The Case for Multiple Views

Andrew P. Black and Mark P. Jones

Department of Computer Science & Engineering, OGI School of Science & Engineering, Oregon Health & Science University

{black, jones}@cse.ogi.edu

Abstract

We argue that viewing a program as a linear sequence of symbols on paper or on a display is outmoded and unnecessarily restrictive. Instead, programs should be regarded as complex multi-dimensional artifacts on which linear text provides but one possible view. Freeing ourselves from these restrictions is very difficult: it requires not only a modest amount of new technology, but also a qualitative change in the way that programmers think.

However, the potential benefits are enormous. Multiple views make it easier to understand complex programs, and provide a unifying framework for many common program transformations. In addition, Multiple views provide a solution for language designers trying to choose between competing alternatives: provide the advantages of both, but in different views.

1. Introduction

Since the dawn of computing, programs have been thought of as linear text, stored first on paper tape and then on magnetic media, and presented to programmers in the form of a "listing". We believe that this one-dimensional view of programs is completely inadequate to represent the complex structure of a large program. Instead, we are seeking to apply the power of information technology to the process of programming itself, using computers to present programs in a far richer form. The long-term goal of our research is to change the way that programs are written, read, maintained, modified, and evolved.

Our approach is based on the realization that people understand a complex artifact more quickly and more easily if they can see *multiple views* of the artifact. An architect seeking to describe the design of a building to a client does not give the client a single drawing: the architect provides site models, building models, plans, elevations, perspectives and interior views. Each of these views shows some aspect of the proposed building to best advantage, while ignoring or glossing over details that are irrelevant to that aspect. The same glossed-over details may take center stage in some other view.

Programs, particularly large programs that have evolved over time and have passed through many hands, are every bit as complicated as large buildings. Indeed, modern software systems may well be the most complex artifacts that humankind has ever created. Chosen judiciously, multiple views can be a powerful aid to mastering this complexity.

We are working to develop both practical tools and the supporting theory that is necessary for constructing and using multiple views. We are focusing first on the coarse structure of object-oriented programs, by which we mean the structure of the program above the level of methods: we focus on using multiple views to elucidate the varied and complex ways in which methods, classes and state interact, rather than to show the internal structure of individual methods.

Our research methodology combines experimentation and theory: we are building prototype tools that can be evaluated through use, but the tools will be supported by a consistent conceptual framework that rests upon a sound theoretical base.

As our prototypes have evolved, we have found an additional and unexpected benefit of multiple views: they provide a way of avoiding many of the thorny political decisions that plague a language design. This includes somewhat superficial decisions, such as whether a language should use keywords rather than symbols, or parentheses, rather than spaces to delimit parameters. Instead of laying down the law in these matters, these features can be represented in different ways in different views. With proper tool support, much deeper questions, such as whether typing should be explicit or implicit, strong or soft, can also be un-asked.

To date, we have been focusing our efforts in two areas: the implementation of a multiple-view browser for Java, and the structuring of classes using traits.

1.1. A Multiple-view Browser for Java

The Java language treats programs as the composition of a set of classes, and most editors and development environments expect the programmer to view a whole class at a time. This is similar to the way that a building is a composed of several floors, and a set of architect's plans lets us view one floor at a time. However, we have found that it is also valuable to treat a Java program as if it were composed

To be presented at the *Workshop on Directions in Software Engineering Environments*, ICSE 2004, Edinburgh

of *generic operations* that are defined on objects of many different classes using case discrimination. The generic operation view has similar properties to elevations of and sections through a building: the elevations and sections contain the same information as the plans, but highlight relationships that would otherwise be hidden. Thus, if our programming task involves adding operations to a family of related classes, the generic operations view is much more useful than the class-by-class view because the latter does not reveal the complex relationships between the methods in different classes. Other programming tasks may be assisted by flattening the inheritance hierarchy and showing all of the methods that are understood by instances of a class, no matter whether they are inherited or implemented locally.

