Software Plans for Separation of Concerns

David Coppit
Department of Computer Science
McGlothlin-Street Hall
The College of William and Mary
Williamsburg, VA 23185 USA
david@coppit.org

Benjamin Cox
Department of Computer Science
McGlothlin-Street Hall
The College of William and Mary
Williamsburg, VA 23185 USA
btcoxx@cs.wm.edu

ABSTRACT

Complex software often has concerns which cut across the modules of the system. Aspect-oriented programming languages such as AspectJ attempt to address this problem by providing a new abstraction for encapsulating such concerns called *aspects*. Aspects are integrated automatically during compilation with the base code at well-defined join points. This approach is difficult to apply when concerns are highly context-dependent and have complex relationships not supported by the language. In this paper, we propose an alternative approach based on *software plans*. In this approach, a specialized editor is first used to annotate code segments as belonging to one or more concerns. The user can then specify a limited view of the code, a *plan*, which consists of some desired subset of the concerns. Using this plan view, the user can directly implement any complex relationship between overlapping, interdependent concerns. We present our approach using a motivating example from the GNU *grep* tool. We also present our prototype editor implementation.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques– *program editors.*

General Terms

design, languages

Keywords

software plans, separation of concerns, aspects

1. INTRODUCTION

Complex software often has multiple overlapping and interdependent concerns. The traditional approach is to attempt to aggregate related concerns using a functional or object-oriented decomposition of the code. More recently, language designers have provided more powerful language abstractions for representing concerns as cross-cutting *aspects* [6,7]. In all of these approaches, source code is re-modularized in an attempt to improve the cohesion of code serving certain concerns while minimizing the coupling between the modules.

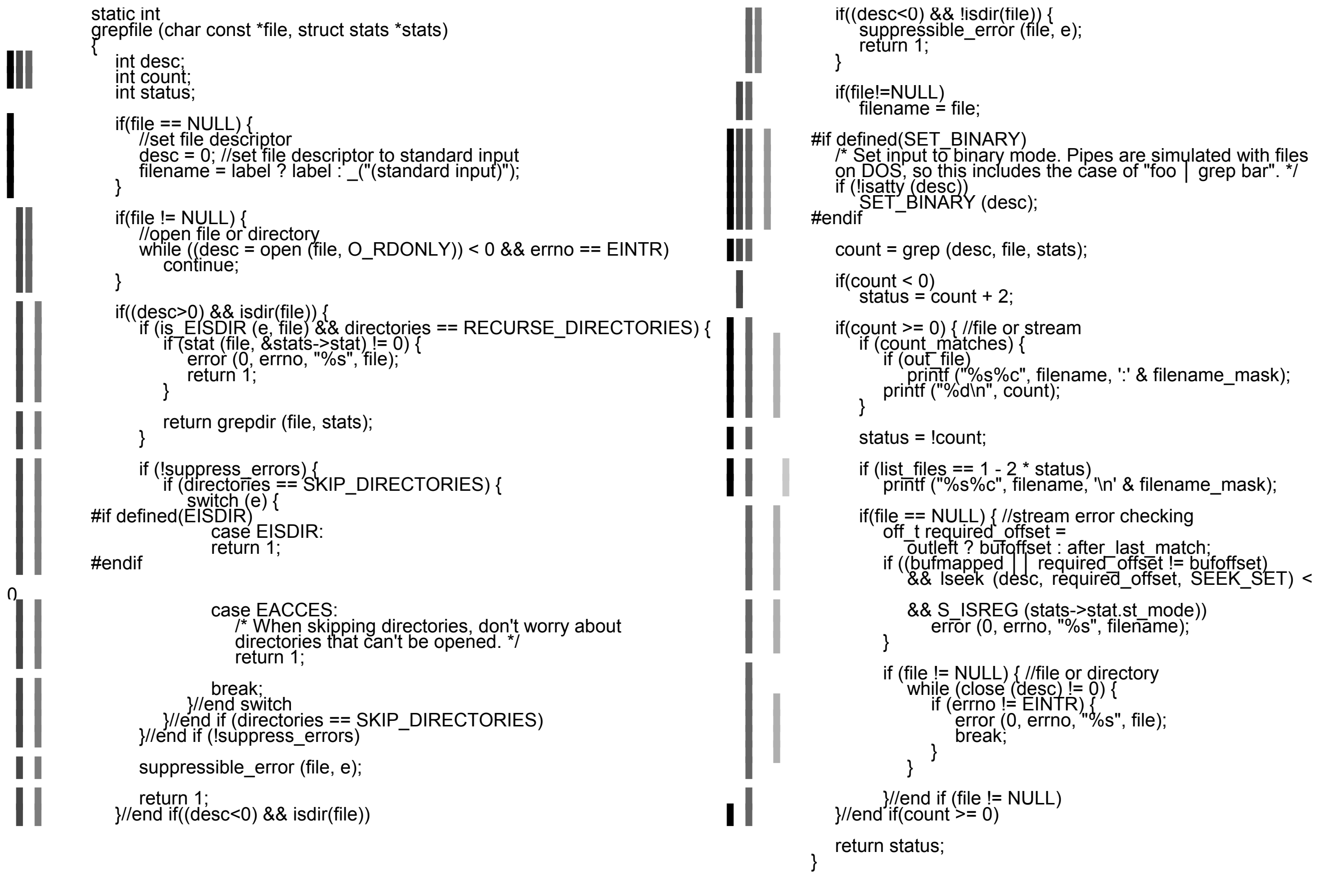
Unfortunately, these approaches are difficult to apply to overlapping and interdependent concerns. In such cases, modularizing a system to improve the coupling and cohesion of one concern may increase the tangling of other concerns. For example, debugging code is often scattered throughout the software. Attempting to restructure the system to improve the cohesion of the "debugging" concern would aversely affect the functional or object-oriented decomposition.

