

A Model for Software Plans

Robert R. Painter
The College of William and Mary
Department of Computer Science
Williamsburg, VA
rrpain@cs.wm.edu

David Coppit
The College of William and Mary
Department of Computer Science
Williamsburg, VA
david@coppit.org

ABSTRACT

Even in well-designed software, some concerns can not be easily encapsulated due to their dependence on surrounding context. Such concerns are intermingled with each other and the context code, making it difficult for developers to reason independently about them. We have introduced *software plans* as an editor-based approach for addressing the tangling of context-dependent concerns. Software plans provide programmers with partial views of the overall software which present only that code related to concerns of current interest. The problem we address is that the traditional sequence-of-characters representation for code is poorly suited for software plans. It lacks the ability to accurately model the concerns associated with a code block, the relationships between code blocks, and the notion of multiple independent plans. In this paper, we present a formally-defined code/concern model that supports these capabilities and more. Using this model, we were able to implement a prototype editing tool that supports software plans.

1. INTRODUCTION

Nontrivial software systems are the composition of multiple concerns. Common functional concerns include the overall computation of the software and particular features. There are also more ancillary concerns such as debugging support, performance optimization, and error checking. Following well-known design principles such as separation of concerns [6], information hiding [17], and coupling and cohesion [20], programmers attempt to modularize the system so that concerns can be properly encapsulated.

Such concerns can be both tangled with each other and coupled to surrounding code, making it difficult to decompose the system into modules. As a result, some concerns necessarily span the modules of the system. For example, debugging code is usually present in many modules and is tightly coupled to its context. Attempting to modularize debugging code would violate the principle of high cohesion and low coupling, as the “debug module” would be tightly coupled to many modules of the system and have only loose internal cohesion. In other words, separating highly context-dependent concerns from their context is a difficult problem.

Multiple concerns within a module increases its complexity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop on Modeling and Analysis of Concerns in Software (MACS 2005)
16 May 2005, St. Louis, MO, USA
Copyright 2005 ACM 1-59593-119-8/05/05 ...\$5.00.

This increases the intellectual difficulty of software development, leading to higher probability of programming errors, higher maintenance costs, etc. Importantly, not all concerns must be addressed at all times. For example, debugging code can be safely ignored unless one is debugging the software. Similarly, error-checking code typically interferes with the process of understanding the primary functionality of the code. Today’s software development environments force the programmer to deal with all of the tangled concerns in a module, even when the current programming task only relates to a subset of those concerns.

We have introduced the notion of *software plans* as a possible solution to this problem [5]. Software plans is an editor-based approach for helping programmers cope with tangled concerns in software. Inspired by architectural plans, the approach allows the programmer to implement the software as a composition of multiple semi-independent plans. A plan is an editable view of the software that contains the code related to one or perhaps a few related concerns, along with any plan-specific code that is necessary to provide the concern code with necessary context. Plans significantly reduce the complexity of the software by abstracting away code related to irrelevant concerns, while allowing the programmer to tangle relevant, context-dependent concerns in arbitrarily complex ways.

In order to implement software plans, it is first necessary to develop a model for code that goes beyond the traditional view of the software as a set of files containing a single view of the code. We need a model based on multiple views of the code, that explicitly represents the relationship between concerns and code, as well as relationships between code in different views. In Section 3, we will discuss the requirements for such a model with respect to the design goals of software plans.

This paper has three contributions. First, we present the design goals of software plans as the motivation for the design of our model. Second, we present a formal definition of the model itself. Third, we describe our successful implementation of the model in a prototype editor for software plans.

The rest of this paper is organized as follows. Section 2 presents the notion of software plans in more detail. Section 3 presents our design goals for software plans. Section 4 presents our prototype implementation and preliminary evaluation. Related work is in Section 5. Section 6 describes future work and concludes.

2. SOFTWARE PLANS

Software plans are inspired by architectural plans, where salient concerns of interest are presented along with sufficient contextual information to make them meaningful. If traditional design approaches provide separation of concerns along a modular dimension, we believe that software plans provides separation of con-

```

static void
xfclose (FILE *fp)
{
    /* Allow reading stdin from
     * tty more than once. */
    if (feof (fp))
        clearerr (fp);
}

```

Listing 1: The standard input plan

```

static void
xfclose (FILE *fp)
{
    if (fflush (fp) != 0)
    {
        error (0, errno,
              _("flushing stdout"));
        cleanup ();
        exit (SORT_FAILURE);
    }
}

```

Listing 2: The standard output and error handling plan

```

static void
xfclose (FILE *fp)
{
    if (fclose (fp) != 0)
    {
        error (0, errno,
              _("closing file"));
        cleanup ();
        exit (SORT_FAILURE);
    }
}

```

Listing 3: The normal file and error handling plan

■ File ■ STDIN ■ STDOUT ■ Error Handling

cerns along an orthogonal dimension. The programmer can create multiple plans for a single module, where each plan addresses a different set of concerns of interest. As in architecture, the building of the module requires that the plans be composed and reconciled, to create the final product.