We are targeting this work on Java because of its growing importance as a commercial programming language. Many vendors offer integrated development environments (IDEs) for writing and maintaining Java programs. However, all of the IDEs of which we are aware compel the programmer to adopt a file-by-file or class-by-class view, which does not always match the programmer's task.

1.2. Structuring a Class using *Traits*

Classes are more than collections of methods and fields; they frequently have a rich internal structure. Some aspects of that structure, such as inter-method dependencies, can be inferred, as is done by Lanza's Blueprints [9]. However, other kinds of structuring, such as logical grouping of methods, should be expressible by the programmer. We are working to develop a theory of class structure, and programming tools that exploit this theory to display the structure of a class using multiple views. These views may be placed on a continuum: at one extreme is the *flat view*, which shows no internal structure at all above the level of field and method. At the other extreme is the *fully structured view*, with fields and methods being grouped into semantically coherent components that are nested inside each other and conjoined to create higher-level components, which in turn are used to build yet higher level components and, eventually, whole classes. The conventional view—and the *only* view supported by common languages and tools—shows a class as comprising exactly three components: its own fields, its own methods, and its superclass. This view lies on our continuum, but is not the *only* useful view between the extremes.

Our theory breaks classes into small components that we call *traits*. A trait represents a unit of reusability smaller than a class; classes are built from traits, but the same trait can be reused in many classes.We have built a Smalltalk browser that supports the decomposition of classes into traits [2], and we have started to use traits on sizable programs [1]. Because of the compositional properties of the trait operators, traits provide a natural framework for viewing classes in multiple ways: the flat view, the fully structured view, and many intermediate views can all be expressed using traits. We have also defined a variety of operators that can be used to modify traits and combine them into larger traits and into classes.

Traits were first defined and implemented in the context of Squeak Smalltalk, an excellent experimental platform that enables rapid development and evaluation of different versions of the trait algebra and supporting programming tools. However, to maximize our influence on practising programmers, we are in the process of developing a similar model for Java. We also plan to enhance the multiple-view browser so that it supports the decomposition of Java classes into traits, and allows this decomposition to be viewed at different levels of aggregation.

Several features of Java—notably types, overloading and the absence of a metaclass hierarchy—raise significant issues that the Smalltalk version of traits did not need to address. We are working to develop the theory of traits to overcome these problems and provide Java programmers with the same benefits that we have begun to enjoy when we program with traits in Smalltalk.

2. Background

The idea behind a software engineering environment that provides multiple views is not new. Multiple views have been explored in requirements engineering (*e.g.*,[12]), and across the software life-cycle (see Scott Meyers' survey [11]). The Desert environment built by Steve Reiss and his students at Brown [14, 15]—focussing as it does on programming rather than other aspects of the life-cycle, and on presenting all the views in a single tool—approaches our ideal more closely, although Desert still treats programs as text objects that are stored in files. Theme-based literate programming [7] is based on a similar ideal. Our focus on treating a program as a higher-level abstraction, of which any given view is just that—a view—draws on several additional sources, which we now describe.

2.1. Refactoring

A refactoring is a source-to-source transformation on a program that preserves its execution semantics [5, 13]. The purpose of a refactoring is to improve the structure of a program, for example, to remove duplication or to make it easier to perform a modification. An example of a refactoring is *abstract field*, which replaces all direct accesses to a field by calls to accessor ('get' and 'set') methods. Another refactoring, enabled by abstract field, is *replace eager by lazy initialization*. The inverse operations are also refactorings; neither form of the program is absolutely "better" than the other, but may be more appropriate for a specific purpose. Refactoring is essential if code quality is to be maintained as a program evolves. The Refactoring Browser [16], a tool for automating the refactoring of Smalltalk programs, showed that tools encourage refactoring and reduce the risk of introducing errors. Several com-

mercial refactoring tools are now available for Java.

