

Spotlight: A Prototype Tool for Software Plans

David Coppit, Robert R. Painter, Meghan Revelle

Department of Computer Science

The College of William and Mary

Williamsburg, VA 23185

david@coppit.org, rrpain@cs.wm.edu, meghan@cs.wm.edu

Abstract

Software evolution is made difficult by the need to integrate new features with all previously implemented features in the system. We present Spotlight, a prototype editor for software plans that seeks to address this problem by providing the programmer a principled way to separately develop and incrementally integrate independent features.

1. Introduction

Software development involves the implementation and integration of multiple concerns, such as the features of the program or more general concepts such as debugging, authentication, and synchronization. The programmer typically adds concerns to a stable codebase, incrementally following an order determined by concern dependencies.

Unfortunately, as the software grows in size, the marginal cost of adding a new concern grows dramatically. This is because the new concern must be implemented and integrated not only with the concerns upon which it depends but also with all other concerns that happen to have been implemented earlier. The programmer cannot implement the new concern solely in terms of the necessary dependent concerns then incrementally integrate the new concern with the previously implemented concerns. As a result, large systems become extremely difficult to evolve, even for relatively small changes.

We present *Spotlight*, a prototype editor for software plans [3,4]. A *software plan* is an editable view of a module that presents the code for the concern being implemented along with the code for the concerns upon which it depends, but not code for independent concerns. The overall implementation of the module is a set of plans.

Spotlight allows the programmer to separate independent concerns by implementing them in separate plans. Because concerns are never fully independent, Spotlight allows shared dependent code to be implemented in a shared parent plan. Our model is textual and fine-grained in the sense that concerns can be associated with code segments

as small as a single character. Once a concern has been implemented and verified, it can be incrementally integrated with an existing independent concern by creating a new plan that incorporates the two plans as parents.

In the next section, we introduce a motivating example that we will use to illustrate the use of our tool. Section 3 introduces the Spotlight tool and describes some of its features. Sections 4 and 5 illustrate the use of Spotlight to implement and integrate a new concern. Section 6 describes related work, and Section 7 concludes.

2. A Motivating Example

During the development of software plans and Spotlight, we have used various versions of the tool to implement a number of case studies including the GNU `sort`, `cat`, and `wc` utilities, and a larger traffic simulator program. We use GNU `cat` as a motivating example in this paper. `cat` concatenates files specified on the command line or standard input and outputs the result to standard output. The GNU version has 12 flags that control features such as numbering of lines and escaping of non-printing characters.

A programmer implementing this program might begin by implementing concerns such as the basic concatenation functionality and parsing of command line arguments. Since features involving the various options of the program depend upon parsing of command line arguments, the options parsing concern might be implemented after the base concatenation functionality. In a similar way the implementation of the remaining concerns such as escaping of non-printing characters, exception handling, numbering of lines, etc. can be completed in an incremental manner. In order to avoid refactoring of code, programmers often attempt to order the implementation of concerns such that dependent concerns are implemented after the concerns they depend on. Concerns that have no relative dependence, such as display of the help message and concatenation of files, can be implemented in some arbitrary order.

Now let us consider a change to the program after the initial development effort described above: the program

must add a line number for the non-blank lines when the `-b` flag is specified. The programmer encounters two problems. First, this new concern only depends directly upon the parsing of arguments and basic concatenation concerns. However, it must be implemented in terms of the entire program, which includes concerns such as showing of non-printing characters, numbering of all lines, display of help information, and squeezing of blank lines. This is more costly than if the feature had been implemented earlier in the development process. Second, the new feature must be simultaneously integrated with all interacting concerns. There is no way for the programmer to separately integrate and verify the behavior of the new feature relative to the other concerns of the software. For example, the interaction with the “number all lines” concern cannot be resolved separate from the other independent concerns.

Admittedly, this example is simplistic, but it illustrates the general problem. In large systems, new features rarely involve all previously implemented features, and their interaction with existing features is rarely straightforward. We need a software model that maintains the independence of independent concerns and allows incremental, pairwise integration of new features with existing ones.