2.1 Terminology

We define a *concern* informally as anything of interest to the programmer. Our definition is intentionally vague to avoid unnecessarily constraining the programmer's flexibility in defining aspects of the code that have some particular importance. In our model, a concern is manifested in software as *concern code*.

A *software plan* is a source code representation consisting of the concern code associated with a set of user-defined concerns, along with any *plan-specific code* necessary for the concern code to be understandable. Plan-specific code exists only in one plan and provides any missing context that is necessary to satisfy conceptual, data, and control dependencies that are important for the correctness and understandability of the plan. In many cases, a software plan is complete enough to be compilable and executable, but this is not a requirement.

The programmer creates the software by developing a number of plans and reconciling their composition. Importantly, any code in one plan can be shared with any other plan, creating links between plans that provide valuable points of reference and ease the reconciliation task. The reconciliation of all plans corresponds to the monolithic source code representation that programmers use today. As a result, software plans are compiler-independent and can be implemented within a programming environment.

2.2 A Simple Example

We now present a motivating example of the use of software plans. We use the `xfclose()` function from GNU `sort`. In the GNU `sort` program, several types of files are closed: standard input, standard output, and normal files. Even though the basic operation is the same, each of the three types of file handles are treated differently. Listings 1, 2, and 3 present three software plans, one for each possible concern-oriented implementation. In each plan, we have indicated the concern code by annotating the line of code with a shaded bar.

This example highlights a number of key aspects of software plans. First, each plan consists of both concern code and shared code. The function bodies are the concern code. The shared code consists of the `xfclose()` type information and outermost brackets. If the programmer wished, he or she could also change the name of the function in a plan, in which case that portion of the code would no longer be shared, and the new name would be plan-specific code.

As with architectural plans, each software plan allows the programmer to think more clearly about the functionality of the `xfclose()` function in terms of each of the three file type concerns. At the same time, the programmer can reason about the impact of concerns such as error handling on each file type concern, independent of the others. And as in architectural plans, the context is shared among the plans and provides a point of reference for reconciling them. That is, the enclosing code is similar to the walls in architectural plans, in that it provides a shared point of reference.

Of course, in order to create a working system involving all three concerns, the programmer would need to reconcile these three plans. The most obvious reconciliation is to test the value of `fp` to determine whether it is `stdin`, `stdout`, or a normal file. This is exactly the strategy used in the actual implementation of `xfclose()`, as shown in Listing 4. In order to create this reconciled plan, the programmer would create a new plan that involves all four concerns. The programmer would then add the context code needed to reconcile the three code blocks that implement the function.

3. A MODEL FOR SOFTWARE PLANS

In order to provide support for software plans, a software development environment must provide a number of capabilities. Here are some of the key features:

- Associate user-defined concerns with code.
- Allow the user to share code blocks between plans.
- Support for plan-specific code blocks.
- Creating new plans should be easy and dynamic. For example, the editor should be able to provide an initial plan based on the plan's set of relevant concerns and existing code in other plans that have already been identified as relating to one or more of those concerns.

The existing model of code as a sequence of characters is poorly suited for software plans. In this section, we present a formal definition of our new model, which provides explicit support for concerns and multiple views. We also describe how this model can be used to implement high-level user-oriented editing operations for software plans.

At a high level, a document is a directed graph in which nodes represent blocks of code, and edges represent the block orders in individual plans. Consider the document graph in Figure 1. The blocks represent individual sequences of text. The edges are labeled with plan names. Starting at the circles, one can follow the edges labeled with the plan name and concatenate the text within the blocks to produce a view for the plan. For example, the view of *Plan 1* consists of the concatenation of text from blocks *A,B,C,D*.

```

static void
xfclose (FILE *fp)
{
    if (fp == stdin)
    {
        /* Allow reading stdin from
         * tty more than once. */
        if (feof (fp))
            clearerr (fp);
    }
    else if (fp == stdout)
    {
        if (fflush (fp) != 0)
        {
            error (0, errno,
                _("flushing stdout"));
            cleanup ();
            exit (SORT_FAILURE);
        }
    }
    else
    {
        if (fclose (fp) != 0)
        {
            error (0, errno,
                _("closing file"));
            cleanup ();
            exit (SORT_FAILURE);
        }
    }
}

```

Listing 4: The tangled `xfclose()` function

File | STDIN | STDOUT | Error Handling

The view of *Plan 3* is an alternate plan consisting of the concatenation of text from blocks *B* and *D*.