Unfortunately, aspect-oriented programming languages only partially address this problem. An inherent assumption of aspect-oriented programming languages is that it is possible to provide general declarative mechanisms for specifying the location of cross-cutting concerns, and that these mechanisms can be used by the *aspect weaver* to automatically integrate the concern code into the existing modular structure. AspectJ [7], for example, allows the programmer to specify *join points* at calls to methods and constructors, references or uses of fields, executions of exception handlers, object initialization, etc. Each of these join points requires only limited context, and are suitable for automatic integration by the weaver. None allow a concern to be integrated into an arbitrary program location. Indeed, it is not clear that this can be done automatically for concern code which depends heavily on its context in the base code. For example, trace messages used during debugging to record the flow of execution of a program depend heavily on context, and can not be integrated automatically by a weaver.

Carver and Griswold's [2] analysis of concerns in GNU `sort` demonstrates that such complex interdependencies between concerns and base code, which they call "invasive compositions", do arise in practice. Proper integration of such concerns by the weaver would require code modifications to be coordinated at multiple locations, and for concerns to be composed in the correct order. They note that one approach to dealing with such difficulties using existing join point models is to decompose complex expressions into a series of more "atomic" expressions, and extend the model to allow any series of statements to be a join point.

Murphy et. al [10] describe the process of restructuring code so that aspects can be encapsulated using the limited join point mechanisms provided by languages such as HyperJ and AspectJ. For example, they describe a somewhat byzantine approach to dealing with concern-specific code in `if-then-else` branches. For AspectJ, an "around" advice is used to bypass the original implementation of a method, instead executing "concern-optimized" versions of code which contain the appropriate branch of the code depending on the concern. They also describe a similar approach in which concern-specific code is moved to the beginning or end of a method, where the aspect weaver can integrate aspects.

