

# Programming With Crosscutting Effective Views

Doug Janzen and Kris De Volder

University of British Columbia  
201-2366 Main Mall  
Vancouver, BC Canada  
{dsjanzen, kdvolder}@cs.ubc.ca

**Abstract.** Aspect-oriented systems claim to improve modularity by providing explicit mechanisms that allow modularization of concerns which crosscut the object-oriented decomposition of a system in terms of classes. However, by modularizing concerns which crosscut classes, at the same time the structure and functionality associated with the classes themselves becomes scattered across the implementation of different aspects. This may hamper system understanding in other ways. In this paper we present a system that addresses this issue by allowing a developer to move fluidly between two alternative modular views on the decomposition of the program, editing the program either as decomposed into classes, or alternatively as decomposed into modules that crosscut classes. Thus developers gain the advantages of open classes, without having to give up the ability to edit the program directly in terms of classes.

## 1 Introduction

Aspect-oriented software development [1] addresses the issue of crosscutting concerns and how they can be more cleanly modularized and dealt with by developers. There are many different approaches towards this goal. What all the approaches have in common is that they attempt to provide explicit representations of crosscutting structure in software.

Language based approaches provide programming language extensions such as open classes, aspects, pointcuts and advice that allow concerns which cut across classes to be captured modularly. Some examples of this are systems like AspectJ [2], Caesar [3], and Aspectual Collaborations [4].

Tool-based approaches on the other hand make crosscutting structure explicit by constructing views on top of the code. In this way tools can make crosscutting structure which is implicit in the code explicitly visible and accessible to the developer. Examples of tool-based approaches are FEAT [5], AJDT [6], JQuery [7], AspectBrowser [8] and Stellation [9]. These tools can be loosely divided into two camps. On the one hand there are tools which work on legacy OO systems and try to show how implicit crosscutting concerns exist within the object-oriented program (examples: FEAT, JQuery, AspectBrowser, Stellation). On the other hand there are tools which try to do exactly the opposite: they try to produce views that recover the object-oriented structure of the program which has been

made implicit by the introduction of aspect-oriented features in the language (example: AJDT for AspectJ).

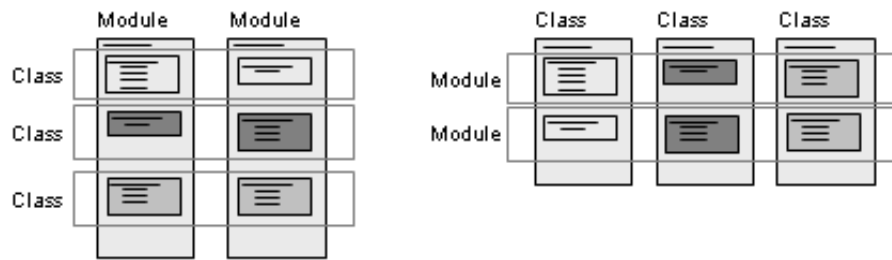
In both cases the fact that one of the views is explicit in the code and the other views are generated by tools implies a different level of support for working with those views. The view which has been made explicit in the code can be edited directly by editing the code. We call such a view an *effective* view. For example, for all programming languages which are based on textual syntax and the use of source files to store programs, the source code of a program represents an effective view, because editing the source code directly affects program structure. On the other hand, the views produced by most tools are non-effective because typically these views cannot be edited in such a way that the actual program structure is affected by the edits to the view. Instead, the typical usage profile for tool-based crosscutting views is that they serve as documentation or navigational aids for developers. But in order to make effective changes to program structure developers must edit the actual source code.

In this paper we present the Decal prototype. Decal is a tool that lets developers work simultaneously with two mutually crosscutting and *effective* source code views. Decal is atypical in that the views it produces are not derived from source code, they *are* source code. In some sense, Decal reverses the roles of tool-based views and source code as found in most tools. A typical tool produces a view *based on* source code. Decal maintains an internal, structured representation of the program and, from this, views are generated *in the form of* source code.

The current prototype is limited to two crosscutting views only but this could easily be extended. In section 7 we will discuss some possible extensions. For now we limit ourselves to a brief introduction to the two views supported by the current prototype.

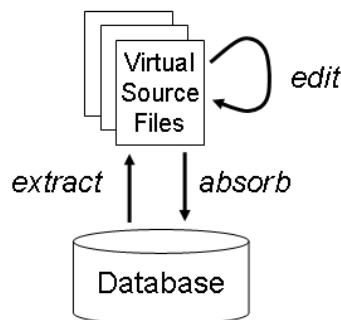
The first of the two views is called the *modules view*. This view provides a decomposition of program structure in terms of modular units which crosscut classes in a way that is similar to open classes. The second view is called the *classes view* and is a more traditional object-oriented view, showing a decomposition of the program structure in terms of classes. The classes view crosscuts the modules view in a similar way that the modules view crosscuts the classes view. Every element in the program belongs simultaneously to both the classes view as well as the modules view. Both the modules view and the classes view are effective and editing the textual representation of either view impacts program structure. Consequently, when changes are made to one view, these changes are also automatically reflected in the other view. Figure 1 depicts this mutually crosscutting relationship between the two views.

To accomplish this bi-directional causal connection between the two views, Decal views are represented as *virtual source files* (VSFs). The term “virtual source file” was introduced in Stellation [9]. VSFs are dynamically generated ASCII files which can be browsed and edited by the developer with commonly available text editors. To be able to generate the VSFs dynamically and have changes from one view be reflected in the other view, Decal internally maintains



**Fig. 1.** The modules view crosscuts classes and the classes view crosscuts modules.

a single common representation of the program structure. After editing a VSF, the developer can commit the file back to Decal. The system will then translate the changes the developer made into corresponding changes to the common representation. At this point, the VSF is “absorbed” back into the system and ceases to exist. This extract-edit-absorb cycle is depicted in Figure 2.



**Fig. 2.** The Extract-Edit-Absorb Cycle

Both the modules view and the classes view in Decal are, on their own, similar to conventional decompositions of program structure into source files. The classes view provides an effective, textual view similar to that of traditional object-oriented source files. The modules view provides an effective, textual view similar to that of a language that supports open classes. The contribution of this paper is to show that these mutually crosscutting effective views can be supported simultaneously.