As we have said, refactoring is classically thought of as a *transformation* between two programs. The point of departure of our research was the realization that each refactoring can also be thought of as defining an *equivalence relation* on programs. Thus, a refactoring partitions the space of all programs into equivalence classes. If we shift our attention from the individual elements of these equivalence classes to the classes themselves, we have raised the level of abstraction of the programming process.

How are these equivalence classes to be presented to the programmer? How can we focus on the essence of the equivalence class—the common behavior of the witness programs—and ignore the superficial differences? We believe that the answer is to provide multiple views on the equivalence class, that is, to represent it by different witness programs depending on the current needs of the programmer, and to make it possible to switch almost effortlessly from one view to another.

2.2. The Tyranny of the Dominant Decomposition

Different programming languages come with different mechanisms to support programmers in the task of structuring the code for large applications. In each case, the goal is always to provide language constructs or features that will allow complex programs to be arranged into small, manageable pieces that can be designed, understood, implemented, and reused as independent building blocks. Unfortunately, these same structuring tools can also make it harder to add or change features of a program that cut across its structure. Empowering programmers to work with multiple views simultaneously would solve these problems.

In Java, for example, programs are organized into packages, each of which contains classes. Sometimes, useful functionality can be added to a program by defining a new class, and making minimal changes elsewhere; such changes fit very naturally into the package decomposition. However, there are many changes that do not fit. For instance, adding a new method to a hierarchy of classes, or adding a parameter to an existing method, will typically require changes to multiple classes, and perhaps even multiple packages.

The problem here is not that the decomposition of Java programs is weak or inadequate. Rather, the problem is that we currently have no choice but to work with a *single view* of the code, with little flexibility or support for other decompositions. Any system or language that imposes a single view of structure on programmers—what Ossher and Tarr have aptly referred to as the *tyranny of the dominant decomposition* [19]—will cause similar problems. Providing *multiple views* of the code is the way to break free from tyranny.

2.3. Design Patterns and Programming Idioms

Once the conflict between different forms of program decomposition has been identified, our first instinct may be to "program around it." For example, the Visitor Pattern [6] may be viewed as an idiomatic response to the need to define operations in a language in which the class decomposition is primary. The effect of the Visitor Pattern is to translate the cumbersome task of adding a new operation into the easier task of adding a new class. The solution is ingenious, but hardly easy to read, even after the pattern has been identified, described in a catalog, and added to university programming curricula. More sophisticated patterns can be applied in situations where they are needed, but there is always a danger that the extra code that is introduced to support them might ultimately become more a part of the problem than the solution. For example, a more general version of the Visitor Pattern [8] supports extensibility in *both* the operation *and* the class dimensions, but requires the implementation and maintenance of a complex programming protocol.

Like refactoring, the introduction of a design pattern or an idiom is a way for programmers to transform their code to fit the decomposition paradigm of their chosen language. These techniques have demonstrated their potential in many real-world projects, but they do have limits. For example, by refusing to step outside the host language, we must sometimes adopt encodings that reduce performance and, more importantly, make code harder to read. In addition, even with the help of a programming environment, it is hard for a programmer to move rapidly between different views of a program. What the programmer conceptualizes as a single change, such as switching from an operation-based to a class-based view, may require many small steps. Because they can represent alternate decompositions of the program as views, the environments that we envision avoid the need to resort to design patterns. For example, if the tools allow the programmer to see the operation view directly, then there is no need to use the visitor pattern to represent an operation as a class.

Note that we are not saying that patterns are bad. On the contrary, they are an effective response to commonly-occurring problems, given the constraint of a fixed programming language. However, once such a problem has been identified, it makes sense to solve it directly in our programming tools, rather than forcing the original programmer, and all of the maintenance programmers that follow, to resort to idioms.