All of the text in a block is associated with the same set of concerns and is contiguous in all plans which use that block. As we will describe shortly, editing operations in the view result in transformations on the document graph, splitting, deleting, and creating blocks as necessary to maintain the proper structure. In this way, blocks of code can be shared between plans and can even appear more than once in a plan.

3.1 Formal Definition

We formalized this model in Z [19]. Each Z schema specifies a type in terms of its components and invariants.

First, we define some primitive datatypes. They are abstract at this point. However, the reader can think of *Text* as the set of all code segments, *Concern* as the concerns that a programmer might identify, and *PlanName* as the set of possible plan names. We will use *ID* to distinguish otherwise identical code blocks. (Unlike most programming languages, objects in Z have no identity independent of their state—two objects with the same value are the same object.)

$[Text, Concern, PlanName, ID]$

A *Block* consists of the code text and a finite set of concerns associated with that text.

```

Block
id : ID
span : Text
relatedConcerns : F Concern

```

A *Plan* is a sequence of blocks and a set of concerns. Every plan has a name. The invariant specifies that the id of a block is unique within a plan.

We include the set of related concerns at the plan level so that we can prevent the user from creating two plans with the same set of

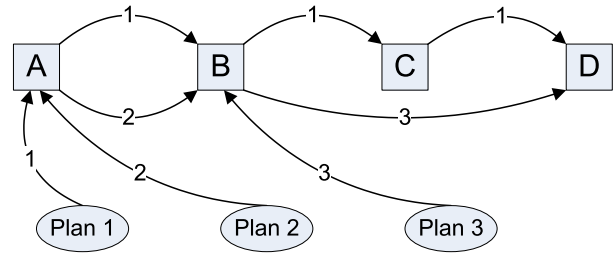


Figure 1: The document graph consisting of three plans and 4 individual code blocks. Views are constructed by concatenating the code blocks in the plan sequence.

related concerns. The idea is that each plan should involve a unique set of concerns, and that the user who wishes to create another plan for the same set of concerns should instead create a new concern that identifies the distinguishing characteristic of the new plan.

```

Plan
planName : PlanName
blockSeq : seq Block
relatedConcerns : F Concern
invariant
forall b1, b2 : ran blockSeq • b1.id = b2.id ⇔ b1 = b2

```

A *Document* consists of a finite set of plans and a finite set of code blocks. The code blocks are not strictly necessary, but are included as a specification and implementation convenience. The first invariant says that the set of blocks is the union of all the blocks in all the plans. The second invariant says all blocks have a unique id within the document.

```

Document
plans : F Plan
blocks : F Block
invariant
blocks = union {p : plans • ran p.blockSeq}
forall b1, b2 : blocks • b1.id = b2.id ⇔ b1 = b2

```

3.2 Model Operations

Next, we describe the set of low-level operations on our model. These low-level operations provide the interface used to implement the user-level operations, described next. In the interest of space, we have omitted trivial operations.

Plan

- *AddBlock*: Given a position and a block, inserts that block into the plan's block sequence at the specified position.
- *DeleteBlock*: Given a position, deletes the block at that position in the plan's block sequence.
- *SplitBlock*: Given a position and two blocks, replaces the block at the specified position with the two blocks.

Document

- *AddBlockToPlan*: Given a block, plan, and position, inserts the block into the document's block set if it does not already exist, and also inserts it into the plan in the specified position.
- *DeleteBlockFromPlan*: Given a position and a plan, removes the block from the position in the plan. No other plans are modified. If the block is no longer used by any plan, it is removed from the document.
- *SplitBlock*: Given a block and a position within the block, the block is replaced with two blocks which are associated with the

same set of concerns and split the original sequence of text at the given position. All plans containing the original block are modified to contain the new split blocks instead.