An additional contribution this paper makes is to identify a number of issues and problems that are specific to a system that supports crosscutting effective views. We believe that the insights gained from the design and implementation

of Decal, and the ways in which we have addressed them are valuable for the future design and implementation of similar systems.

The rest of this paper is structured as follows. In the following section, we present a motivating example. This example illustrates why it is desirable to provide an effective view for classes as well as modules at the same time. Section 3 presents the Decal system and its two different views. Section 4 describes the Decal database and issues surrounding its design. Section 5 looks at issues that arise when editing virtual source files. Section 6 describes the implementation of Decal. Section 7 discusses how the ideas explored in our prototype might be applied to a more realistic language. The last three sections describe future and related work and end with some concluding remarks.

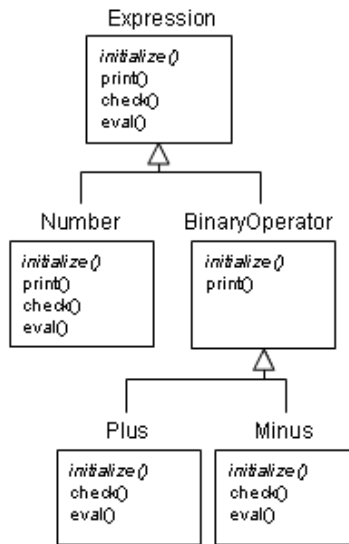
## 2 Motivating Example

The example presented here is derived from a similar example in Tarr et. al. [10]. What the example tries to illustrate is that no matter how well designed a program may be and no matter how carefully developers may have considered the decomposition of their program to anticipate future evolution, the choice of any decomposition makes an implicit tradeoff, making certain tasks easier at the expense of making other tasks harder.

The example is phrased in the context of an implementation for an environment for working with programmatic expressions. The example is strongly simplified for presentation purposes. The class diagram in Figure 3 depicts a simplified version of the subsystem for representing expression abstract syntax trees. It depicts the most natural way to represent an AST in terms of a hierarchy of classes. This is also most naturally mapped onto an implementation in an object-oriented language such as Java, mapping every “class box” in the diagram to a compilation unit in the implementation language.

This natural object-oriented decomposition makes it hard to add new features to the system that require the implementation of new operations on AST structures. For example, suppose we wanted to extend our implementation with a pretty-printing feature. This would require the addition of one or more printing related methods to most, if not all, of the AST classes. Good designers may have anticipated the need for such extensions and included a Visitor pattern [11] in their design. The Visitor pattern comes at a price however because it introduces additional complexity into the program. In many ways the Visitor pattern is just a way of reifying a more procedural abstraction as an object/class, so that it can be represented with the available decomposition mechanism of classes.

Some languages, such as MultiJava [12] and AspectJ [2] provide the concept of open classes which leads to more flexibility in this regard. With open classes it is possible to declare methods outside of the class that they “belong to”. Thus a developer may simply add additional operations on AST structures in a separate module and need not modify the original source code of the AST classes. With open classes, one may achieve a decomposition which has a modular structure similar to that of the Visitor pattern, but without the complications of



**Fig. 3.** Most natural OO decomposition for a simple AST implementation

implementing procedural abstractions in terms of closed classes. This situation is depicted in Figure 4.

Unfortunately, even the extra flexibility of open classes ultimately is not sufficient. When developers choose to structure modules around certain operations such as printing, type checking, etc. they are gaining more ease to maintain these types of abstractions, but at the same time they inevitably make it harder to do other types of extensions because the implementation of classes is scattered across multiple modules. This makes it harder to perform tasks that more naturally align with the class structure of the program. For example, suppose a future extension requires adding a new type of expression node to the AST. This task will be complicated because it will require making additions to several modules to implement all the required functionality for the new class, such as printing, type checking, etc.

Decal addresses this problem by letting developers alternate between editing the program in the modules view or the classes view. In the following sections we will explain each view in terms of the example from this section. At the end of the section we will show how the classes view eases the extensions of the AST implementation with new classes, while at the same time program structure is decomposed in terms of modules that crosscut those classes.

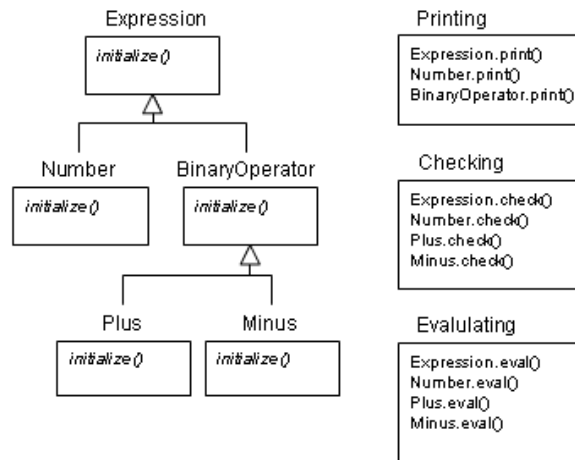


Fig. 4. A decomposition of a simple AST implementation with open classes

### 3 The Decal System

At the core of the Decal system is a simple object-oriented language with support for open classes. To keep the prototype lightweight and suitable for quick exploratory experiments we wanted to keep the language as simple as possible. The Decal language therefore supports only a minimal subset of object-oriented features. Specifically, it supports single inheritance with the ability to override operations, but does not support advanced features such as interfaces, overloading, static members and constructors<sup>1</sup>.

To this basic object-oriented core language, Decal adds support for a simple form of open classes (but does not support multiple dispatch as in MultiJava). We consider open classes the simplest “flavor” of aspect-oriented programming<sup>2</sup>. Our current prototype does not support more advanced aspect-oriented language features such as pointcuts and advice.

