IVCon: Inline Visualization of Concerns

Nalin Saigal and Jay Ligatti
Department of Computer Science and Engineering
University of South Florida
{nsaigal, ligatti}@cse.usf.edu

Technical Report CSE-110909-SE

Abstract

Code modularization provides benefits throughout the software life cycle; however, the presence of crosscutting concerns (CCCs) in software hinders its complete modularization. This paper describes IVCon, a tool with a novel approach for completely modularizing 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 provides an interface for conveniently creating, examining, and modifying code in, and translating between, the woven and unwoven views.

1 Introduction

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" [TOSH99]. 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 generally more difficult than updating an isolated security module.

This paper describes IVCon (Inline Visualization of Concerns), a 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*, while 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, a prototype version of 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 and can assign a single code segment to multiple, overlapping 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 following code:

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

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. At the same time, token-level granularity prevents unreasonable concern assignments possible with finer (e.g., character-level) granularities, such as assigning just the '`i`' in `JOptionPane` to its own concern; this would be unreasonable because if '`i`' implements a concern C, then the rest of the `JOptionPane` token must implement C as well (the `JOptionPane` token has a semantics in the programming language, while the '`i`' alone does not). As far as we are aware, IVCon is the first concern-manipulation tool with token-level granularity in concern assignments.

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

There are many projects related to IVCon, which we next discuss in detail. Readers may wish to skip this subsection on a first reading and instead consult Figure 1 for a summary of IVCon's relationships with related work.

A closely related body of research is Aspect Oriented Programming (AOP) [KLM$^+$97]; like IVCon, AOP eases the specification and manipulation of CCCs in software. AOP languages (AOPLs) such as AspectJ [KHH$^+$01] and AspectC [Coa03] 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. 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 AOPLs typically do allow.

Turning our attention to specific AOP tools related to IVCon, Aspect-jEdit [PKH03] plugs into the jEdit [jEd08] 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 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, but Aspect-jEdit does not support multi-concern code (which simplifies its interface and text-manipulation algorithms). 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).

Concern-management tools such as AspectBrowser [GKY99], FEAT [RM02], and SoQueT [MMvD07] enable users to specify concerns in the form of regular expressions over characters in the source code. Among these tools, AspectBrowser relates most closely to IVCon due to its use of colors to represent CCCs. AspectBrowser displays a high-level program map in which colors 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 a colored line to view that line's CCC and its context (i.e., the core code surrounding the CCC). IVCon does not build a high-level program map, though it does graphically identify all code regions implementing each CCC (using colors, flags, tooltips, and a concern-at-current-position panel in the woven view and colors, flags, and explicit region names in the unwoven view).

The Visualiser [Vis08] is a plugin to the Eclipse [ecl08] 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. 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.

The C4 toolkit provides an aspect-oriented approach to system-level programming [YFG+06]. 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. Similar to the Visualiser, C4 inherits AspectC's lack of support for multi-concern code and its statement-level granularity in concern code.

Hyper/J is a Java-based AOPL that introduces the concept of a hyperspace, an imaginary space consisting of multiple dimensions of concerns [OT00]. 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. 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

Tool	Can edit code in dual views	Isolates concerns	Can modify identical concern code in one place	Allows many-to-many relationships between concerns and code	Provides GUI	Can build programs by composing concern code (features)	Region (i.e., joinpoint) specification	Granularity of concern assignment
Aspect-jEdit	√	√	-	-	√	-	Explicit	Line-level
AspectBrowser	-	-	-	√	√	-	Regular expressions over concern code	Character-level
FEAT	-	-	-	√	√	-	Regular expressions over concern code	Character-level
SoQueT	-	-	-	√	√	-	Regular expressions over concern code	Character-level
Visualiser	-	-	-	-	√	-	AspectJ pointcut language	Line-level
C4	√	√	-	-	-	-	AspectC pointcut language	Line-level
Hyper/J	-	√	-	√	-	√	Package, class, interface, method, or field names	Declaration-level
CIDE	-	-	-	√	√	√	Explicit	Nodes in ASTs
IVCon	√	√	√	√	√	-	Explicit	Token-level

Figure 1: Comparison of concern-manipulation tools

displayed (i.e., included in the program).