2.4. Multidimensional Separation of Concerns Hyper/J

Harold Ossher and Peri Tarr have developed an approach to software composition and decomposition that is based on a multi-dimensional separation of concerns [19], and have prototyped this approach in Hyper/J, a tool for Java developers. With Hyper/J, the programmer constructs a

"concern matrix" that maps each primitive unit in a given program — such as individual variable or method definition — to a particular point in a user-specified coordinate space. An additional "hypermodule" input to Hyper/J is used to describe how different views of a program can be projected out by giving the dimensions or regions of the coordinate space that are of most interest.

Hyper/J and its underlying foundations are an important experiment with more flexible notions of program representation. Our work takes some of these ideas to the next level, focusing more directly on interactive features for constructing and switching between different views, emphasizing that the multiple views are of equal importance, and incorporating a broader range of automated techniques for classification and analysis of code.

3. Towards Multiple Views

As we explained in the introduction, we have been exploring the application of the multiview concept to Object-oriented programs in Smalltalk and Java.

3.1. Sweet

As a first experiment, we implemented a tool called Sweet, a "Static weaver and editing tool". This prototype is designed as a tool for object-oriented program development using Java, although the underlying ideas could be adapted to other settings. The objective of Sweet is to support more flexible decomposition and construction of Java programs than is permitted by conventional Java compilers. Specifically, Sweet gives programmers the opportunity to organize, arrange, and group fragments of code in a way that reflects the problem that is being solved or the algorithm that is being used

Sweet is a practically motivated tool, but it represents a radical new approach to program construction that we have not seen elsewhere. In the input to Sweet, a programmer blends definitions of points in an abstract program structure—each of which might correspond, for example, to a package, class, or method—with descriptions of the executable content that should be placed within that structure. The difference between writing programs with conventional Java tools and writing programs with Sweet is analogous to the difference between printing an image on a piece of paper or drawing it by hand. To print an image, we must convert it into a collection of pixels and then send data for each one to the printer in a specific order. Typically, neither the decomposition into pixels or the sequencing of pixel data has any direct relation to the objects depicted in the image. Moreover, the pixels corresponding to individual objects in the image will often end up scattered across the output pixel stream, concealing the relationship between them until the whole image has been rendered. By contrast, when we draw a picture by hand, we typically work at a much higher and more natural level, painting complete objects and proceeding in an order that is determined by the image itself and not by the orientation or size of the paper on which it is drawn. Only the most abstract of artists would consider drawing an image in the way that the printer does!

With the current, batch-oriented implementation of Sweet, a program is written as a sequence of declarations, the text of which may be spread across several files. As the Sweet tool processes each declaration, it builds up a representation of a complete program in an internal data structure referred to as a *repository*. These declarations should be thought of, not as the code for a particular program, but as instructions that describe how that particular program might be constructed. As such, Sweet provides a primitive form of meta-programming, although it differs from much of the previous work in that area where the focus has been on mechanisms for programming in the small. In its current form, Sweet provides only one view on the repository, using the repository contents to generate a set of conventional Java source files that can be compiled using standard tools.

We have already used this prototype in experiments to build several non-trivial programs including a compiler for a "mini-Java" language that consists of over 60 classes, and a reusable library for dependency analysis. In the case of the compiler, an existing Java program was refactored by hand so that it could be rewritten in a more natural way. In contrast, the dependency analysis library was developed from scratch using Sweet.

The syntax of Sweet has also been designed to reduce the need for tedious boilerplate, duplication of information and tangling of datatype definitions with the implementations of associated operations. For example, Sweet provides a compact notation for describing class hierarchies, and allows uses of the visitor pattern to be described directly at the level of source code, rather than forcing the programmer to use an encoding. There is not space here to give a realistic example of Sweet code, but the following vignette illustrates the benefits in miniature.

```
public abstract class List {
        public case Nil
        public case Cons(private int x, private List xs)
}
public int length()
        case List abstract;
        case Nil  {return 0;}
        case Cons {return 1+xs.length();
}
```

The first code fragment defines a hierarchy of three classes: an abstract superclass List and concrete subclasses Nil and Cons representing empty and non-empty lists. The second fragment defines the generic operation length on lists, collecting together in one place the definitions of the length method for each of the classes. Sweet would output exactly the same Java program if the input were presented to it in the ordinary Java style as three separate classes, each containing the appropriate length method. In some

cases, and for some purposes, this version of the program might be preferred, but in others the code shown above is preferable because it highlights more clearly the relationship between the classes, collects the definitions for a single conceptual unit—the length operation—into a single location; and emphasizes the separation between these two aspects of the program. However, the important point is that Sweet gives programmers the opportunity to *choose among these options* (and others), whereas conventional compilers and IDEs support only a single view.

