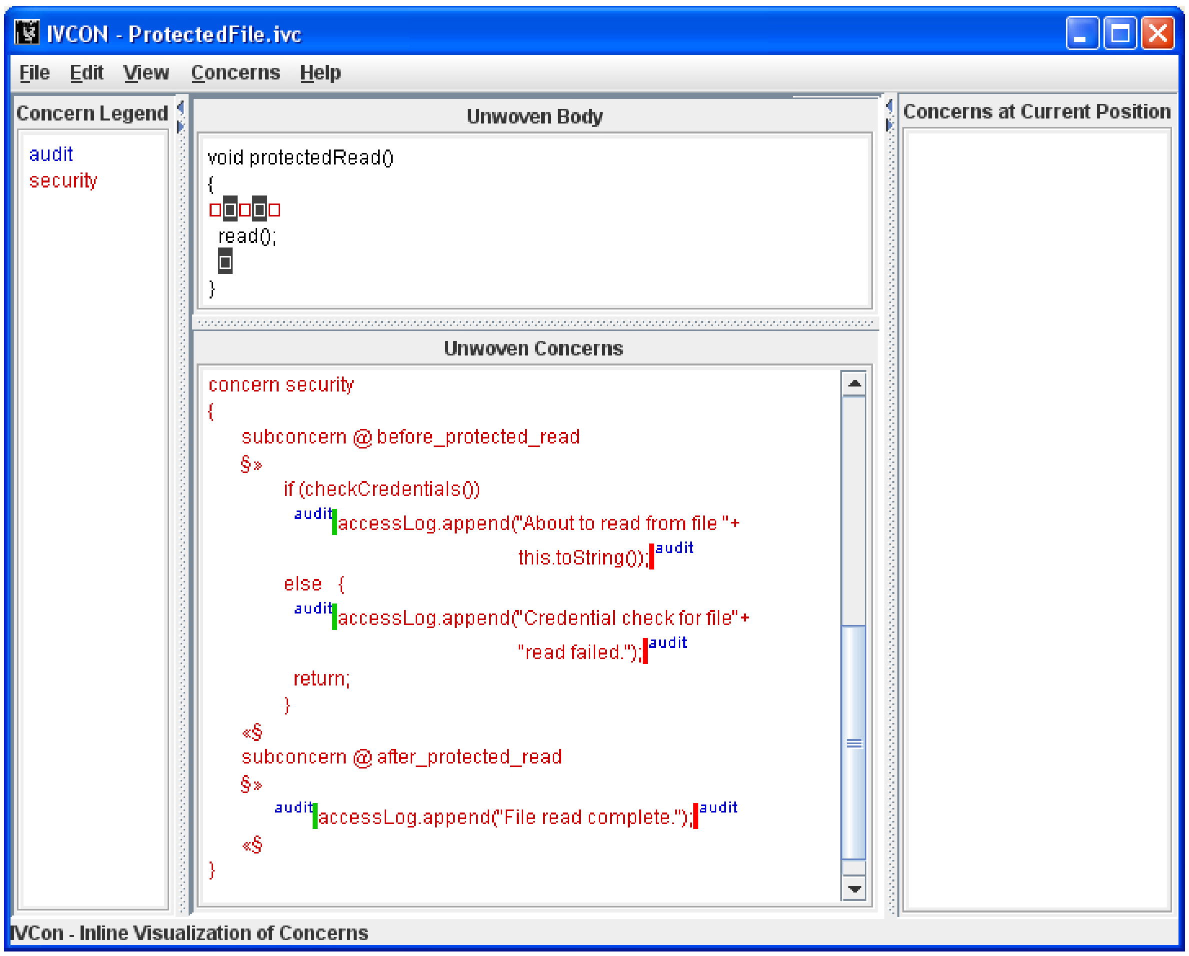
2. The woven-body panel contains user code displayed in colors that indicate concern assignments.