CIDE (Colored Integrated Development Environment) [KÖ7, KAK08] is a 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 either a separate panel or a tooltip.

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 and is therefore 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, so CIDE users cannot modify all identical feature code at once in a central module.

Most of these differences arise out of the subtly different objectives of CIDE and IVCon: CIDE (and related technologies such as Software Plans [PC05]) focuses on constructing software as a set of features, while IVCon focuses on creating, viewing, and modifying CCCs in isolation (as well as in the regular, woven view of the code).

Finally, we note that although IVCon is closely 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 [FF00]. 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 [LWZ06].

1.2 Contributions

IVCon contributes features beyond those of existing concern-management tools, as summarized in Figure 1. IVCon does all of the following[1]:

- Enables programmers to conveniently translate between, and edit code in, woven and unwoven code views
- Isolates concern code in the unwoven view
- Enables users to modify all identical concern code in one place in the unwoven view
- Allows many-to-many relationships between concerns and code
- Provides a novel GUI for conveniently managing CCCs
- Enforces token-level granularity in concern assignment

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

This technical report derives from an earlier conference paper [SL09] but extends that work in the following ways:

- We provide much greater detail when comparing IVCon with related work in Section 1.1.
- We discuss a new case study in Section 4.
- We present results from, and an analysis of, IVCon's performance tests in Section 5.
- We revise text and figures in several places to match IVCon's current design.

Roadmap The rest of this paper is organized as follows. Section 2 describes IVCon's interface, Section 3 discusses our prototype implementation, Section 4 describes a case study of IVCon, Section 5 evaluates the performance of our implementation, and Section 6 summarizes.

2 User Interface

IVCon displays code in two different but equivalent forms: the woven view (Figure 2) and the unwoven view (Figure 3). 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, 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. Users can still edit all code after defining concerns; newly typed code gets assigned to the concern(s) present at the position where the user started typing.

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

[1]The only two major features listed in Figure 1 that IVCon does not provide are the abilities to specify programs as a composition of features and to specify joinpoints as conditions on the control flow of programs. As described in Section 1.1, IVCon lacks these features because its objectives differ from those of feature- and aspect-oriented programming tools.