- *NewPlan*: Given a plan name and a set of concerns, this operation adds a new plan to the document, using the specified concerns as the related concerns. The block sequence for the new plan will contain all blocks from existing plans that are associated with one or more of the specified concerns. The block order in the new plan should, as much as possible, maintain the ordering and proximity of blocks in the existing plans.

3.3 User-Level Operations

We believe the model that we have described allows us to explore the design space for software plans. Consider, for instance, the addition of code in the middle of an existing code block that is shared among plans. One policy might be to update all plans, in which case this user level operation would involve splitting the existing block and then inserting a new block in between the two new halves. Another policy might be to only modify the current plan. In this case, the block would first be replaced with a duplicate, then split before the new block is inserted.

Each of the operations described below operate on the user's view of a plan and are implemented in terms of the model-level operations described in the previous subsection.

- *ComputeView*: Constructs a string for the user to edit, by concatenating the spans in the sequence of blocks for the plan.
- *InsertSpan*: Given a span and a view offset, computes the block that corresponds to the position in the view, splits that block if necessary, then inserts a new block for the given span.
- *DeleteSpan*: Given the start of a span and its length, computes the blocks that correspond to the start and end positions in the view, splits those blocks if necessary, then removes them from the plan.
- *CopyBlocksToClipboard*: Given a view offset and length, computes the blocks that correspond to the start and end positions in the view, splits those blocks if necessary, then copies them to the clipboard.
- *PasteBlocksFromClipboard*: Given a view offset, computes the block that corresponds to the position in the view, splits that block if necessary, then inserts the blocks currently on the clipboard.
- *CutBlocksToClipboard*: Given a view offset and length, performs a *CopyBlocksToClipboard* operation followed by a *DeleteSpan* operation.

The clipboard operations are vital because they allow the programmer to share code among plans. This greatly eases the reconciliation of plans, because the editor can infer the relative ordering of code in different plans, and can avoid duplicating code that exists in multiple plans.

4. IMPLEMENTATION

To help validate our model, we used it to implement a prototype editor for software plans. While limited space precludes a detailed discussion of the application, we were able to create an editor that allows the user to create new concerns and plans and to edit those plans independently. Using the intuitive copy and paste operations, the user can share code between plans. Edits to the plan automatically and transparently update the underlying document structure.

Perhaps most surprising was the difficulty of automatically generating a good initial sequence of code blocks for a new plan. The heuristic algorithm we used computes a proximity score for every

block that follows another in a plan. The closer a block B is to a block A, or the more it appears after block A in the plans of the system, the higher the score. We then construct a sequence of all relevant blocks by attempting to maximize the overall score.

Another issue that became apparent during the implementation was the need for detecting and automatically merging code blocks whenever possible. Without merging, the splitting of blocks caused by the editing operations can result in a large number of blocks, degrading performance. Overall, the model seems to have enough flexibility to allow us to explore these issues.

5. RELATED WORK

Other works propose alternate methods to manage and separate concerns, ranging from code analysis to the creation of new programming paradigms. In this section, we will discuss past research done in this area and relate it to our research.

5.1 Views in Software Engineering

The notion of multiple views in software engineering is not new. For example, program slicing [18, 21] attempts to reduce the complexity of code by extracting only those lines of code that can alter, or are altered by, the value of a particular variable. Similarly, partial evaluation [10, 11] creates a specialized version of a program based on given values of a subset of the inputs. In both cases, the resulting code is a working program that is similar to a plan, insofar as it is a complete, compilable subset of the original code. However, the key difference is that these techniques are useful for understanding existing code, rather than providing a concern-oriented view of code as it is being developed.

More generally, the notion of multiple views of software have been used in a number of different software engineering settings. For example, Garlan's research on software development environments [8] used views on a common database of program information. Each view provided the tool an appropriate representation of the shared data, while still enabling tight collaboration among tools. While the notion of multiple views is prevalent in many fields of software engineering (and indeed computer science), relatively little work has been done on independent development and integration at the source code level.

5.2 Language-Based Solutions

Most of the work on tangled concerns has focused on the development of new programming language abstractions for separating and encapsulating them. The most widely-known of these is Aspect-oriented programming (AOP) [13], which augments the programming language with the *aspect* abstraction. Aspects are meant to encapsulate state and computation related to cross-cutting concerns and to provide a high-level definition of how the aspect should integrate with the base code. For example, AspectJ [12] provides declarative pointcut definitions for join points such as the access or modification of an attribute, or the call or return from a method.