3. Spotlight

Spotlight is a prototype editor for *software plans* [3, 4]. The tool implements a program storage model that enables code fragments to be associated with concerns and shared between plans [8]. In the model, each plan corresponds to a concern. As code is edited, it is grouped into “blocks” of contiguous text. Blocks are shared between plans, either through the creation of a plan from another (inheritance of blocks) or through copy and paste operations (duplication of block references). Concern dependencies are expressed in terms of the *parents* relation of plans.

Importantly, this model is largely managed transparently to the user, who simply edits code as he or she always would. For example, the creation of a new plan from another parent plan causes the parent plan’s blocks to be inherited in the new child plan. If the user inserts or deletes text in the new plan, the inherited blocks are automatically split and merged as necessary. In this way, the original parent plan’s relationship with the child plan is maintained without affecting the parent plan.

Sharing of code blocks is important because it enables code to be merged when two plans are integrated. To integrate two plans, the user creates a new plan having the two plans as parents. The new plan inherits a set of code blocks from the two parents, comprised of shared blocks as well as blocks unique to the two parent plans. Spotlight implements a topological block ordering heuristic that we developed. The algorithm computes a new ordering that at-

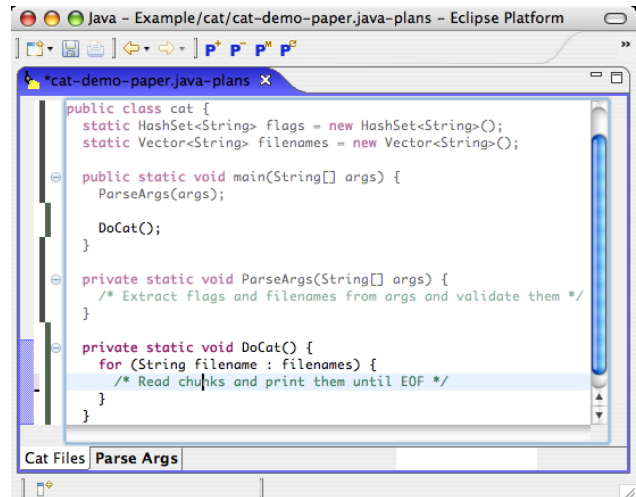


Figure 1. Spotlight showing the “Cat Files” plan

tempts to maintain the partial orderings of the blocks in the parent plans while avoiding interleaving the unique blocks inherited from different plans.

We implemented software plans in Spotlight, an extension of the Eclipse [5] IDE. This approach allowed us to leverage the rich functionality of Eclipse and greatly reduce development effort. Although software plans are language independent, our current implementation uses Eclipse’s Java Development Toolkit for syntax highlighting, annotations, compilation, etc.

Figure 1 shows a screenshot of the tool. The figure shows the plan for the “concatenation of files” feature for our `cat` example. The main editor area displays the code for the plan, which is simply the concatenation of the plan’s blocks¹. As this text is edited, Spotlight captures the changes and updates the underlying plans model. Along the bottom of the editor is a row of tabs, each of which contains a software plan for the module.

The tool allows the programmer to specify one plan as the compilable representative for the module. In this case, the “Parse Args” plan is the compilable plan. Using this feature, the programmer can run and test features in isolation. For example, the programmer can make sure the implementation of the number non-blank lines concern is correct before integrating it with the other features.

Spotlight provides features to view the association of code to plans. As shown in Figure 1, inherited code such as that for the “Parse Arg” plan can be dimmed. The colored columns on the left side of the editor indicate the plans from which code was inherited. By placing the mouse cursor on a column, the user can view the name of the plan associated with the column. The tool also has a feature to draw block boundaries in the view (not shown).

¹In the figures, we abbreviate the implementation of the methods.