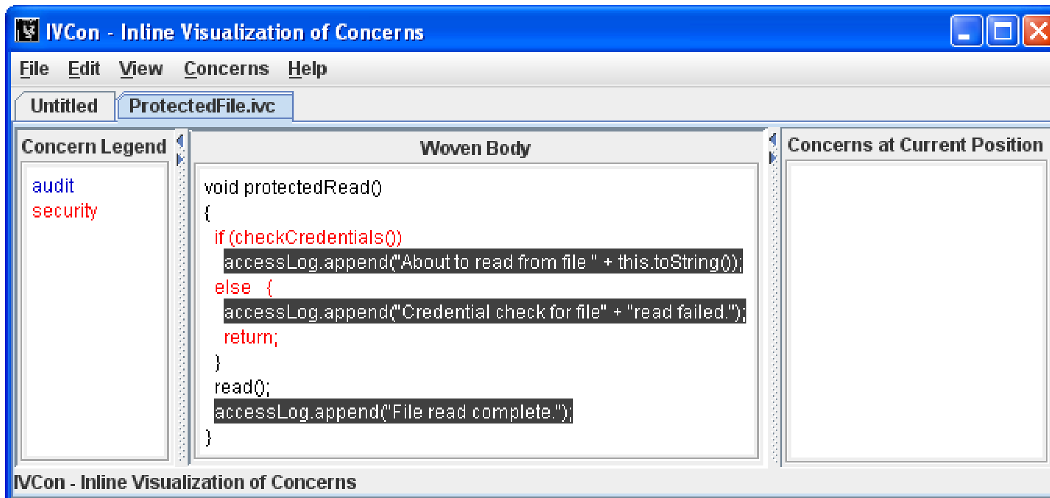

Figure 2: IVCon's woven view of the same code shown in Figure 3

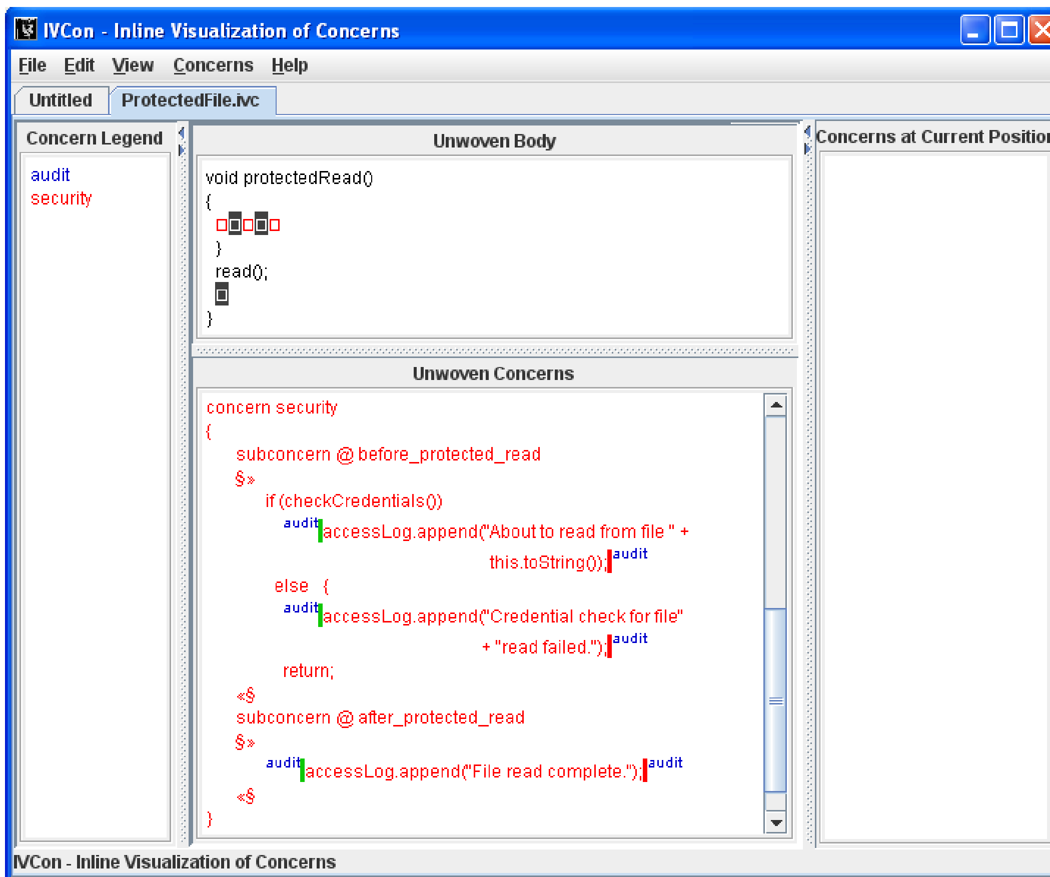

Figure 3: IVCon's unwoven view of the same code shown in Figure 2

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 shows the woven-view window divided into three panels:

- 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.
- The woven-body panel contains user code displayed in colors that indicate concern assignments.
- 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 3, 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:

- 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 a hole (□) of the same color as the extracted code. Thus, holes indicate extracted concern code.
- 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 3 displays the `security` and `audit` concerns in the format shown in Figure 4. On clicking any 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 4 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., security▌ and ▌security), which convey information about concern assignment in multi-concern code segments. Section 2.3 provides additional explanation of IVCon flags.