In this paper we propose an alternative approach which avoids the difficulties associated with an automatic weaver, while still allowing concerns to be conceptually separated. The code is



```
        static int
        grepfile (char const *file, struct stats *stats)
        {
            int desc;
            int count;
            int status;

            if(file == NULL) {
                //set file descriptor
                desc = 0; //set file descriptor to standard input
                filename = label ? label : _("(standard input)");
            }

            if(file != NULL) {
                //open file or directory
                while ((desc = open (file, O_RDONLY)) < 0 && errno == EINTR)
                    continue;
            }

            if((desc>0) && isdir(file)) {
                if (is_EISDIR (e, file) && directories == RECURSE_DIRECTORIES) {
                    if (stat (file, &stats->stat) != 0) {
                        error (0, errno, "%s", file);
                        return 1;
                    }

                    return grepdir (file, stats);
                }

                if (!suppress_errors) {
                    if (directories == SKIP_DIRECTORIES) {
                        switch (e) {
        #if defined(EISDIR)
                            case EISDIR:
                            return 1;
        #endif

                            case EACCES:
                                /* When skipping directories, don't worry about
                                directories that can't be opened. */
                                return 1;

                            break;
                        }//end switch
                    }//end if (directories == SKIP_DIRECTORIES)
                }//end if (!suppress_errors)

                suppressible_error (file, e);

                return 1;
            }//end if((desc<0) && isdir(file))

            if((desc<0) && !isdir(file)) {
                suppressible_error (file, e);
                return 1;
            }

            if(file!=NULL)
                filename = file;

        #if defined(SET_BINARY)
            /* Set input to binary mode. Pipes are simulated with files
            on DOS, so this includes the case of "foo | grep bar". */
            if (!isatty (desc))
                SET_BINARY (desc);
        #endif

            count = grep (desc, file, stats);

            if(count < 0)
                status = count + 2;

            if(count >= 0) { //file or stream
                if (count_matches) {
                    if (out_file)
                        printf ("%s%c", filename, ':' & filename_mask);
                    printf ("%d\n", count);
                }

                status = !count;

                if (list_files == 1 - 2 * status)
                    printf ("%s%c", filename, '\n' & filename_mask);

                if(file == NULL) { //stream error checking
                    off_t required_offset =
                        outleft ? bufoffset : after_last_match;
                    if ((bufmapped || required_offset != bufoffset)
                        && lseek (desc, required_offset, SEEK_SET) <

                        && S_ISREG (stats->stat.st_mode))
                            error (0, errno, "%s", filename);
                }

                if (file != NULL) { //file or directory
                    while (close (desc) != 0) {
                        if (errno != EINTR) {
                            error (0, errno, "%s", file);
                            break;
                        }
                    }

                }//end if (file != NULL)
            }//end if(count >= 0)

            return status;
        }
```

Figure 1: The grepfile function tagged with concerns

treated by the source editor as multiple inter-related layers or *plans*. A plan is a view of the software that contains only the code segments related to those concerns of immediate interest. The developer can edit the code in this view, in which case the editor automatically updates the concern information (e.g. tagging new code as belonging to the same set of concerns as the edited code). Because a particular code segment may be tagged as belonging to multiple concerns, it may also be visible in a different plan. When the source code is finally compiled, the editor renders the tagged code as a traditional monolithic code representation.

Currently, we have finished enhancing an integrated development environment to support editing of plans. Our next step is to test the approach in one or more case studies. Eventually we hope to enhance the editor to provide better automated support for tagging and editing of code related to particular concerns.

In Section 2 we present our approach in more detail, with a motivating example. Section 3 describes the implementation of plans in the Eclipse IDE. Section 4 describes our planned evaluation. Section 5 presents related work. Section 6 describes key challenges, and Section 7 concludes.

2. APPROACH

In this section we present our approach in more detail. We will use the GNU `grep` [4] program as a running example, showing how even a simple program can have complex relationships between concerns.

Figure 1 presents the key function in `grep` for searching a file, directory, or input stream for a given pattern. [1] In this example, we have used a line of code as the smallest code segment that can be related to a concern. The bars to the left of the lines indicate the concerns that are related to the line. The bars are colored, and bars of the same color are aligned in the same column. In this case, we have tagged the code with seven concerns:

▌Processing of input streams

▌Processing of a directory

▌Processing of a file

▌Error handling

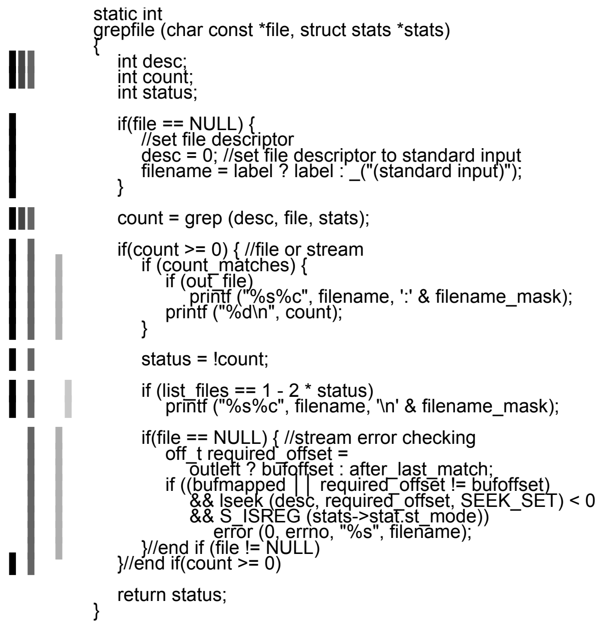
▌Binary files

▌The `-c` option to output the number of matches

▌The `-l` option to output the matching filenames

For example, the first and last few lines are not tagged, indicating that they appear in all plans. The first conditional block is tagged as belonging to the "Processing of input streams" concern, and the next conditional block is tagged as belonging to both the "Processing of a directory" and "Processing of a file" concerns.