The addition of more advanced object-oriented features would make the design and implementation of several components of the Decal system technically harder, but we do not believe it should pose fundamental problems. The situation is not as clear-cut for adding additional aspect-oriented features such as

<sup>1</sup> We assume reliance on ordinary methods and programming conventions for initializing objects as is also the case in, for example, Smalltalk[13]

<sup>2</sup> Some people might argue that open classes belong in the camp of traditional object-oriented languages because open classes do not include a mechanism of implicit invocation. What set of features makes a language aspect-oriented (or object-oriented for that matter) can of course be debated. Our view is that they are a simple form of aspect-orientation because they explicitly provide a mechanism to support a form of crosscutting modular structure.

support for pointcuts and advice. In Section 7 we discuss how the choices made in designing and simplifying the Decal language may affect the generalizability of our approach both in terms of object-oriented as well as aspect-oriented features.

### 3.1 The Modules View

The modules view in Decal allows programs to be edited according to a modular structure that may crosscut classes. Declarations related to a particular class may be spread across multiple modules, and each module may contain declarations related to more than one class. Figure 5 shows the `ast` module from our running example. It contains the “bare bones” declaration of several classes to represent a simple AST structure.

```
module ast {  
  
    public class Expression {  
    }  
  
    public class Number extends Expression {  
        public int value;  
        public void initialize(int value) {:  
            this.value = value;  
        }:  
    }  
  
    public class BinaryOperator extends Expression {  
        public Expression left;  
        public Expression right;  
        public String op;  
        public void initialize(Expression left, String op, Expression right) {:  
            this.left = left;  
            this.op = op;  
            this.right = right;  
        }:  
    }  
  
    public class Plus extends BinaryOperator {  
    }  
  
    public class Minus extends BinaryOperator {  
    }  
  
}
```

Fig. 5. Example VSF: The `ast` module

Decal modules are a means to group declarations related to a specific concern. Each declaration can be marked `public` to indicate that it is visible outside the module or `private` to indicate that it is visible only within the same module.

Besides declaring classes of its own, a module may also introduce additional declarations into classes that are publicly defined in other modules. Figure 6 shows a `printing` module that provides an additional printing-related operation for the classes defined in the `ast` module. As with open classes, using the modules view in Decal allows programmers to work with concerns that crosscut the class structure in a modular way. All code related to the printing concern are brought together in the `printing` module. Similarly, it is possible to add operations related to semantic checking functionality to all the AST classes in a separate module VSF as well (this is not shown to save space).

```
module printing {  
  
    import ast;  
  
    Expression {  
        public void print(Printer out);  
    }  
  
    Number {  
        print {:  
            out.printInt(value);  
        :}  
    }  
  
    BinaryOperator {  
        print {:  
            left.print(out);  
            out.printString(op);  
            right.print(out);  
        :}  
    }  
}
```

**Fig. 6.** Example VSF: The `printing` Module

Decal is similar to Java minus a number of features such as interfaces, static members, overloading and constructors. Besides these simplifications, there are a few other differences that are worth mentioning.

First, Decal makes a distinction between “operations” and “methods”. We will have more to say on the reason for this distinction when we discuss program representation in Section 4. For now we merely explain the nature of the difference.

An operation represents an operational abstraction. For example, the `printing` module provides a declaration for an operation called `print` on `Expression`. A method declaration on the other hand provides a specific implementation for an operation on a specific class. A method declaration provides the method body that is to be executed in the event of an invocation of the operation on some specific type of object. Therefore, generally an operation is implemented by one or more method declarations. Methods are dispatched at run time according to the standard single dispatch semantics of overriding and inheritance.

In our example the `print` operation is implemented by two methods, for the `Number` and `BinaryOperator` classes respectively. Note that an operation declaration is where the signature of the operation resides, but that it is not repeated in the method declaration which only requires the name of the operation to identify it unambiguously (since there is no overloading in Decal). An alternative VSF syntax might redundantly repeat the operation signature with the method declaration, for the sake of readability. However, in this version, the syntax was designed to resemble the database representation as closely as possible and redundant information in the syntax is factored out, just as it is in the database representation.

### 3.2 The Classes View

The classes view in Decal contains the same declarations that are present in the modules view except that they are organized according to class membership rather than module membership. A VSF can be generated for each class in the system. Each VSF contains all the declarations related to the chosen class grouped in blocks according to which module they are declared in. Figure 7 shows what the VSF looks like for the `Number` class from our example.

Note that in conventional languages the boundaries of information hiding are typically defined in terms of lexical position. In Decal, where every declaration occurs simultaneously in two textual locations, a class VSF and a module VSF, this alignment of scope with textual nesting only really happens in the modules view. This is because we can think of the modules view as the “primary” view, which is most like the only textual view one would get if Decal was a conventional programming system. The classes view is conceived of as a secondary, derived view which is generated from the primary view. However, because both views are mutually effective this distinction between primary and secondary view is not all that clear. Scoping rules however, is one way in which there is still a clear qualitative difference between the nature of the two views. Classes, unlike modules, do not define a name space in Decal and do not have import statements. The names referred to in the code in the class VSF in Figure 7 therefore are to be interpreted in relation to the modules they belong to and to their respective import statements. For example, the `print` method and the `value` field are defined in the same class. However this alone is not sufficient to make a reference like `this.value` legal. It is further required that the name `value` can be resolved in terms of the import statements of the `printing` module (as it was shown in Figure 6). Similarly, it would be legal for other modules to declare additional

```
public class ast.Number extends ast.Expression {

    module ast {
        public int value;
        public void initialize(int value) {
            this.value = value;
        }
    }

    module printing {
        print {
            out.printInt(value);
        }
    }

    module checking {
        check {
            ...
        }
    }

    module evaluating {
        eval {
            ...
        }
    }
}
```

**Fig. 7.** Example VSF: The Number class

fields called `value` on the `Number` class. These fields would be considered as distinct. This is especially useful for private fields, providing modules with a safe way to add additional state to classes without having to worry about accidental name conflicts with extensions defined by other modules on the same class.

### 3.3 Motivating Example Revisited

Now let us return to the motivating example again. The explanations in the preceding sections already show how the modules view allows extending classes with new functionality, for example to modularize behavior such as printing, or checking across multiple classes. Let us now examine how the classes view aids in extending the system with an additional AST class, even when the system has been decomposed around behavior oriented modules as shown above.