Figure 4 also demonstrates the usefulness of having descriptive, user-specified names for regions. Descriptive region names help users 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.

```
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 4: Concern-module formatting in IVCon

As another example, let us consider Figure 5, which shows how IVCon groups into one subconcern all syntactically equal code assigned to the same concern. Figure 6 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 (IVCon's token-level granularity in concern assignment enables such operations). 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 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). Actually, IVCon provides the added benefit that users can see (in the unwoven-concerns panel) a region-name reference to every use of every constant and can click any of those region names to see the context of that use in the unwoven-body panel. Of course, this example is just a special case of the general use of IVCon to provide both a global (i.e., woven) view of the code and a concern-specific (i.e., unwoven) view of the same code.

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). For simplicity, though, IVCon does not allow users to create or edit concerns or regions in the unwoven view. Although all concern- and region-manipulation operations must be performed in the woven view, users can easily transition from the unwoven to the woven view.

```
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 5: Unwoven view of the code in Figure 6

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

Figure 6: Woven view of the code in Figure 5

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. 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 3 that implements the `security` concern at region `before_protected_read`. In the woven view (Figure 2), 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 3 displays green flags (audit▌) and red flags(▌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.’’)`

is not 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. If IVCon did not consider flags when testing for concern-code equality, it would group both `accessLog.append(‘‘File read complete.’’)` and $^{\text{security}}$`accessLog. append(‘‘File read complete.’’)`$^{\text{security}}$ into a single subconcern, but the presence or absence of security flags in such a subconcern would be confusing because one of the instances of this subconcern has a nested security region, while the other does not.

3 Implementation

We have implemented a prototype of IVCon on a Java platform and made its source code publicly available [SL08]. The core IVCon application consists of 7793 lines of code, of which 7620 implement the GUI and 173 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 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 [Gut84, ABHY08]. 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 current position of the cursor or mouse). 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 [Had].

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

3.2 Translation Algorithms

Given a `regionMap`, `concernMap`, and `regionTree`, it is generally straightforward to implement the translations between woven and unwoven views.

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 enforce token-level granularity in concern assignment. After lexical analysis, IVCon starts with the woven code in the unwoven-body panel and iterates through all concerns in the `concernMap`, extracting the code in all of that concern's regions from the unwoven-body panel to the unwoven-concerns panel. Extracted concern code gets replaced with holes (□) of the appropriate color in the unwoven-body panel. During the extraction process, IVCon groups concern code into syntactically equal subconcerns in the unwoven-concerns panel 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 reconsider the woven-body code of Figure 2. 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, to 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 3 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.

While designing the weaving algorithm, one interesting issue arose, which relates to filling multi-concern holes with code. Because a code segment that fills in a multi-concern hole has been assigned to multiple concerns, it appears in multiple places in the unwoven-concerns panel. The interesting issue occurs when the user modifies multi-concern code in one place but not another in the unwoven-concern panel. Such a modification leads to code inconsistencies when filling multi-concern holes (i.e., which one of the concern modules should be used for filling a multi-concern hole). To ensure that such code inconsistencies do not occur, we have implemented linked editing in the unwoven-concerns panel [TBG04]. Code segments in the unwoven-concerns panel that fill in the same (multi-concern) hole get linked at runtime, so any changes made in one code location get reflected immediately in all the other code locations that fill in the same hole.

4 Case Study

To improve our understanding of IVCon's advantages and disadvantages, we used IVCon to implement several additional features for IVCon itself.

4.1 Experiment Design

Our case study involved extending IVCon with the following features:

1. Search for text in code. This serves a feature similar to <`ctrl-f`> in common editors.

2. Jump to a particular line number of code specified by the user. This helps users debug, as compilers typically report warnings and errors by line number.

3. Open multiple files simultaneously. This enables users to build projects that span multiple files.

4. Show in a tool-tip the concerns implemented by the code that the mouse pointer points to. This enables users to see which concerns a code segment S implements without actually moving the cursor to S.