3.2. Traits

Traits have been developed through a collaboration with IAM/UniBern, which was initiated in the autumn of 2001 while the first author (Black) was on sabbatical at the Software Composition Group of the University of Bern. Because classes can be decomposed into traits, and because this decomposition can be viewed at any level of nesting without any change in semantics, traits provide a rich set of alternate views on a class.

Stripped to its essentials, a trait is a first-class collection of named methods. Methods in a trait must be *pure behavior*; this means that they cannot directly reference any fields, although they can do so indirectly. The purpose of a trait is to be composed into other traits and eventually into classes. A trait itself has no superclass; if the keyword **super** is used in a trait, **super** is treated as a parameter that becomes bound when the trait is eventually used in a class. A more complete description of traits can be found in the ICSE conference proceedings [2].

Because traits contain pure behavior, and because of the richness and careful design of the combinators, traits provide the advantages of multiple inheritance and mixins without the associated complexity. Moreover, it is always possible to flatten a program containing traits into an equivalent program that uses only conventional classes. Similarly, traits that are composed from other traits can be *flattened*—can have this internal structure removed—without changing their semantics. Flattening makes traits very useful for providing multiple views on complex class libraries because the programmer can choose to show or hide detail as required by the task at hand.

4. The Role of Language

As we have described in section 2.1, the realization that multi-view programming could be a powerful tool came from an appreciation of the way that refactoring can help a programmer to understand legacy code. However, because refactorings are source-to-source transformations, the power of refactoring is limited by the expressive power of the programming language. That is, the only views of the program that can be obtained through refactoring are those that can be expressed in the source language!

To see this more clearly, imagine that you are modifying a compiler written in Java and working with a deep inheritance hierarchy that represents expressions; it contains subclasses for terms, factors, literals, variables, and so on. Imagine further that you are trying to implement compile-time evaluation of constant expressions, but that something is not working. You need to examine the code, but you do not want to see all of the methods in all of the expression sub-classes: you want to focus on the evaluateConstant methods. In fact, you want to think of evaluateConstant as a *single generic operation* defined by cases on all of the subclasses. However, no amount of refactoring in Java can provide you with this view, because the Java language cannot express the idea of a generic operation: all it can express are individual methods and classes. What you need is an extended notation like that of Sweet (see Section 3.1).

Thus, in order to provide some of the views that we have found to be useful for particular tasks, we must first extend the underlying language to permit a wider range of expression. The extended language makes possible more refactorings; these refactorings define more equivalences, and thus more ways to view the program. The key point is that we are not proposing language extensions that increase the power of the language to communicate with the computer: we are proposing extensions that enable programmers to view existing programs more directly, in ways that clarifies their meaning. Thus, although accomplishing our aims requires extending the base programming language, developing language extensions is for us a means to an end and not an end in itself.

Traits provide a good example of a language feature designed in this vein. Because traits do not bind **super**, any composite built from traits can also be built as a flat collection of methods. Thus, it is simple to provide several semantically equivalent views of a composite class, at several levels of decomposition. (This is not true for mixins.) Similarly, the alias mechanism in traits was designed so that its effect could be understood entirely at the class level: the internals of the aliased method are not affected. This would not be true of a rename operation.

5. Ongoing Research

In the MultiView project we are investigating the practicality and the impact of multiple-view programming environments. As we have hinted above, the work involves language design and theory as well as tool-building and evaluation.