3. The concerns-at-current-position panel lists the concern(s) implemented by the code at the current cursor position.

Apart from creating and modifying code, defining concerns, and assigning code to concerns, users can edit concern names, edit concern colors, remove concerns, de-assign code from concerns, change the multi-concern background, rename regions, and open, save, and close files in the woven view.

2.2 Unwoven View

The unwoven view, shown in Figure 2.2, displays the program core and each concern in isolation. The unwoven-view window is the same as the woven-view window, except that the unwoven-view divides the woven-body panel into two subpanels: *unwoven body* and *unwoven concerns.*


IVCON - ProtectedFile.ivc
File Edit View Concerns Help
Concern Legend
audit
security
Unwoven Body
Unwoven Concerns
Concerns at Current Position
IVCon - Inline Visualization of Concerns

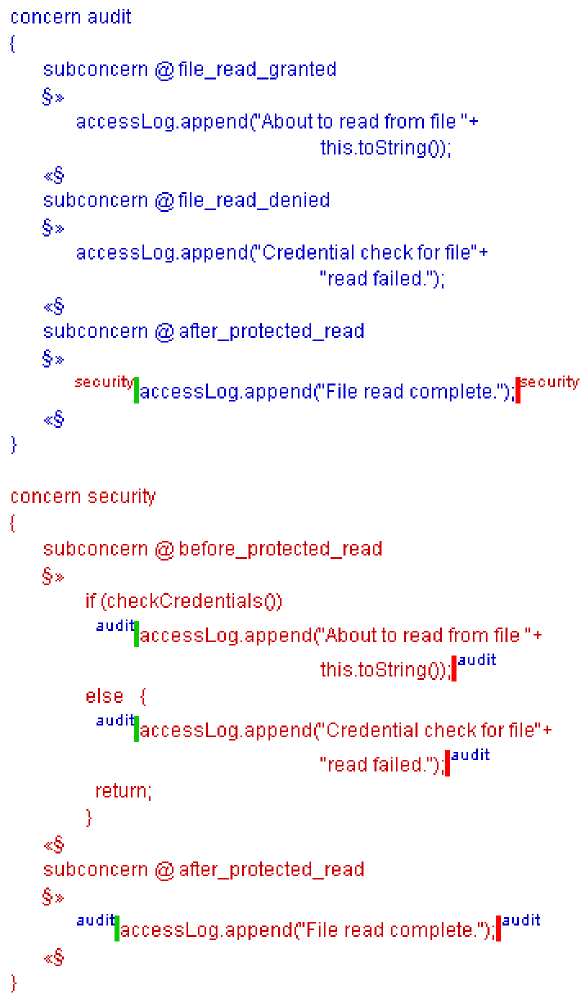
```
void protectedRead()
{
□■□■□
 read();
 ■
}
```

```
concern security
{
    subconcern @ before_protected_read
    §»
          if (checkCredentials())
            audit accessLog.append("About to read from file "+
                                    this.toString()); audit
          else  {
            audit accessLog.append("Credential check for file"+
                                    "read failed."); audit
            return;
          }
    «§
    subconcern @ after_protected_read
    §»
         audit accessLog.append("File read complete."); audit
    «§
}
```

Figure 2.2. IVCon’s unwoven view of the same code shown in Figure 2.1

1. The unwoven-body panel displays the core of the program. Code that has been assigned to one or more concerns is extracted (into the unwoven-concerns panel) and replaced by holes (□) of the same color as the extracted code. Thus, holes indicate extracted concern code.