[1] The code has been modified slightly to improve clarity.



```
static int
grepfile (char const *file, struct stats *stats)
{
    int desc;
    int count;
    int status;

    if(file == NULL) {
        //set file descriptor
        desc = 0; //set file descriptor to standard input
        filename = label ? label : _("(standard input)");
    }

    count = grep (desc, file, stats);

    if(count >= 0) { //file or stream
        if (count_matches) {
            if (out_file)
                printf ("%s%c", filename, ':' & filename_mask);
            printf ("%d\n", count);
        }

        status = !count;

        if (list_files == 1 - 2 * status)
            printf ("%s%c", filename, '\n' & filename_mask);

        if(file == NULL) { //stream error checking
            off_t required_offset =
                outleft ? bufoffset : after_last_match;
            if ((bufmapped || required_offset != bufoffset)
                && lseek (desc, required_offset, SEEK_SET) < 0
                && S_ISREG (stats->stat.st_mode))
                    error (0, errno, "%s", filename);
        }//end if (file != NULL)
    }//end if(count >= 0)

    return status;
}
```

Figure 2: The stream-only plan for the grepfile function

Note that even in this simple function there are many crosscutting concerns that make the code difficult to understand. For example, the binary filesystem concern is completely independent of the error handling concern. In this case, we could create a plan in which either concern is viewed and edited without the other.

There are also concerns that are dependent on other concerns. For example, the error handling concern is dependent on the directory, file and stream concerns. Viewing the error handling concern code which deals with directories without also viewing the directory concern would result in meaningless code. There is also an implicit ordering dependency between the "Binary files" concern and the file, directory, and stream processing concerns–the file descriptor must be set to binary mode before calling the `grep()` function.

The editor automatically tags new lines of code as belonging to the concerns of the edited text. For example, if the programmer is editing a block of code related to the "binary files" concern, the editor will automatically tag new code as belonging to that concern. While this approach suffices for the majority of editing operations, it is not a complete solution. For less common editing of concern code, the developer can manually tag a code segment as belonging to a concern. In using our prototype implementation, we have identified several situations where program analysis by the editor can provide automated assistance to further reduce the need for manual tagging. We discuss this issue in more detail in Section 6.

Once the code is tagged, the developer can specify a plan consisting of one or more concerns. Plans allow the developer to deliberately ignore concerns which are not apropos to the current activity. For example, consider the plan shown in Figure 2, a view of the system that contains the stream concern but not the file, directory, or error-checking concerns. The code is more than half as short and is easier to understand. In addition, the plan provides a coherent, even compilable, view of the code.

Plans are easy to use and allow the programmer to focus on different aspects of interest. The programmer can use plans to manage complex overlapping concerns, and can easily resolve interactions between two concerns by creating a new plan that shows both. Tags also serve as documentation, helping a developer unfamiliar with the code to easily and quickly determine the concerns associated with a given line of code, as well as interactions between concerns.

3. PROTOTYPE IMPLEMENTATION

Figure 3 shows a screenshot of our prototype implementation. In this view, the code for the `grep` utility is currently being edited. In the left are the colors associated with the various concerns. The programmer has selected some text to be tagged, and one can see the names of the available concerns in the cascaded context menu. As the programmer modifies the code, the IDE will automatically update the concern meta-data.