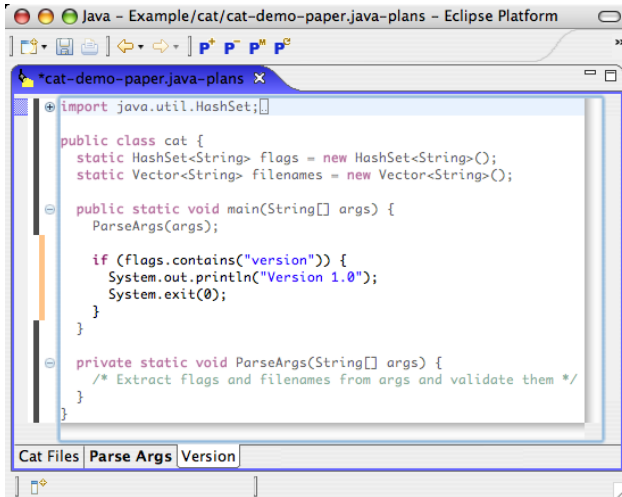


Figure 2. Spotlight showing the “Version” plan

The right-most toolbar above the editor provides buttons to create, delete, modify or update plans. Modifying a plan allows the user to change its name or parent plans. Updating a plan causes new blocks in parent plans to be incorporated into the current plan. We also enhanced the IDE by adding a menu (not shown) that provides additional functionality to set the compilable plan for the module and control various display options.

4. Implementing Independent Concerns

Spotlight allows programmers to implement a new concern in terms of only those concerns upon which it depends. To do so, the programmer creates a new plan, specifying the plans that contain the dependent concerns as parents. The tool automatically includes the code from the parent plans in the new plan. The user can then edit the plan to implement the new concern.

In general, the concern dependencies form a graph that is directed and acyclic. Independent concerns result in tree-like relationships among plans, in which siblings implement independent concerns and share parent plans that implement shared concerns. Consider, for example, the implementation of the version concern in Figure 2. This plan is implemented in terms of the “Parse Args” plan upon which it depends but not in terms of the “Cat Files” plan upon which it does not depend.

Spotlight’s parent plan relationship is flexible and can be evolved as the programmer’s understanding of the concern dependencies changes. For example, the programmer could have implemented the version concern as a program that always displays the version regardless of the program arguments. In this case, its plan would not need to have the parsing of arguments plan as a parent, which would allow the programmer to defer the integration of this concern.

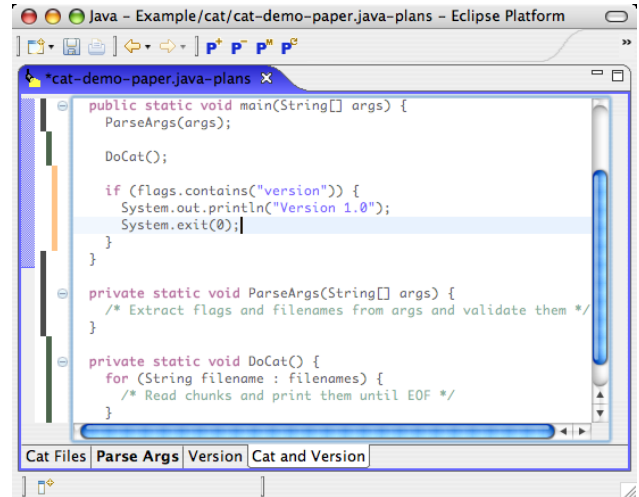


Figure 3. Spotlight showing the integration plan for the “Cat Files” and “Version” plans

5. Incremental Integration of Concerns

Separately implemented concerns must be integrated and reconciled, either in order to implement a new concern or to produce the final module that implements all of the desired functionality. To do so, the user creates a new plan in Spotlight that specifies the plans for the concerns to be merged as parents. Spotlight then employs our block ordering heuristic to create an initial ordering of the blocks inherited from the parent plans. The resulting plan can then be edited, and the merged concerns can be reconciled.

Figure 3 shows the plan that results from combining the “Cat Files” and “Version” plans using the block ordering heuristic. Because the underlying model shares blocks, Spotlight can avoid duplicating code such as the declaration of `main` that appears in both parent plans. The ordering of the calls to `ParseArgs` and `DoCat` is determined by the ordering provided by the original “Cat Files” plan.

However, other blocks have no known ordering. In many cases, such as method definitions and variable declarations, the relative ordering does not matter. But in the figure, one can see that the printing of version information is improperly positioned directly after the call to `ParseArgs`. This is an example of one concern interfering with another. In this case, the resolution is simple: reorder the blocks. Other resolutions may include the addition of conditional code or other editing. Later plans derived from this one will include these resolutions.