2. The unwoven-concerns panel displays in isolation each of the program’s concerns (as extracted from the unwoven body). Each concern is divided into *subconcerns*, which are syntactically different code segments assigned to the same concern. IVCon displays subconcerns in two parts: a list of the regions in which the subconcern appears and then the subconcern code itself. For example, the unwoven-concerns panel in Figure 2.2 displays the `security` and `audit` concerns in the format shown in Figure 2.3. On clicking any



```
concern audit
{
     subconcern @ file_read_granted
     §»
          accessLog.append("About to read from file "+
                                        this.toString());
     «§
     subconcern @ file_read_denied
     §»
          accessLog.append("Credential check for file"+
                                        "read failed.");
     «§
     subconcern @ after_protected_read
     §»
          security|accessLog.append("File read complete.");|security
     «§
}

concern security
{
     subconcern @ before_protected_read
     §»
           if (checkCredentials())
            audit|accessLog.append("About to read from file "+
                                        this.toString());|audit
           else  {
            audit|accessLog.append("Credential check for file"+
                                        "read failed.");|audit
            return;
           }
     «§
     subconcern @ after_protected_read
     §»
          audit|accessLog.append("File read complete.");|audit
     «§
}
```

Figure 2.3. Concern-module formatting in IVCon

region name in the unwoven-concerns panel, IVCon automatically focuses the unwoven-body panel to show that region's location in the context of the program core.

Code for the `security` concern in Figure 2.3 indicates the presence of two subconcerns (at regions `before_protected_read` and `after_protected_read`). The code segments beginning with `if (checkCredentials())` and `accessLog.append` implement those subconcerns. Similarly, code for the `audit` concern indicates the presence of three subconcerns (at regions `file_read_granted`, `file_read_denied`, and `after_protected_read`). The unwoven-concerns panel may also contain constructs called *flags* (e.g., $^{\text{security}}\rule{0.3em}{1em}$ and $\rule{0.3em}{1em}^{\text{security}}$), which convey information about concern assignment in multi-concern code segments. Section 2.3 provides additional explanation of IVCon flags.

Unwoven Body:

```
if (buffer.getSize() > □)
   buffer.truncate(□);
if (getTimeElapsed() > □)
   JOptionPane.showMessageDialog(frame,
           "Request timed out","Error",
           JOptionPane.ERROR_MESSAGE);
```

Unwoven Concerns:

```
concern constant
{
  subconcern @ max_buffer_size_0
             @ max_buffer_size_1
  §≫
    512
  ≪§
  subconcern @ timeout_ms_0
  §≫
    2000
  ≪§
}
```

Figure 2.4. Unwoven view of the code in Figure 2.5

Figure 2.3 also demonstrates the usefulness of having descriptive, user-specified names for regions. Descriptive region names help software engineers quickly understand where subconcern code exists in relation to the rest of the program logic. Nonetheless, if a region name provides insufficient contextual information, the user can always click on the name to see that region's context in the unwoven-body panel.

As another example, consider Figure 2.4, which shows how IVCon groups into one subconcern all syntactically equal code assigned to the same concern. Figure 2.5 contains the woven view of the same program. In the woven view, the user has defined a concern named `constant` and has assigned the two constants 512 and 2000 to that concern. Normally, programmers using standard software-development tools would define these values in memory declared immutable and would always refer to the constants' values with variables like `MAXBUFFERSIZE` and `TIMEOUTMS`. This technique enables the programmers to make a global change to a constant value by modifying just one central definition. However, this benefit comes at the price of not

Woven Body:

```
if (buffer.getSize() > 512)
   buffer.truncate(512);
if (getTimeElapsed() > 2000)
   JOptionPane.showMessageDialog(frame,
           "Request timed out","Error",
           JOptionPane.ERROR_MESSAGE);
```

Figure 2.5. Woven view of the code in Figure 2.4

being able to immediately see the values of constants in the source code. In contrast, IVCon's dual woven and unwoven views provide *both* benefits: users can update constant values centrally (in the `constant` concern of the unwoven-concerns panel) and can view the constant values directly in the source code (in the woven-body panel). Figures 2.4 and 2.5 also demonstrate the usefulness of token-level granularity in concern assignments.

IVCon's unwoven view allows users to create and edit core-program code (in the unwoven-body panel) and concern code (in the unwoven-concerns panel). To avoid ambiguity in the weaving algorithm (described in Section 3.2.2), IVCon does not allow users to delete holes in the unwoven-body panel or to edit non-concern code (e.g., concern and region names) in the unwoven-concerns panel.