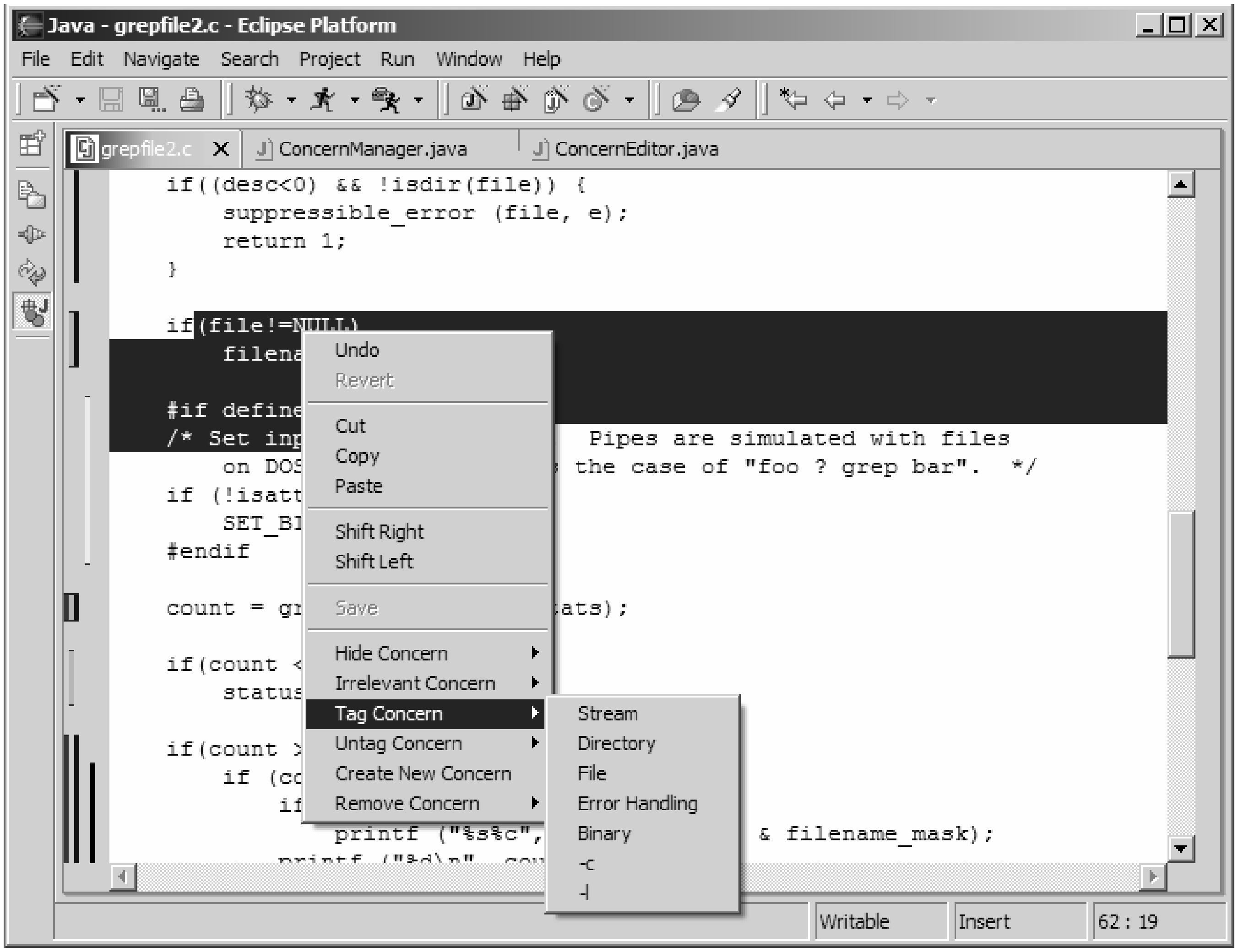
In our current implementation, the smallest code segment that the editor allows to be tagged is a single line. Currently the source code is stored internally as a single monolithic representation (even though, in general, lines of code for unrelated concerns can have any ordering). When the file is saved, the monolithic representation is saved as the file, and the concern information is saved separately. This provides backwards-compatibility with tools that expect a traditional monolithic format. Currently the tool does not perform any analysis for automatic tagging of code.

In order to implement this functionality, we customized the open source Eclipse IDE [2]. Eclipse provides an API for the IDE which allows developers to extend its functionality. For example, we mark ranges of text for a particular concern using the `Position` class. Similarly, our annotations are implemented using the `Annotation` and `AnnotationRulerColumn` classes. We have also modified the Eclipse IDE to allow the user to specify a plan as a set of visible concerns. Our current policy allows the user to force concerns to be hidden, or to optionally hide concerns. Code related to the latter type of concern will be visible if it is also tagged with some other visible concern.