Feature Implemented	Lines of code added	Time spent implementing, excluding time to assign code to concerns (hr:min)	Number of times used weaving and unweaving	Time spent defining and assigning code to concerns (hr:min)
Find text	57	3:07	11	00:07
Goto line	41	0:48	7	00:00
Multiple files	301	8:10	61	00:03
Tool tips	53	1:39	8	00:05
Linked editing	364	15:30	93	00:09
Flags in the woven view	346	9:42	108	00:01
Jump to a concern	81	0:43	4	00:00
Compile and execute	203	3:38	15	00:03
Total	**1446**	**43:17**	**307**	**00:28**

Figure 7: Usage statistics from case study

5. Integrate linked editing in the unwoven-concerns panel. This ensures that there are no code inconsistencies when filling in multi-concern holes during weaving (as mentioned in Section 3.2.2).

6. View flags in the woven view. This provides users with the same clarity of concern assignment in the woven view that currently exists in the unwoven view. Because concern assignments also serve as code documentation, this feature can improve users' comprehension of woven-view code. In addition, because flags stand out well from the rest of the code, this feature helps users quickly locate code assigned to particular concerns.

7. Jump to a specified concern module in the unwoven-concerns panel. This enables users to easily navigate in the unwoven-concerns panel.

8. Compile and execute code directly from IVCon. This saves users from having to switch to external tools for compilation and execution.

After implementing each of these features, we recompiled IVCon and worked in the newer version to implement the next feature. This bootstrapping approach ensured that we could draw the benefits of each new feature while implementing subsequent features.

4.2 Implementation Effort

As shown in Figure 7, implementing the case study entailed adding 1446 lines of code to our initial IVCon prototype. The total time spent implementing the case-study features was 43 hours and 44 minutes, out of which we spent 27 minutes and 31 seconds defining and assigning code to concerns. Hence, 1.05% of the total design and implementation time involved performing overhead actions necessary to reap IVCon's benefits (of creating, viewing, and editing code in both woven and unwoven views). The code assigned to concerns in the case study occupied 123 regions, which we assigned to 43 concerns. Taken together, these data show that our case study was a practical, medium-sized implementation effort (about one work week of time), and only a small portion (about one percent) of the implementation time had to be spent defining and assigning code to concerns.

4.3 Experiential Observations

IVCon aided code production in several ways, by providing all the benefits outlined in Sections 1, 2, and 4.1. Assigning all code implementing each of the case-study features to a

File Name	File Size (LoC)	Total No. of Concerns	Total No. of Regions	Avg. Region Size (chars)	Max. Region Size (chars)
IVCON.java	61	5	7	135.9	337
Windows.java	2759	20	84	914.4	24902
StressTest.java	100,000	1000	5000	998.0	3794

Figure 8: Test-file characteristics

File Size (LoC)	Type of File	File-open Time (ms)	File-save Time (ms)
61	.java file	29.43	45.00
	.ivc file (no concerns)	27.57	43.70
	.ivc file (5 concerns)	31.23	42.98
2759	.java file	499.57	46.24
	.ivc file (no concerns)	425.21	49.39
	.ivc file (20 concerns)	387.07	44.75
100,000	.java file	88,741.59	182.82
	.ivc file (no concerns)	17,010.13	100.26
	.ivc file (1000 concerns)	15,957.03	133.02

Figure 9: Performance opening and saving files

feature-specific concern was a significant aid when locating code for each concern as it was being implemented and tested, particularly because some of the concerns (e.g., `MultiFile`) crosscut several classes and methods (e.g., `FileUtilities.newFile()`, `FileUtilities.openFile()`, `Windows.stateChanged()`, etc.). It was also useful to assign syntactically equal code segments, which crosscut various functions, to the same concern (i.e., the concern of the feature being implemented), so we could update those code segments centrally. For example, while implementing the multiple-files feature, we made 9 centralized updates to 3 instances of a line of code used for switching between per-file data structures; these 9 updates would have required 27 updates in standard code editors.

5 Performance Evaluation

To evaluate the practicality of our design, we tested[2] IVCon by assigning code to concerns in two of our own IVCon source-code files: `IVCON.java` and `Windows.java`, respectively containing 61 and 2759 lines of Java code (in the woven view) and 7 and 84 regions assigned to 5 and 20 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. Figure 8 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 defined, and `.ivc` files with all concerns defined. Figure 9 displays 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

[2]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. 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.