2.3 Display of Multi-concern Code

The woven view displays multi-concern code in white text over the multi-concern background, while the concerns-at-current-position panel indicates which concern(s) the code at the current cursor position implements. Similarly, the unwoven view displays multi-concern code in the unwoven-body panel as a hole colored white over the multi-concern background, while the concerns-at-current-position panel continues to indicate which concern(s) the hole at the current cursor position implements.

In addition, the unwoven-concerns panel uses flags to convey information about the concerns associated with multi-concern code (as mentioned in Section 2.2). Flags serve as a quick reference for visualizing where overlapping concerns begin and end. To illustrate the use of flags, consider the code in Figure 2.2 that implements the `security` concern at region `before_protected_read`. In the woven view (Figure 2.1), we assigned this code segment to the `security` concern, and we assigned the two nested statements beginning with `accessLog.append` to the `audit` concern. As a result, the unwoven-concerns panel in Figure 2.2 displays green flags (${}^{\text{audit}}$▌) and red flags(▌${}^{\text{audit}}$) to indicate the nesting of `audit`-concern code within the `security`-concern code.

Green and red flags within the unwoven-concerns panel indicate the beginning and ending of overlapping concerns. Green and red flags do not always appear together within a subconcern; depending on the overlap between concern regions, there may be a green flag only, a red flag only, both green and red flags, or no flags at all within subconcern code. Also, multiple red and green flags of the same concern, or multiple flags of various concerns, may be present within a subconcern.

The syntax of code in IVCon's unwoven-concerns panel includes flags, so two subconcerns are syntactically equal if and only if the text of those two subconcerns—including flags within the subconcerns—is the same. Thus, the subconcern `accessLog.append(‘‘File read complete.’’)` would not be syntactically equal to ${}^{\text{security}}$▌`accessLog.append(‘‘File read complete.’’)`▌${}^{\text{security}}$. This distinction matters because, as described in Section 2.2, syntactically equal subconcerns are grouped together in the unwoven-concerns panel.

CHAPTER 3

IMPLEMENTATION

We have implemented IVCon on a Java platform and made its source code publicly available [19]. The core IVCon application consists of 6235 lines of code, of which 6107 implement the GUI and 128 implement backend data structures. IVCon also uses several third-party libraries (e.g., clipboard and lexical-analysis libraries) for auxiliary functions.

3.1 Data Structures

IVCon maintains three key data structures.

1. A *regionMap* is a hash table that maps a numerical region identifier (needed for the RTree implementation described below) to that region's user-visible name, beginning and ending positions for the region, and a list of the concerns to which the region has been assigned.

2. A *concernMap* is a hash table that maps a unique concern name to that concern's display color and a list of the regions assigned to that concern.

3. A *regionTree* is an RTree, a dynamic structure for storing data about potentially overlapping regions in space [20, 21]. When queried about a particular region r, an RTree can efficiently return the set of stored regions that overlap r. RTrees are ideal data structures for IVCon because they enable efficient querying to determine which regions are defined at a given character position in the source file (e.g., at the position of the cursor or within a newly defined region). Once IVCon determines which regions are present at a given position, it can use the `regionMap` to look up and display all the concerns assigned to those regions. IVCon uses Hadjieleftheriou's implementation of RTrees [22].

Together these structures enable reasonable performance in all of IVCon's core operations.

We researched other spatial data structures such as the P-R Tree [21], IBS Tree [23] and skip lists [24]. However, we had a problem finding a correct implementation of the P-R Tree; IBS Tree and skip lists were not suited to our purpose because they do not allow dynamic addition or deletion of intervals, which is important for IVCon as users define regions dynamically.

3.2 Translation Algorithms

Given a `regionMap`, `concernMap`, and `regionTree`, it is generally straightforward to implement the translation from woven to unwoven view, and vice versa.