For example, assume we needed to add a new class `UnaryOperator`. This can be conveniently accomplished in the classes view. Indeed, it can be done in a way very similar to what we might do if the program was written in a style similar to the most natural class-based decomposition shown in Figure 3. What we can do is — inspired by the assumption that `UnaryOperator` is most similar to `BinaryOperator` — extract the `BinaryOperator` class VSF and use it as a template. We create a new VSF for `UnaryOperator` and copy-paste the contents of the `BinaryOperator` class into the new VSF. Then we edit it at will into what we need. This works very well, because the `BinaryOperator` class will already have all the right pieces of functionality from various crosscutting modules in place, telling us exactly what functionality we also need to implement for a `UnaryOperator` and providing us with some convenient code snippets to base our implementation on. The task of adding a new AST class would be significantly harder in the modules view. This is true even in a fancy development environment that provides some additional non-effective tool-based views that show a view recovering the class structure. This is so because although a non-effective class view will help by telling us what we need to do, and where we need to do it, we would still need to add snippets of code in multiple different modules across the system. In Decal we get spared from this because we can directly edit the classes view and even meaningfully copy-paste code from one class VSF into another class VSF.

We can extend this example a little further. Assume that after considering the copy-pasting solution, we do not like the amount of code duplication that was introduced as a result. So we want to refactor this code and introduce a common superclass to capture the similarity between unary and binary operator expressions. This refactoring task is also more easily carried out on the class-based decomposition. We can do the refactoring by working with only three class VSFs, the class VSF for the new class, plus VSFs for the two classes to be refactored. It would be significantly harder to do this refactoring in the modules view because the functionality to be factored out into the superclass is scattered across many modules.

## 4 The Decal Database

To support multiple crosscutting effective views Decal uses a database as a single common representation of the program, from which all views are generated on-demand. We have chosen an RDBMS over other alternatives, not for any particularly significant reason, except perhaps that it is the most standard and commonly available type of database. In this section we provide some information about the design of the Decal program database.

### 4.1 Developing a Schema

The database schema of Decal is basically a fairly straightforward mapping of the structure of Decal’s modules view into database tables. Some aspects of the schema design are non-obvious and, we believe, key to making Decal work. We refrain from discussing all the details of the schema but provide some information here about some key issues and design decisions.

**Granularity** The core object-oriented structure of the database is made up of tables to represent classes, fields, operations, methods and the relationships between them. A key choice that we made in designing this part of the schema was to store method bodies as blocks of text, rather than explicitly representing every statement and expression as a separate fact. What is necessary for generating Decal VSFs is the relationship between high-level declarations, not the details of the AST structure of method bodies. A further motivation for this choice was the experience of the OMEGA project [14] which reported serious performance problems due to the number of queries required to generate the text of even simple programs.

**Object IDs** The database representation makes an explicit distinction between the identity of an entity (module, class, operation, method, etc.) and its name. This is important because it is possible that several distinct entities exist that have the same name. For example multiple classes called `TraversalState` might be defined as helper classes in different modules for printing, checking etc. in the AST example. For convenience and efficiency (to avoid performing name lookup each time), references to program entities in database tables are represented in terms of some globally defined non-ambiguous object IDs (OIDs).

**Soft keys** Another key feature of the schema design is the notion of “soft keys”. In order to support incremental name resolution and error tolerance, an extra level of indirection was introduced to represent references that use names. For such references, rather than storing a direct link to an OID, a so called “soft-key” is stored. A soft key points to a table entry that stores what information is needed to resolve the reference and caches the OID of its target once it becomes resolved.

Soft keys provide a form of intentional name capture. Even if a reference can be resolved immediately, the target of the reference may later be deleted or renamed. By retaining the name used to resolve a reference, Decal can ensure that the reference can be resolved and unresolved many times as the program evolves while retaining, to some degree, the original intention of the programmer.

## 4.2 Impact of Schema Design on the Language

It is interesting to note that the design of Decal's database schema, has had an impact on the language design. In particular, in the process of designing the schema for the representation of method declarations we realized that information about method signatures would be represented redundantly. Standard normalization practices in database design suggest that this information should be factored out into an additional table. Refactoring the database tables in this fashion, we also decided to make the language itself reify the notion of an operation explicitly. Though we don't think this is essential to make our approach work, it leads to a cleaner correspondence between the database schema and the syntax of the language. As a result, the implementation of generation/absorption of VSFs from/to the database is more straightforward.

## 5 A Practical Editing System

In this section we will look at Decal's editing system. When we say editing system we mean more than just a tool for editing the text of a VSF. The editing system includes everything required to support the extract-edit-absorb cycle. The editing system of Decal, which is characterized by a complete separation between the editing format and storage format of programs, introduces some new issues that do not arise in traditional programming systems, where the editing format and the storage format are one and the same. In the following subsections we elaborate on some of those issues and how we have addressed them in Decal.

### 5.1 Editing Semantics

In a conventional programming system, where programs are stored as text files, and where developers can edit the stored program representation directly, the meaning of edits is unambiguously defined. However, when the text being edited is part of a virtual source file the meaning of the edits may not be as clear.

An important design decision that we made is that declarations can belong to only one module. For a brief moment we deliberated about supporting overlapping modules, so that one piece of information (e.g. a method declaration) would be represented simultaneously in multiple modules. We found this idea very appealing, because it would make it possible to support modules that not only crosscut classes, but also modules that crosscut each other. However, this raised the issue that the editing semantics of module VSFs are no longer intuitively unambiguous. For example, if a declaration can belong to more than

one module VSF at the same time then it is not clear when a developer deletes a declaration if the intent is to delete the declaration from this module only, or from the system as a whole. Assuming that declarations belong to only one module naturally gives VSFs an editing semantics very similar to regular source files. I.e. it is intuitively clear that deleting declarations *must* remove them from the program altogether.

We call the property that a piece of information is represented in at most one VSF within a particular view, the *disjointness* property. It is a desirable property because it allows an editing semantics of VSFs which is intuitive, in that it is designed to behave similarly to what would be expected of “real” source files. Note that the converse property, completeness — that every a piece of information is represented in at least one VSF — is not as significant in defining the semantics of edits. It does however affect the usefulness of a view since it determines *what* information can be manipulated through it. This is not to say that incomplete views are necessarily less useful (they are more abstract in a sense).