In general, independent concerns can interfere with each other in nontrivial ways that must be resolved by the programmer as part of the integration process. Programmers intuitively understand this, usually planning integration of modules in order to address complex integrations first. The approach when using software plans is similar: highly in-

terfering concerns should be integrated first in a pair-wise fashion. In cases where several concerns tend not to interfere, one plan can be created to reconcile them all. Spotlight thus provides a mechanism for incremental integration of separately implemented concerns.

6. Related Work

There is a large amount of related work in the areas of separation of concerns and multiple views in software engineering. In the interest of space, we focus here on editors that support partial views of software.

A number of tools have been developed to hide portions of code in order to highlight relevant code. One approach taken by many IDEs, including Eclipse, is “code folding” or “code elision” in which blocks of code can be collapsed. Other tools use programmer metadata to identify collapsible code. One example is P-Edit [7], which allows the user to associate a Boolean expression with each line of code. Given an “effective mask” of Boolean values, the editor dynamically generates a version of the code that contains only those lines whose expressions are satisfied by the mask. Unlike our approach, such editors operate on a traditional code representation that is not concern-oriented and does not model independence of concerns.

A number of concern-oriented tools have also been developed using various means of identifying concern code and removing irrelevant code. The Aspect Browser [6] uses lexical techniques to find matching code and display it in a single view and provides tools for navigating to the different program locations. Black and Jones [1] use an abstract program structure to create “perspectives” on the code. Work by Chu-Carroll on visual separation of concerns [2] is similar to ours, presenting sets of related methods in a single editable view. In contrast to our approach, theirs does not model concerns and dependencies explicitly, and it does not operate at a fine-grained character level.

7. Conclusion

Research on Spotlight and software plans is ongoing. In our current prototype, programmers can only use Spotlight on legacy code to implement and integrate new concerns. One author (Revelle) developing techniques for reverse engineering software plans from legacy code. This would allow previously implemented concerns to be separated and maintained, and would allow pairwise integration of new concerns with existing concerns. We also plan to implement plan refactoring operations for cases when dependent concerns are implemented out of order.

Our current prototype requires the programmer to manually reconcile merged plans. Another author (Painter) is developing a notion of plan interfaces to ease the integra-

tion of plans. Interface information is injected into other plans so that they can be implemented in a way that will ease later integration. We also plan to revise our model to use the parse tree, which would allow us to utilize language semantics in the block merging heuristic.

Spotlight is still a prototype that lacks many features that are necessary in practice. For example, there are no features to manage global concerns that span multiple modules of the system. The software plans approach also needs further evaluation, perhaps by further developing the Spotlight tool and employing it in larger case studies.

Despite its rough edges, Spotlight demonstrates the potential of the software plans approach. Using Spotlight, a programmer can decouple the implementation and integration activities of new features. It also enables an incremental approach for the integration of the new feature with independent but possibly interfering features.

Acknowledgments

The authors thank Ben Cox and Justin Manweiler for their efforts implementing earlier versions of Spotlight. This research was supported in part by the United States Air Force Office of Scientific Research under grant number FA9550-07-1-0030.

References

- [1] Andrew P. Black and Mark P. Jones. The case for multiple views. In *ICSE 2004 Workshop on Directions in Software Engineering Environments*, May 2004.
- [2] Mark C. Chu-Carroll, James Wright, and Annie T. T. Ying. Visual separation of concerns through multidimensional program storage. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 188–97, Boston, Massachusetts, 17–21 March 2003. ACM.
- [3] David Coppit and Benjamin Cox. Software plans for separation of concerns. In *Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Lancaster, UK, 22 March 2004.
- [4] Ben Cox. ConcernEditor: A prototype editor for software plans. Undergraduate thesis, The College of William and Mary, Williamsburg, Virginia, May 2004.
- [5] Eclipse.org. The Eclipse homepage. URL: <http://www.eclipse.org/>.
- [6] UCSD Software Evolution Group. Aspect browser homepage. URL: <http://www.cs.ucsd.edu/users/wgg/Software/AB/>.
- [7] Vincent Kruskal. Multiple cross-cutting architectural views. In *A Blast from the Past: Using P-EDIT for Multidimensional Editing*, June 2000.
- [8] Robert R. Painter and David Coppit. A model for software plans. In *Proceedings of the First International Workshop on the Modeling and Analysis of Concerns in Software*, pages 14–8, St. Louis, Missouri, 16 May 2005.