3.2.1 Unweaving

Unweaving is the process of translating a woven-view program into an equivalent unwoven-view program. Unweaving in IVCon begins with lexical analysis to (1) ensure that all tokens in the current program are valid Java tokens and (2) enforce token-level granularity in concern assignment (by requiring that all concern regions begin and end at beginnings and endings of Java tokens). After lexical analysis, IVCon starts with the woven code in the unwoven-body panel and iterates through all concerns in the `concernMap`, extracting that concern's code regions one at a time from the unwoven-body panel to the unwoven-concerns panel. Extracted concern code gets replaced with holes (□) of the appropriate color. During the extraction process, IVCon groups concern code into syntactically equal subconcerns and displays isolated concerns in the format described in Section 2.2.

Although the unweaving algorithm just described is straightforward, one interesting issue arose during implementation. The issue concerns how to display holes in place of overlapping, but unequal, regions. For example, let us return to the woven-body code of Figure 2.1. In that figure, a programmer has assigned an entire `if` statement to the `security` concern and has nested two `audit`-concern regions within that `security` region. There are two reasonable alternatives for unweaving this `if` statement:

1. We could replace the entire `if` statement with *one* hole of the multi-concern color to indicate that one region of code, which in total implements multiple concerns, has been extracted.

2. We could replace the entire `if` statement with *five* holes that alternate between the security-concern color and the multi-concern color. These holes would indicate that the extracted code first contains security-concern code, then some multi-concern code, then more security-concern code, and so on.

Because it provides more precise concern-assignment information, we implemented the second of these alternatives in IVCon. Figure 2.2 shows the resulting five-hole concern extraction in the unwoven-body panel.

3.2.2 Weaving

Weaving is the process of translating an unwoven-view program into an equivalent woven-view program. To weave, IVCon builds the woven body from the unwoven body by iterating through all the holes in the unwoven body and filling in each hole with the appropriate concern code.

More specifically, for every hole in the unwoven body, IVCon uses the `regionTree` and `regionMap` to look up the regions and concerns associated with that hole. In the simplest case, only one concern is associated with the hole, so IVCon can immediately fill that hole with the unwoven-concerns-panel code that implements the hole's concern at the hole's region. Otherwise (that is, if more than one concern is associated with the hole), the situation is more complex because the code segment that fills in the hole has been assigned to multiple concerns, so that code segment appears in multiple places in the unwoven-concerns panel. For example, the code that fills in every multi-concern hole in Figure 2.2 exists in two places in the unwoven-concerns panel of Figure 2.3. If all the code segments that fill in the same multi-concern hole are identical, IVCon can simply fill in the hole with that code segment. However, the code segments filling in the same hole may not be identical because a user may have modified code in the unwoven-concerns panel (e.g., a user may have changed the `log.append`s to `log.prepend`s in the `security` concern of Figure 2.3 without modifying the `log.append`s in the `audit` concern). In this case, IVCon cannot immediately determine with what code to fill the hole, so it opens a popup window to (1) notify the user of the concern-code inconsistency and (2) allow the user to select which code segment to fill the hole with (e.g., to use `log.prepend` or `log.append`).

After IVCon fills in all the holes, it has a complete woven view of the program and can perform lexical analysis. Like the lexical analysis at the beginning of the unweaving process, lexical analysis at the conclusion of the weaving process ensures that all tokens in the current program are valid Java tokens and enforces token-level granularity in concern assignment. Although for simplicity we have not done so, it would also be useful to integrate a complete Java compiler into IVCon to check, beyond simple lexical analysis, that woven programs are statically valid Java programs. Integrating a complete compiler and virtual machine into IVCon would also save programmers from having to switch to external tools for code compilation and execution.