4. EVALUATION

In order to evaluate our approach we will conduct several case studies in which our editor is used to develop several software systems. While developing the software we will investigate the theoretical as well as practical strengths and weaknesses of our approach:

- Are concerns conceptually separable? It may be the case that there is a poor correspondence between concerns and code.
- Is an editor-based application sufficient to easily separate the concerns? A primarily syntax-based tool may not be powerful enough to allow the user to easily separate concerns.
- Does this approach lower the conceptual complexity? Is it easier to write and understand code with tangled concerns? Is it easier to maintain code using this method?
- Is it possible to effectively filter irrelevant concerns while preserving all the necessary details in a coherent manner? We believe that our proposed approach to filtering lines



```
    if((desc<0) && !isdir(file)) {
        suppressible_error (file, e);
        return 1;
    }

    if(file!=NULL)
        filena

    #if define
    /* Set inp                    Pipes are simulated with files
        on DOS                  the case of "foo ? grep bar".  */
    if (!isatt
        SET_BI
    #endif

    count = gr                  ats);

    if(count <
        status

    if(count >
        if (cc
            if
                printf ("%s%c",              & filename_mask);
```

Figure 3: The grepfile function tagged with concerns

will yield coherent plans. However, it may be the case that this approach, more often than not, results in plans that are not understandable.

- What programming languages work well using this approach? Because of the line-oriented nature of this approach, procedural languages seem most suited. However, object-oriented languages may also work well.

5. RELATED WORK

Aspect-oriented programming (AOP) [6] uses “aspects” to encapsulate the concerns. The aspects are then “woven” into the code automatically by the compiler. The original formulation of AOP required custom compiler support for weaving different types of aspects. More recent efforts in the development of AspectJ [7] have attempted to provide a general method for writing aspects and weaving them into the base object-oriented code. Our approach is editor-oriented rather than language- or compiler-oriented, and can therefore be used with a range of languages. In addition, our approach allows (and requires) the programmer to express the complex relationships between overlapping and interdependent concerns. In contrast, languages such as AspectJ limit the integration of aspects and base code to only those program locations (join points), which are supported by the language. In particular, the language does not allow arbitrary aspect code to be inserted into arbitrary locations in the main code. For example, the two lines in Figure 1 which implement the `-l` functionality (near the middle of the right column) are dependent on the context. They are dependent on the execution of the grep function call, as well as the previous line assigning setting the value of `status`. The former is supported by AspectJ’s “after returning” advice, but AspectJ’s “set()” pointcut designator does not provide enough context to allow the `-l` code will be integrated after the assignment.

Lai and Murphy [8] analyze the relationship between concerns and code structure. Their FEAT tool allows the user to tag lines of code in a manner very similar to ours. However, their tool does not support the notion of software plans—all code related to all concerns is always visible. However, their tool does parse the code to create an abstract syntax representation, which allows them to analyze the relationship of a set of concerns to the existing code structure. In particular, they measure the proportion of files which contain code related to a concern (“spread”), the proportion of tokens for a concern which are also involved with another concern ("tangle"), and the proportion of tokens in files for a concern which involve that concern ("density").

Program slicing [11] attempts to reduce the complexity of code by extracting only those lines of code that can alter, or are altered by, a particular variable. The extracted subset is a working program that is similar to our "plans". Unlike their automated approach, our approach is manual but more flexible in that any set of lines can be associated with a concern. Also, it is not always the case that a program variable correlates to a single concern. A variable may have multiple uses in different concerns in a program; conversely, a particular concern may require the use of multiple variables.

Information transparency [5] attempts to identify related sections of code that are dispersed throughout the source code, by using inference and searching tools. The basic idea is to identify concerns lexically, based on characteristics such as variable names, or syntactically, based on characteristics such as loop structure. Unlike information transparency, in our approach the tool helps the programmer explicitly define which sections of code are related, and does not involve after-the-fact deduction. More effort is involved to tag lines of code, but our approach can provide coherent views of the code, while information transparency presents disconnected but related lines of code.

Finally, some editors support hiding of `#ifdef`/`#endif` text based on user-specified values for the relevant symbols. Emacs [3], for example, has a hide-ifdef-mode [9]. The basic idea is similar to what we propose, although editor support is limited. In fact, our early experiments to assess the feasibility of a line-based tagging strategy used the C pre-processor in this manner. However, using pre-processor directives is obviously tedious and results in overly difficult to read code.