In the case of ambiguity, it is of course possible to explicitly assign a semantics one way or another, or to design a mechanism to let developers explicitly state their intent. However it should be clear that the design of an editing semantics that would feel intuitive to developers becomes significantly more difficult in the absence of the disjointness property. Note that a similar problem of confusion around editing semantics does not arise because of overlapping VSFs that are in different views. The fact that the semantics of an edit is defined in one view automatically defines what it means in the other view as well.

We believe that studying mechanisms to deal with ambiguity in editing semantics is an interesting and important topic for future research. Indeed, resolving this issue is a necessary condition for supporting a more complex set of aspect-oriented language features, as will be discussed in Section 7.2. However we considered it outside the scope of the current paper.

## 5.2 Name Resolution

The generation of multiple views requires that the database provides the necessary connections to determine what VSFs (module or class) a specific element belongs to. The availability of that information in the database is dependent on the successful resolution of names. For example, in Figure 6 the identifier “Number” refers to the class `Number` as defined in the `ast` module. To generate the class VSF for class `Number` the system must resolve all references to `Number` from all the modules that add members to that class. This dependence on name resolution implies that in a system that supports multiple views, name resolution needs to be done as early as possible. Therefore, in Decal name resolution takes place each time a changed VSF is saved, rather than at compile time as is usual in traditional languages.

### 5.3 Error Tolerance

An additional complication arises because an incremental name resolution mechanism, as described above, needs to be tolerant of errors and incompleteness. During the process of constructing a program developers often leave parts of the program in an inconsistent state. This is not due to bad programming, but is simply a consequence of the fact that programmers can only do one thing at a time. It is often convenient for a programmer to work on one part of a program and refer to elements that he or she intends to define later on. Or sometimes a developer may wish to delete several parts of a program that contain references to each other. Thus it is highly impractical to force all names to be resolved before a VSF can be saved.

As mentioned earlier, we use a representation for name references that we call a soft key. Soft keys retain all the information needed to resolve a reference. This includes the name used to make the reference and the module from which the reference is being made. If the reference can be resolved then the OID of the target is also stored. As elements are added to the database, unresolved soft keys are checked to see if they refer to the new element. When an element is deleted, soft keys that point to the element have their OID removed, but retain all the information needed to resolve the reference again as new elements are created. The use of soft keys gives Decal VSFs a similar kind of flexibility that regular source files have while at the same time allowing the kinds of queries that depend on direct cross-referencing information.

Another place where tolerance of errors becomes an issue is when checking the syntax of VSFs at the time they are saved. It would be impractical to expect developers to remove all syntax errors just to be able to save their work. However some degree of syntactic correctness is necessary for Decal to be able to map regions of text back to the database. To minimize the impact of this requirement Decal does not force method bodies to be syntactically correct before they are saved. This can be easily supported thanks to our choice to store method bodies as blocks of text. Also, we have designed Decal's method declaration to have its body delimiters easy to recognize, by adding an additional colon next to the outer braces. This makes it easy to identify the textual area of the body without having to parse what is inside them.

### 5.4 Temporal Continuity of Views

In a conventional programming system, it is trivially true that a file saved at one time, when revisited at some later time, will look identical. However, in a system based on VSFs, this property does not automatically hold. The exact layout of a VSF depends on how it is generated. Since it is regenerated on each successive visit, the VSF layout may differ each time. We call this the issue of "temporal continuity of views".

The fact that the contents of a VSF may change between visits is a necessary property of the system, because it is what allows changes in one view to be reflected onto the other view as well. However, changes in a VSF between

visits also have the potential for causing disorientation. This is because there is typically a lot of freedom in the use of indentation, whitespace and the order of declarations. This freedom is actively used by developers. It is therefore desirable that VSFs should look and feel as much as possible like “real” files, preserving not just the semantic content, but also non-semantically meaningful attributes of their textual layout.

In the current implementation of Decal we did not spend a great deal of effort on this issue. Our choice of representation for method bodies ensures that their formatting is retained in the database exactly as the programmer typed them, including comments. We also provide limited support for preserving comments outside of method bodies. Each element in Decal can have a Javadoc style comment attached to it that is stored with the element in the database. Other kinds of comments, such as single line comments, are not allowed outside method bodies as it is not always possible to determine which element they should be attached to in the database.

The order of declarations and white space between declarations are not currently retained. Instead we settled for maintaining a consistent use of white space and ordering (alphabetic). This guarantees some continuity: there will not be huge discrepancies in layout unless substantial edits are performed. Nevertheless, this implementation is clearly far from optimal since it takes away most of the freedom that developers have in conventional text-based editing systems in constructively using the layout of the source text.

Even in the limited setting where we have used Decal for toy examples, we found the “jumping text” behavior (on saves) to be disorienting. Therefore we believe that temporal view continuity is important for practical usability. We also believe it is technically feasible to make VSFs feel nearly the same as real files, by storing more information about their layout in the database. A good solution is not trivial to implement and requires detailed consideration about what additional information to store, but we believe it is mostly a technical implementation issue.

## 6 A Prototype Implementation

The user interface for our prototype is implemented as an Eclipse [15] plugin. Users can browse the contents of the database using two simple views that show the inheritance hierarchy of classes and the list of modules. Selecting a class or module from one of these views opens the corresponding VSF in the editor.

At the core of Decal is an API for manipulating Decal programs. This API uses a relational database called HSQLDB [16] to actually store the program information. HSQLDB is a lightweight embeddable database that can be distributed as part of Decal thus eliminating the need for users to install and maintain a full database management system. Its use of JDBC would make it relatively easy to switch to a more industrial strength database in the future.

The generation of virtual source files is built as a layer on top of the core API. This layer generates text and maps changes back to the database independently

of how the text is edited. This makes it easier to integrate Decal into a variety of programming environments.

The prototype is still in an experimental stage and is not yet available to the general public. We are currently developing a third view whose VSFs correspond to proper Java source code suitable for compilation. Since Decal's name resolution mechanism is incompatible with Java packages all VSFs in this view are written to a single directory and a name mangling scheme is used to avoid name collisions. Thus it is not a VSF that can be edited and saved back to the database, but will allow us to compile and run programs written in Decal with a standard Java compiler.