CHAPTER 4

PERFORMANCE EVALUATION

To evaluate the practicality of our design, we tested[1] IVCon by assigning code to concerns in three of our own IVCon source-code files: `IVCON.java`, `FileUtilities.java`, and `ConcernManipulation.java` (our largest source-code file), respectively containing 49, 313, and 3342 lines of Java code (in the woven view) and 7, 25, and 182 regions assigned to 5, 11, and 36 total concerns. We also created an impractically large file of 100,000 lines, each containing 20 randomly generated single-character tokens, in order to stress test IVCon's performance. Table 4.1 summarizes our test-file characteristics. We emphasize that `StressTest.java` would be an unreasonably large (2-million-token) source-code file in practice; we included it in our test suite to better understand IVCon's performance limitations.

IVCon allows users to open and save Java source-code (`.java`) files and IVCon (`.ivc`) files. IVCon files contain several serialized objects: the Java source-code string in the woven view, plus IVCon's `regionMap`, `concernMap`, and `regionTree` for that program. We measured IVCon's performance opening and saving our test files as `.java` files, `.ivc` files with no concerns

Table 4.1. Test-file characteristics

File Name	File Size (LoC)	Total # of Concerns	Total # of Regions	Avg. Region Size (chars)	Max. Region Size (chars)
IVCON.java	49	5	7	135.9	337
FileUtilities.java	313	11	25	155.7	788
ConcernManipulation.java	3342	36	182	269.9	3461
StressTest.java	100,000	1000	5000	1016.0	1996

[1]The tests were performed on a Dell Latitude D830 with dual Intel Core2 2.2 GHz CPUs and 2 GB of RAM, running Windows XP. The times represent real time at low average load. We performed each test in sets of 100; the results shown are the averages of those sets. During stress testing, we had to increase the virtual machine's heap size to 1.5 GB.

Table 4.2. Performance opening files

File Size (LoC)	Type of File	File-open Time (ms)
49	.java file	58.92
	.ivc file (no concerns)	33.91
	.ivc file (5 concerns)	39.52
313	.java file	167.3
	.ivc file (no concerns)	89.37
	.ivc file (11 concerns)	115.8
3342	.java file	937.2
	.ivc file (no concerns)	584.5
	.ivc file (36 concerns)	999.5
100,000	.java file	112,722
	.ivc file (no concerns)	24,117
	.ivc file (1000 concerns)	26,820

Table 4.3. Performance saving files

File Size (LoC)	Type of File	File-save Time (ms)
49	.java file	1.730
	.ivc file (no concerns)	38.09
	.ivc file (5 concerns)	42.55
313	.java file	4.040
	.ivc file (no concerns)	106.7
	.ivc file (11 concerns)	139.5
3342	.java file	10.83
	.ivc file (no concerns)	659.9
	.ivc file (36 concerns)	1282
100,000	.java file	176.6
	.ivc file (no concerns)	2952
	.ivc file (1000 concerns)	3811

defined, and `.ivc` files with all concerns defined. Tables 4.2 and 4.3 display the results. IVCon opens `.ivc` files more quickly than `.java` files because the contents of `.ivc` files, being serialized, can be efficiently input as Java objects, while the contents of `.java` files are input line by line and concatenated to form the overall woven body. Conversely, IVCon saves `.java` files more

Table 4.4. Performance weaving code

Lines of Woven Code	Number of Concerns	Holes in Unwoven Body	Weaving Time (ms)
49	0	0	1.720
	5	7	20.70
313	0	0	12.52
	11	26	88.71
3342	0	0	94.40
	18	129	237.6
	36	205	566.8
100,000	0	0	9823
	1000	7282	465,000

Table 4.5. Performance unweaving code

Lines of Woven Code	Number of Concerns	Holes in Unwoven Body	Unweaving Time (ms)
49	0	0	3.130
	5	7	4.370
295	0	0	8.980
	11	26	21.70
3476	0	0	85.80
	18	129	253.2
	36	205	501.9
100,000	0	0	8836
	1000	7282	481,737

efficiently than `.ivc` files because `.ivc` files contain several potentially large, serialized data structures; as expected, IVCon's file-save times are proportional to the length of the Java-code output and the size of the objects being serialized. The file-save times in Table 4.3 do not include the time taken to close the files, but file-closing times were negligible anyway (i.e., unobservably small for all but the `StressTest` files, which took about 83ms to close).