6. OPEN QUESTIONS

Initial use of our tool has already revealed a number of key open questions. The first question is the extent to which the management and tagging of code with concern information can be automated. Aspect languages relieve the developer of the burden of integrating aspects into base code. Our approach, in contrast, allows the user to integrate highly context-dependent concerns into the base code, but provides editor-based concern management capabilities instead of automated integration. The costs associated with manual integration of concerns are no worse than that of code developed without aspects in mind. However, tagging of concerns is an additional cost, and should therefore be as inexpensive as possible. We are not yet sure of the extent to which our current editing operations help the user to tag code. One method to enhance automatic tagging is to employ program analysis to infer that lines of code belong to the same concern. For example, the use of a variable defined to be in another concern would indicate that the code using the variable belongs to that concern.

The second issue is the view consistency problem. Editing operations in a given plan should modify the hidden code in a consistent manner. For example, there are a number of ways to handle the situation in which the user deletes a block of code containing hidden text belonging to a concern not in the current plan. Our tool's current strategy is to detect this situation and disallow the operation. In effect, this forces the user to make the hidden text visible and resolve the conflict. An alternative is to use an internal representation of the code which better models concern dependencies—if the hidden concern is independent of the current plan, the visible code can be deleted while leaving the hidden concern. Obviously, a complete solution requires program analysis to guarantee that the deletion of the visible code does not change the semantics of the hidden code.

A third open question is the extent to which code can truly be simplified in the manner illustrated in Figure 2. It seems that some rewriting of the visible code in a plan is necessary in order to arrive at a concise, easy-to-understand representation. We took some liberty in Figure 1 by splitting an `if-then-else` statement into the first two `if-then` statements. This allowed us to tag the entire `if-then` statements as belonging to one concern or the other. In the original representation, we would have been forced to tag the contents of the branches and not the `if-then` statements themselves in order to avoid `else` clauses without associated `if` statements. A side effect of this strategy is empty "`{}`" blocks in certain plans. Clearly some sort of "pretty printing" of the code is necessary to remove such noise, as well as careful management of editing operations.

Finally, we must expand our own evaluation of the approach outlined in Section 4 to include evaluation via user studies. Addressing the issues described above can help reduce the costs associated with using this approach. However, it should be possible to evaluate the basic idea using the prototype we have already implemented.

7. CONCLUSION

In this paper we have presented a new, editor-based approach to dealing with tangled concerns. Inspired by the use of plans in other engineering disciplines, our approach attempts to provide the developer with the capability to create complex relationships between concerns, while, at the same time, providing mechanisms for keeping them manageable.

While our approach shows some promise, evaluation is an obvious area of future work. In addition, there is an opportunity to exploit information from analysis of the source code in order to automate much of the manual labor required by our initial prototype. In addition, the filtering can be made "smarter" to address anomalies such as empty "{}" brackets resulting from hiding the body of the block.

ACKNOWLEDGEMENTS

We would like to thank the Eclipse developers, especially Tom Eicher, for their technical assistance in modifying Eclipse to support editing of plans. We would also like to thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] Lee Carver and William G. Griswold. Sorting out concerns. Position paper for Multi-Dimensional Separation of Concerns Workshop, OOPSLA 1999.

[2] Eclipse.org, The Eclipse homepage. URL: http://www.-eclipse.org/

[3] The GNU Project, The Emacs homepage. URL: http://-www.gnu.org/software/emacs/emacs.html

[4] The GNU Project, The grep homepage. URL: http://www.-gnu.org/software/grep/grep.html

[5] W. G. Griswold. Coping with Crosscutting Software Changes Using Information Transparency. In *Reflection 2001: The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Kyoto, September 2001.

[6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97: Proceedings of the European Conference on Object-Oriented Programming*, pages 220-42. Springer-Verlag, 9-13 June 1997.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001: Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327-353. Springer-Verlag, Budapest, Hungary, June 2001.

[8] Albert Lai and Gail C. Murphy. The Structure of Features in Java Code: An Exploratory Investigation. Position paper for Multi-Dimensional Separation of Concerns Workshop, OOPSLA 1999.

[9] Brian Marick and Daniel LaLiberte. hide-ifdef-mode.el. URL: http://www.mit.edu/afs/athena/contrib/epoch/lisp/hideif.el

[10] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating Features in Source Code: An Exploratory Study. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 275-85, Toronto, Canada, 12-19 May 2001. IEEE.

[11] Mark Weiser. Program slicing. IEEE Transactions on Software Engineering, SE-10(4):352-7, 1984.