Similarly we have done only a small amount of exploratory work on type checking Decal programs. Although type checking such a system is not a trivial issue, we did not think the implementation of the type checker would be a substantial contribution compared to other languages that support open classes, such as MultiJava and AspectJ. For this reason we decided to defer this work and focus on crosscutting effective views.

## 7 Supporting More Realistic Languages

The Decal prototype shows how multiple views can be implemented on a simplified language, but to be of practical value, our methods must be applicable to more realistic languages. Adding support for standard OO features would be a good first step, but the real power of multiple views is in their application to aspect-oriented languages.

### 7.1 Adding More OO Features to Decal

To extend Decal to include more advanced OO features such as interfaces, static members, constructors, overloading etc, the first thing that must be done is to design a more extensive database schema to represent the additional features. This is mostly a technical issue not a fundamental one.

Second, the additional features make the design of the Decal module system more complex. MultiJava has already proven that it is possible to support open classes in a clean way in conjunction with a rich set of core object-oriented language features, adding both open classes and multi-methods to Java in a way that is backwards compatible. Thus, the design of the module system is technically complex, but doesn't present any unsolvable problems.

Third, we have to consider potential complications that might arise with respect to editing semantics. Assuming that we design the language from the point of view of the modules view and consider the classes view as secondary, then the editing semantics of the modules view is automatically clearly defined. We also do not foresee any real complications with editing semantics of the generated classes view because adding conventional OO features only introduces new or more elaborate declarations that can be associated with a specific location in the class structure of the program. Therefore, they would not introduce information

overlap within the classes view. As discussed before, the disjointness of different VSFs within a view implies that editing semantics can be unambiguously and intuitively defined to mimic the semantics of real source files.

## 7.2 Adding More Aspect-Oriented Features

Besides considering the extension of Decal to support a more complete set of conventional object-oriented features, we may also wish to consider extending its arsenal of features to express crosscutting in the language. For example, we may wish to include mechanisms for pointcuts and advice instead of just the simpler notion of open classes.

Presumably, the modules view would be augmented to show the system from an aspect-oriented point of view. Whereas the classes view would show where the aspects apply in terms of the class-based decomposition.

We believe this will pose more fundamental challenges than the addition of object-oriented features because the expressiveness of pointcuts and advice make the connection between the two views fundamentally more complicated. Intuitively, open classes provide only a very simple form of crosscutting, in the sense that any declaration in the modules view can be interpreted as applying to a single location in the classes view. This property, which guarantees an easy translation between the two views, is destroyed by the introduction of an expressive pointcut language. Pointcuts allow the application of advice to a large number of locations in the classes view, or even dynamically, in terms of properties that cannot necessarily be associated directly with static locations at all. This raises some issues with the generation of the classes view, such as how to represent dynamic advice. Additionally, independent of its dynamic nature, the fact that advice may apply to many locations in terms of the classes view destroys the disjointness property of the classes view: a single advice body may occur in multiple places in the classes view. This implies that we will have to tackle some complications with the editing semantics: for example, what does it mean to edit, add or delete advice from within the classes view?

## 8 Future Work

We believe that aspect-oriented languages which offer advanced features for expressing crosscutting structure would be exactly where support for multiple effective views would pay off the most. We believe that tackling the issues raised above, which complicate the design of the editing semantics is feasible. However it is a non-trivial problem which requires more research.

Another possible extension is to support additional effective views beyond the two that are currently supported. A natural view to add would be one that shows all the methods that implement a particular operation. Given that the concept of operations are already explicitly reified in the Decal language and also explicitly represented in the Decal program database, the addition of this view would be a very straightforward but potentially useful extension.

There are numerous other views which might be worth exploring. For example the authors of this paper have frequently been frustrated with the fact that object-oriented inheritance makes it hard to get a complete view of a class, scattering its implementation over multiple superclasses. It would be possible to address this issue by providing an additional object-oriented view, in which the contents of superclasses is “unfolded” into the VSF of its subclasses. In fact, we consider it an interesting idea to explore different forms of this kind of “unfolding” or “expanding” of a VSF to include source code from “semantic neighbors”. For example, we think this could be a very effective metaphor for user interface support that lets developers dynamically tailor and grow a VSF file to include exactly what they need to see and edit for a task. Simple versions of this idea are implemented in the Visual Studio [17] IDE, although in Visual Studio, understandably it is only possible to unfold/fold code within a single source file. So it can be used to elide details within a file, but not expand the current textual view into entities residing in different files. However, in a system where source files are virtual, and boundaries between files become much less real, exciting new possibilities for extending this simple but powerful idea present themselves.

Besides these somewhat futuristic ideas, there are also some more pressing needs to empirically validate the current ideas and approach in terms of a more realistic language and realistic code. Our next step therefore will likely be to explore how to extend the current implementation with a sufficiently rich set of OO features, so that it is possible to import legacy Java code into the system and conduct some experiments on realistic code.

## 9 Related Work

The main contribution of this paper is to show how it is possible to construct an editing system that lets a developer alternate between editing a system through either one of two mutually crosscutting, effective views. This work is most closely related to tool-based aspect-oriented approaches that support the creation of crosscutting views on top of source code. Decal distinguishes itself from most of these other tools in that it produces *effective* textual views. In contrast, most other tools provide views which are non-effective. In such tools the main utility of the view is to serve as documentation and to support navigation. Some examples of tools which fit this description are AspectBrowser [8], JQuery [7], FEAT [5] and AJDT [6]. Of these tools, AJDT is the only one that is in the same camp as Decal, generating views that help developers to recover object-oriented structure which has been made implicit by the introduction of more aspect-like modularity in the language.

While the majority of tools provide only non-effective views, there are some notable exceptions. The oldest one is probably MasterScope in the Interlisp [18] development environment. In MasterScope, a developer may request the generation of a textual view by writing a query that identifies what declarations in the program are of interest. The declarations which match the query are returned in

a textual, editable view and can then be edited as a group. A similar mechanism was provided more recently by the Stellation [9] system from which we adopted the idea of VSFs. In both cases, the views are textual and effective. The main difference is that in Decal, views are first class and have a well-defined intuitive semantics for all possible edit operations, whereas in Stellation and Interlisp, the views are arbitrary groupings of declarations and the editing system does not attach a specific semantics to the grouping of elements into a view. This has consequences for the semantics of additions and deletions of declarations to/from views which — in Stellation or MasterScope — don't have a clearly defined and intuitive editing semantics. Our example provided in section 3.3 illustrates the importance of this difference. The example relied on copying and pasting declarations between different class VSFs in order to effectively move or copy them from one class to another. This requires that insertion and deletion of declarations has the appropriate semantics with respect to the classes view in which these operations take place. Consequently, as far as the authors understand, this example would not work in either Stellation or MasterScope.