Although creating a new concern in IVCon takes only a constant amount of time and is a fast (0ms to 16ms) operation, we next describe IVCon's performance during three heavier-weight concern-manipulation operations. Tables 4.6, 4.7, and 4.8 show IVCon's performance

Table 4.6. Performance assigning code to a concern

File Size (LoC)	Region Size (LoC)	Number of Nested Regions	Concern-assignment Time (ms)
49	49	7	17.35
313	295	25	50.58
3476	100	1	4.210
	500	33	83.89
	1500	73	192.7
	3342	182	519.1
100,000	1000	44	3373
	10,000	473	35,183
	100,000	5000	312,519

Table 4.7. Performance editing a concern color

File Size (LoC)	Number of Characters Assigned to Concern	Concern-edit Time (ms)
49	332	5.310
313	647	14.49
3342	988	21.09
	2154	27.52
	6217	30.00
100,000	13,876	582.9
	15,170	1039
	50,212	2276

during concern assignment, editing, and removal. IVCon's performance when assigning code to a concern (Table 4.6) depends on the size of the region r being assigned to the concern and the number of regions nested within r; higher values for these two parameters imply more considerations of code-color changes and therefore more time to complete the concern-assignment operation. Because editing concern names takes a negligible amount of time (0ms to 16ms), the biggest factor in editing a concern is actually changing its color (Table 4.7), which again depends on the amount of text (i.e., the number and size of regions) assigned to the concern being recolored. Similarly, when a user completely removes a concern in the woven view, most of the time IVCon spends completing this operation involves recoloring the code

Table 4.8. Performance removing a concern

File Size (LoC)	Number of Characters Assigned to Concern	Concern-removal Time (ms)
49	332	7.020
313	647	20.58
3342	988	34.01
	2154	38.31
	6217	84.02
100,000	13,876	909.5
	15,170	917.5
	50,212	3742

that had been assigned to that concern; hence, concerns assigned to more and larger regions take more time to remove than concerns assigned to fewer and smaller regions (Table 4.8).

Finally, we measured IVCon's performance weaving and unweaving code, as presented in Tables 4.4 and 4.5. Based on the descriptions of these algorithms in Section 3.2, we would expect that the biggest factors determining weaving and unweaving times are the number of concerns being woven or unwoven, the number of holes being filled in or created, and the number of characters filling in or being extracted from holes. The results in Tables 4.4 and 4.5 match these expectations. Completely weaving and unweaving code in all the reasonable-sized test files took less than one second.

Taken together, our performance results demonstrate that IVCon's design is sufficiently efficient when operating on reasonably sized source-code files. However, IVCon would need additional optimizations to perform tolerably efficiently on extremely large source-code files, such as those containing millions of source-language tokens.

CHAPTER 5

CONCLUSIONS

We conclude by summarizing and describing future extensions to this work.

5.1 Summary

We have presented IVCon, a GUI-based tool for conveniently creating, examining, and modifying code in, and translating between, woven and unwoven views of code. IVCon differs from existing aspect-visualization tools by providing a combination of (1) translations between woven and unwoven views (2) many-to-many relationships between concerns and code, (3) token-level granularity in concern assignment, and (4) a GUI designed to make all of this convenient. We have described IVCon's interface and implementation details and have made its source code is publicly available [19]. In all of our empirical tests, IVCon performed tolerably well on reasonably sized source-code files but could benefit from optimizations to improve efficiency when manipulating extremely large files.

5.2 Future Work

Several opportunities exist for improving IVCon. For one, we would like to collect data from real users to measure IVCon's effectiveness at improving program comprehension. Moreover, we would like to continue evaluating IVCon's empirical performance to determine whether and how the weaving and unweaving operations could be optimized, especially for larger source-code files.