File Size (LoC)	Region Size (LoC)	Number of Nested Regions	Concern-assignment Time (ms)
61	61	7	23.78
2759	500	21	79.51
	2759	84	352.40
100,000	1,000	53	760.95
	10,000	525	10,246.56
	100,000	5000	104,976.56

Figure 10: Performance assigning code to concerns

File Size (LoC)	Number of Characters Assigned to Concern	Concern-edit Time (ms)	Concern-removal Time (ms)
61	332	7.34	9.41
2759	1531	17.00	32.31
	1789	25.34	34.90
	3386	32.34	91.85
100,000	23335	367.29	512.38
	38462	419.38	685.00
	85069	810.26	1050.18

Figure 11: Performance editing concern colors and removing concerns

and concatenated to form the overall woven body. Conversely, IVCon saves `.java` files more 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 Figure 9 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. Figures 10 and 11 show IVCon's performance during concern assignment, editing, and removal. IVCon's performance when assigning code to a concern (Figure 10) 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 (Figure 11), 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 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.

Finally, we measured IVCon's performance weaving and unweaving code, as presented in Figure 12. 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 Figure 12 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 efficient when operating on reasonably sized source-code files. However, IVCon would need additional opti-

Lines of Woven Code	Number of Concerns	Holes in Unwoven Body	Weaving Time (ms)	Unweaving Time (ms)
61	0	0	3.27	2.66
	5	7	7.03	18.76
2759	0	0	109.67	59.39
	20	127	606.54	775.80
100,000	0	0	3,527.19	3445.31
	1000	7760	537,959.40	89,534.50

Figure 12: Performance weaving and unweaving code

mizations to perform tolerably efficiently on extremely large source-code files, such as those containing millions of source-language tokens.

6 Summary

We have presented IVCon, a tool for creating, examining, and modifying code in, and translating between, woven and unwoven views of code. IVCon differs from existing tools by providing features summarized in Figure 1. We have described IVCon's design, implementation, and case study and have made its source code publicly available [SL08]. In all of our empirical tests, which involved assigning concerns to, and modifying, real IVCon source code, IVCon performed efficiently on all practical source-code files.

Acknowledgment This research was supported in part by NSF grant CNS-0742736.

References

[ABHY08] 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.

[Coa03] Monica Yvonne Coady. *Improving evolvability of operating systems with AspectC*. PhD thesis, The University of British Columbia, 2003.

[ecl08] Eclipse, 2008. `http://www.eclipse.org/`.

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical Report 01.12, RIACS, 2000.

[GKY99] 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.

[Gut84] 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.

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

[jEd08] jEdit, 2008. `http://www.jedit.org/`.

[K0̈7] 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.

[KAK08] 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.

[KHH+01] 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.

[KLM+97] 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.

[LWZ06] 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.

[MMvD07] Marius Marin, Leon Moonen, and Arie van Deursen. SoQueT: Query-based documentation of crosscutting concerns. In *Proceedings of the International Conference on Software Engineering*, pages 758–761, 2007.

[OT00] 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.

[PC05] 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.

[PKH03] 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.

[RM02] M.R. Robillard and G.C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 406–416, 2002.

[SL08] Nalin Saigal and Jay Ligatti. IVCon – Inline Visualization of Concerns, 2008. `http://www.cse.usf.edu/~ligatti/projects/ivcon/`.

[SL09] Nalin Saigal and Jay Ligatti. Inline visualization of concerns. In *ACIS International Conference on Software Engineering Research, Management, and Applications (SERA)*, December 2009.

[TBG04] M. Toomim, A. Begel, and S.L. Graham. Managing duplicated code with linked editing. *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 173–180, Sept. 2004.

[TOSH99] 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.

[Vis08] The Visualiser, 2008. `http://www.eclipse.org/ajdt/visualiser/`.

[YFG+06] 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.