The *Smalltalk Envy* [19] programming environment is similar to Decal in a number of ways. The role of source code in Smalltalk differs from most other programming systems and is similar to Decal in that program structure is not stored as source code but in a more structured format, as a coarse grained object-oriented data structure. Smalltalk Envy also has a mechanism for packaging of applications, similar to Decal modules. The main difference is that Envy does not provide textual editable views of classes or application packages as a whole. Instead, views are created by, and manipulated through different GUI source code browsers. These browsers provide some level of effectiveness by providing refactoring and restructuring commands in menus. Decal on the other hand preserves the “illusion” of source files and lets developers edit source file views of classes or modules as a whole. We believe the GUI browser approach has advantages as well as disadvantages over a VSF-based editing metaphor. Moreover, they are complementary, in the sense that GUI browsers can be added to an editing environment based around VSFs, just as many modern IDEs provide browsers for real source files. Some advantages of GUI based refactoring and restructuring commands is that they allow developers to express their intent more clearly. This is one way of resolving ambiguity in editing semantics. Nevertheless, we believe it is hard to design and implement a “complete” set of GUI tools to support the refactoring-type of editing operations. Textual manipulation of source code seems to be still what developers always need to revert to when specific refactoring operations are not supported directly in the GUI. This is also true for Envy. Although the lower-level text editing operations do not capture the intent of the developer directly, all edits can be accomplished by copying, pasting and editing text. Moreover, editing programs in terms of source files is what the majority of developers are already most familiar with.

Intentional Programming (IP) [20], like Decal also uses a rich data structure to store programs and provides different kinds of views to edit this structure. However IP views can be graphical as well as textual. Furthermore, its main

purpose is to facilitate language extensions of various kinds, not the creation of crosscutting views (though this might be a potential application of IP). The complexity of coordinating the interactions between several language extensions and visual editors make IP a much more ambitious project than ours. The Decal approach consequently is more lightweight. It uses a much coarser grained representation of programs, and requires only limited support from some fairly simple and cheap tools: a standard SQL engine, standard text editors and some simple VSF parsers and VSF code generators.

HyperJ [10] and its notion of multi-dimensional separation of concerns are closely related. In HyperJ’s terminology, Decal could be characterized as “a system that supports two orthogonal dimensions of program decomposition simultaneously”. Decal modules are similar to HyperSlices. The main differences are the following. First of all, Decal modules are true modules and have true support for encapsulation and information hiding (public and private), whereas HyperSlices have no mechanisms to hide internal structure<sup>3</sup>. Second, HyperJ focuses more on the composition of HyperSlices rather than the generation of views from existing program structure. To this end HyperJ provides a very complex mechanism of composition rules. In contrast, the composition rules for modules in Decal are embodied by a straightforward mechanism of name binding.

The concept of mixin layers [21] proposed by Smaragdakis and Batory are similar to Decal modules. The mixin layers approach focusses primarily on the complexity of composition of families of systems whereas our focus is not on the composition of modules into multiple systems but rather on supporting editing a single system simultaneously from multiple points of view. It is noteworthy that in a recent paper [22] Batory et. al. report they implemented and used an “unmixin” tool that allows a system composed from mixin layers to be edited directly, and the edits to be mapped back into the corresponding mixin layers. They report that the ability to edit from both a composed and uncomposed point of view was tremendously useful. They do not however provide any details or insights about issues such as editing semantics, temporal continuity of views, etc.

We used MultiJava’s open classes and AspectJ’s inter-type declarations as models for designing our module system. Both languages support more advanced kinds of modularity than just open classes. MultiJava includes support for multi-methods and AspectJ supports pointcuts and advice. We have already discussed in Section 7 what the possible implications of adding such features to Decal are.

The Decal extract-edit absorb cycle depicted in Figure 2 bears a remarkable similarity to the idea of roundtrip engineering as supported by many CASE tools [23, 24]. The main difference, as we see it, between Decal and such roundtrip engineering tools is that Decal transforms between (textual) representations that are both at the same level of abstraction – implementation – whereas CASE tools

---

<sup>3</sup> HyperJ supports public and private to be used in HyperSlices, but these relate to the class structure and have no bearing on whether something is visible or not to another HyperSlice composed with it.

transform between representations at different levels of abstraction, for example connecting design to implementation or architecture to design.

In his work on trying to define a theoretical framework for Automated Roundtrip Engineering (ARE) [25], Assmann has proposed a theoretical definition of an ARE system and a classification of AOP systems in the context of roundtrip engineering systems. In Assmann's terminology, Decal is *not* an ARE system, i.e. it does not provide a means to automatically generate an unweaver from a weaver. However, the Decal editing system's extract and absorb functions could be interpreted as hand crafted instances of what Assmann calls a bi-directional weaver, or "Beaver". Although Assmann provides good arguments why such bi-directionality is desirable, he does not provide a clear insight on how it can be achieved in practice.

Decal has some similarities with integrated environments as proposed by Garlan [26] and Herrmann and Mizini [27] where a common data structure is shared by a collection of tools. In these systems each tool is given its own view of the common data structure and is also able to keep its own data structures that are unique to its operation. Such integrated environments could be used to produce a variety of effective views, including something like Decal. However their focus is on the problems of tool coordination and integration, and they do not address the specific problems of working with virtual source files such as editing semantics.

## 10 Conclusion

We have presented the Decal system, an unconventional programming system which provides two mutually crosscutting, effective, textual views. One view, called the modules view lets developers decompose and edit program structure in a way similar to open classes. The second view, called the classes view, presents a more traditional object-oriented decomposition of the system into classes. Developers can alternate arbitrarily between the two views. Both views are presented as a collection of virtual source files (VSFs) and can be edited by developers to effect changes to the system. Both the modules view and the classes view in Decal are, on their own, similar to conventional decompositions of program structure into source files. The main contribution of this paper is to show that both views can be supported simultaneously, as effective views on the same program.