To aid users in building projects that span over several files, we would also like to extend IVCon to handle multiple source-code files simultaneously, for example, to allow all files in a project to share the same set of concerns.

In addition, to provide the same clarity of multi-concern code in the woven view that currently exists in the unwoven view, a future implementation of IVCon could allow users to view concern flags in the woven body; users would be able to toggle between viewing and hiding flags in the woven view.

The concerns legend in the current implementation of IVCon is a textual list of concerns. In a future implementation, we could embed navigational aids into this list. For example, each entry in the concern legend could be accompanied by two buttons, "previous" and "next", that could refocus the woven-body panel to the previous or next instance of code implementing the selected concern.

Finally, we would like to model our IVCon language in a formal calculus and prove that (1) our translation algorithms are inverses of each other, (2) our translations are semantics preserving (i.e., translation *soundness*), and (3) we can unweave any concern in any woven program and weave any concern into any unwoven program (i.e., translation *completeness*).

REFERENCES

[1] Nalin Saigal and Jay Ligatti. Defining and visualizing many-to-many relationships between concerns and code. Technical Report CSE-090608-SE, University of South Florida, September 2008.

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

[3] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

[4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, 2001.

[5] Monica Yvonne Coady. *Improving evolvability of operating systems with AspectC*. PhD thesis, The University of British Columbia, 2003.

[6] W. G. Griswold, Y. Kato, and J. J. Yuan. AspectBrowser: Tool support for managing dispersed aspects. Technical Report CS1999-0640, University of California at San Diego, La Jolla, CA, USA, 1999.

[7] Macneil Shonle, Jonathan Neddenriep, and William Griswold. AspectBrowser for eclipse: A case study in plug-in retargeting. In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 78–82, 2004.

[8] The Visualiser, 2008. `http://www.eclipse.org/ajdt/visualiser/`.

[9] Eclipse, 2008. `http://www.eclipse.org/`.

[10] T. Panas, J. Karlsson, and M. Högberg. Aspect-jEdit for inline aspect support. In *Proceedings of the Third German Workshop on Aspect Oriented Software Development*, 2003.

[11] jEdit, 2008. `http://www.jedit.org/`.

[12] Harold Ossher and Peri Tarr. Hyper/J: Multi-dimensional separation of concerns for Java. In *Proceedings of the International Conference on Software Engineering*, pages 734–737, 2000.

[13] Marco Yuen, Marc E. Fiuczynski, Robert Grimm, Yvonne Coady, and David Walker. Making extensibility of system software practical with the C4 toolkit. In *Proceedings of the Workshop on Software Engineering Properties of Languages and Aspect Technologies*, March 2006.

[14] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical Report 01.12, RIACS, 2000.

[15] Jay Ligatti, David Walker, and Steve Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming: Special Issue on Foundations of Aspect-Oriented Programming*, 63(3):240–266, December 2006.

[16] Christian Kästner. CIDE: Decomposing legacy applications into features. In *Proceedings of the 11th International Software Product Line Conference (SPLC), second volume (Demonstration)*, pages 149–150, 2007.

[17] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 311–320, May 2008.

[18] Robert R. Painter and David Coppit. A model for software plans. In *Proceedings of the 2005 Workshop on Modeling and Analysis of Concerns in Software*, pages 1–5, 2005.

[19] Nalin Saigal and Jay Ligatti. IVCon – Inline Visualization of Concerns, 2008. `http://www.cse.usf.edu/~ligatti/projects/ivcon/`.

[20] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[21] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Trans. Algorithms*, 4(1):1–30, 2008.

[22] Marios Hadjieleftheriou. Spatial index library. `http://www.research.att.com/~marioh/spatialindex/index.html`.

[23] E. Hanson and M. Chaabouni. The IBS tree: a data structure for finding all intervals that overlap a point. Technical Report WSU-CS-90-11, Wright State University, 1990.

[24] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.