While it is possible to use aspects to perform fine-grained tangling of concerns, doing so requires the programmer to be especially clever about the use of features of languages such as AspectJ. More importantly, this approach violates coupling and cohesion for concern code fragments that are highly context-dependent and loosely related to each other. This view is supported by recent research [1, 4, 15] that critiques the capabilities and general suitability of the aspect composition models. Some researchers have also suggested the utility of concern-oriented views of a program [7, 2], a notion we are pursuing in our software plans research.

5.3 Tools Supporting Partial Views of Code

A number of tools have been developed to hide portions of code in order to highlight relevant code. One approach taken by many IDEs is “code folding” or “code elision” in which blocks of code can be collapsed. Other tools use programmer metadata to identify collapsible code. For example, P-Edit [14] allows the user to associate a Boolean expression with each line of code. Given an “effective mask” of Boolean values, the editor can dynamically generate a version of the code that contains only those lines whose expressions are satisfied by the values in the mask.

A number of concern-oriented tools have also been developed, using various means of identifying concern code and removing irrelevant code. The Aspect Browser [9], for example, uses lexical techniques to find matching code and display it in a single view, and provides tools for navigating to the different program locations. In other work, Newman [16] suggests the use of semantic information about the code in order to create “localizing views,” and Black and Jones [3] use an abstract program structure to create “perspectives” on the code.

Each of these approaches has a common feature: they try to reduce or modify the way an existing program is viewed. The reduced complexity of the viewed code is the main advantage. Our work differs from these in that we seek to provide tool support for integrating separately developed plans. Our tool and approach seeks to change the way that programmers think about code, so that multiple reconciled plans is the fundamental method of developing software, rather than a single monolithic representation.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented a document model that explicitly supports software plans by representing the code as a graph of code segments having additional concern information. We have presented a formal definition of the model and described how it is used to implement key software plans editing operations.

We believe that the model provides a flexible basis for exploring and evaluating the notion of software plans. In the future, we plan to implement our model as part of a plugin for the Eclipse programming platform. We also plan to explore techniques for improving the user-level operations, and ultimately to evaluate the software plans approach in terms of a number of case studies.

7. ACKNOWLEDGMENTS

The authors thank Justin Manweiler for his initial implementation of the concern/code model, and Meghan Revelle for comments on an earlier draft.

8. REFERENCES

- [1] J. Aldrich. Challenge problems for separation of concerns. In *OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Oct. 2000.
- [2] E. L. Baniassad, G. C. Murphy, C. Schwanninger, and M. Kircher. Where are programmers faced with concerns? In *OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Oct. 2000.
- [3] A. P. Black and M. P. Jones. The case for multiple views. In *ICSE 2004 Workshop on Directions in Software Engineering Environments*, May 2004.
- [4] L. Carver and W. G. Griswold. Sorting out concerns. In *OOPSLA '99 Workshop on Multi-Dimensional Separation of Concerns*, Nov. 1999.
- [5] D. Coppit and B. 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 Mar. 2004.
- [6] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [7] E. Ernst. Separation of concerns and then what? In *ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*, Finland, 2000.
- [8] D. Garlan. Views for tools in integrated environments. In *Advanced Programming Environments. Proceedings of an International Workshop*, pages 314–43, Trondheim, Norway, 16–18 June 1986. Springer-Verlag.
- [9] U. S. E. Group. Aspect browser homepage. URL: <http://www.cs.ucsd.edu/users/wgg/Software/AB/>.
- [10] R. Heldal and J. Hughes. Partial evaluation and separate compilation. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–11, Amsterdam, 12–13 June 1997. ACM SIGPLAN.
- [11] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, Sept. 1996.
- [12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. 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.
- [13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.
- [14] V. Kruskal. Multiple cross-cutting architectural views. In *A Blast from the Past: Using P-EDIT for Multidimensional Editing*, June 2000.
- [15] G. C. Murphy, A. Lai, R. J. Walker, and M. 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.
- [16] E. Newman. Localizing views for separation of concerns. In *ICSE 2001 Workshop on Advanced Separation of Concerns in Software Engineering*, 15 May 2001.
- [17] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–8, Dec. 1972.
- [18] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glueck, and P. Thiemann, editors, *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, pages 409–429, Schloss Dagstuhl, Wadern, Germany, 12–16 Feb. 1996. Springer-Verlag, New York, NY. URL: <http://www.cs.wisc.edu/wpis/papers/dagstuhl96.ps>.
- [19] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, Hertfordshire, UK, 2nd edition, 1992.
- [20] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–39, 1974.
- [21] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–7, 1984.