The design and implementation of a system which simultaneously supports several textual, crosscutting, effective views poses some unique challenges. Little or no work exists in the literature on how such systems should be designed and implemented. Thus, another contribution this paper makes is to identify some of the issues that arise in the design and implementation of such a system, and to describe specific principles and techniques we have used to address those issues.

Below, we provide a brief overview of the most important issues that were touched upon in different parts of the paper, and summarize what has been done in the context of Decal to address those issues.

**Program Representation** A system that supports multiple views must perform early name resolution, be tolerant of errors, and have clearly defined editing semantics. Some form of a database is necessary to allow for efficient querying of program information. The Decal database uses a representation that is tolerant of unresolved references while at the same time allows efficient querying.

**Editing Semantics** In a system where the storage format and the editing format of programs are separated, it becomes an issue to define the meaning of edits in terms of changes to the stored representation of the program. In Decal this issue is addressed by respecting the disjointness property which makes it possible to define editing semantics that mimic those of “real” source files.

**Temporal Continuity of Views** In a system where source files are generated on the fly the layout of the generated text can change between editing sessions, resulting in a sense of disorientation for the developer. We believe it is technically feasible to design VSF generators that produce stable views by adding explicit information to the database about the textual structure of VSF files at the moment they are saved. We have not implemented this in the current version of Decal but leave it as a topic for future research.

**Incrementality and Tolerance of Errors** To be able to generate one view from another a minimum of information about the structure of the program needs to be updated incrementally as developers edit VSFs. To support the typical editing patterns of developers this process must support a certain amount of error tolerance. When saving a VSF Decal tolerates syntax errors in method bodies by parsing and storing them as single tokens. Decal is also tolerant of unresolved names and supports incremental name resolution through the mechanism of soft keys in its database representation.

While the issues and solutions presented here stem only from the experience in designing and implementing a single prototype, and therefore do not necessarily generalize to all systems, we believe that our experience recorded in this paper will prove a valuable starting point for the design of other systems that support multiple crosscutting effective views simultaneously.

## 11 Acknowledgments

This work was supported in part by an IBM Eclipse Innovation Grant and the University of British Columbia. We thank Gregor Kiczales, Brian de Alwis and Mik Kersten for their valuable comments, insights and stimulating discussions which have greatly contributed to this paper.

## References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proc. of European Conference on

- Object-Oriented Programming (ECOOP). Volume 1241 of Lecture Notes in Computer Science., Springer Verlag (1997) 220–242
2. Kiczales, G., Hilsdale, E., Huginim, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In Knudsen, J.L., ed.: European Conference on Object-Oriented Programming. (2001) 327–353
  3. Mezini, M., Ostermann, K.: Conquering aspects with caesar. In: Proceedings of the 2nd international conference on Aspect-oriented software development, ACM Press (2003) 90–99
  4. Lieberherr K., L.D., J, O.: Aspectual collaborations – combining modules and aspects. *The Computer Journal* **46** (2003) 542–565
  5. Robillard, M.P., Murphy, G.C.: Concern graphs: finding and describing concerns using structural program dependencies. In: Proceedings of the 24th international conference on Software engineering, ACM Press (2002) 406–416
  6. AJDT: Aspectj development tools website. <http://www.eclipse.org/ajdt/> (2003)
  7. Janzen, D., De Volder, K.: Navigating and querying code without getting lost. In: Proceedings of the 2nd international conference on Aspect-oriented software development, ACM Press (2003) 178–187
  8. W.G. Griswold, Y.K., Yuan, J.: Aspect browser: Tool support for managing dispersed aspects. In: First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems - OOPSLA 99). (1999)
  9. Chu-Carroll, M.C., Wright, J., Shield, D.: Aspect-oriented programming: Supporting aggregation in fine grained software configuration management. In: Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering, ACM (2002) 99–108
  10. Tarr, P.L., Ossher, H., Harrison, W.H., Jr., S.M.S.: N degrees of separation: Multi-dimensional separation of concerns. In: International Conference on Software Engineering. (1999) 107–119
  11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns*. Addison Wesley (1995)
  12. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: Multijava: modular open classes and symmetric multiple dispatch for java. In: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press (2000) 130–145
  13. Goldberg, A., Robson, D.: *Smalltalk-80: The Language and its Implementation*. Addison Wesley (1983)
  14. Linton, M.A.: Implementing relational views of programs. In: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments. (1984) 132–140
  15. IBM: Eclipse website. <http://www.eclipse.org/> (2003)
  16. HSQLDB: Hsql database engine. <http://hsqldb.sourceforge.net/> (2003)
  17. Microsoft: Visual studio. <http://msdn.microsoft.com/vstudio/> (2003)
  18. W., T., L., M.: The interlisp programming environment. *IEEE Computer* **14** (1981) 25–33
  19. Object Technology International Inc.: ENVY/Developer R3.01. (1995)
  20. Simonyi, C.: The death of computer languages, the birth of intentional programming (1995)
  21. Smaragdakis, Y., Batory, D.: Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.* **11** (2002) 215–255

22. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. In: Proceedings of the 25th international conference on Software engineering, IEEE Computer Society (2003) 187–197
23. Borland: Together CASE tool. <http://www.borland.com/together/> (2003)
24. IBM: Rational software. <http://www.ibm.com/software/rational/> (2003)
25. Aßmann, U.: Automatic Roundtrip Engineering. In Aßmann, U., Pulvermüller, E., Cointe, P., Bouraquadi, N., Cointe, I., eds.: Proceedings of Software Composition (SC) – Workshop at ETAPS 2003. Volume 82 of Electronic Notes in Theoretical Computer Science (ENTCS), Warsaw, Elsevier (2003)
26. Garlan, D.: Views for tools in integrated environments. In: An international workshop on Advanced programming environments, Springer-Verlag (1986) 314–343
27. Herrmann, S., Mezini, M.: Pirol: a case study for multidimensional separation of concerns in software engineering environments. In: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press (2000) 188–207