Our experience with Sweet has been encouraging, but has also revealed some significant limitations and weaknesses, which indicate that an integrated development environment (IDE) is *essential* if we are to realize our objective of programming with multiple views. Our current work is thus focused on the following three tasks.

5.1. Building a Browser supporting multiple *simultaneous* views.

While Sweet permits great flexibility in the way that programs are described, it supports this by batch *translation* from the original source code view to the Java output view. As such, Sweet still provides programmers with only a single view of their code at a given time; it does not allow them to move easily between different views. We plan to construct an interactive environment that will support multiple views simultaneously, and that will enable programmers to move easily from one view to another. This activity will extend in time through the life of the project, as the results of other activities are incorporated into this environment.

To see how this environment might work, let us assume that the user starts with the source code for a compiler, structured as a conventional Java program: a hierarchy of packages, classes, and methods. This view of the program is represented by the tree structure shown on the left of Figure 1. Let us further assume that the user wishes to work with a different view that organizes the code according to its function in the compiler; this might result in a presentation of the program with a different tree-like structure, in which the code fragments concerned with lexical analysis, with parsing, with type checking, and with code generation are each organized into different branches, as shown on the right of Figure 1.

To define this new view, the programmer would start with a new empty view, create a hierarchy of new 'folders' into which different sections of the code can be organized, and describe how parts of the original view should be mapped to the structure of the new one. This latter step might be accomplished by enumeration (*e.g.*, by dragging a part of the original view and dropping it at an appropriate location within the new one) or by a computational rule. We are considering many candidate computational rules, from simple pattern-based rules (*e.g.*, methods with signatures matching a particular regular expression, or methods in which the code fits the pattern of a simple accessor method) to more sophisticated dependency-based techniques such as program slicing [10]. Additional mechanisms, such as the heuristic method classification techniques used in Codecrawler [9], and closure operators

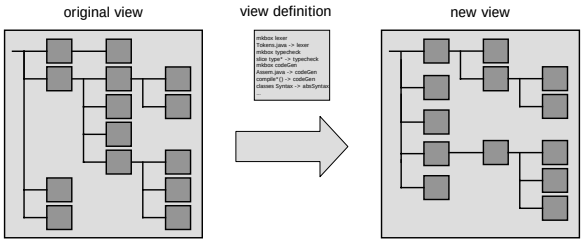

Figure 1. A view definition captures the relationship between two views by describing how one of them can be derived from the other.

(*i.e.*, adding the minimum amount of code from the original view that will make the new one executable) will also fit naturally into this framework.

We plan to implement our tools as plugins to the Eclipse framework. Eclipse (http://www.eclipse.org) is "an open extensible IDE for anything" that is rapidly gaining in popularity. It will give us a high-level platform on which to build, and also a vehicle for disseminating our work: Eclipse users will be able to reap the benefits of multiple views without giving up the other tools with which they have become familiar.

An important aspect of the multiple-view browser is that the steps taken to build the new view are recorded in a *view definition* that can be used to reconstruct the new view if changes are subsequently made to the original view. As a program evolves, we expect it to accumulate a library of such view definitions, each corresponding to different views that have been found to be useful. These definitions would themselves be first class entities, on a par with the program source code, and could be edited and revised as necessary. Indeed, in some cases, definitions might even be shared between different projects or development teams. After a while, users of such a system might begin to loose the sense of there being any particular, distinguished *original* view, seeing instead just a menu of *different* views. Internally, the relationships between views might be represented by a graph, as shown in Figure 2, with translations between adjacent views corresponding to particular definition, and with moves between arbitrary connected views made possible by composing the definitions.

5.2. Extending the use of Traits for structuring classes.

This research builds on our successful experience with traits in Smalltalk, and is being conducted in collaboration with the members of the Software Composition group at the University of Bern.

Traits as described in section 3.2 contain pure behavior, that is, they contain methods but no fields (*i.e.*, instance or

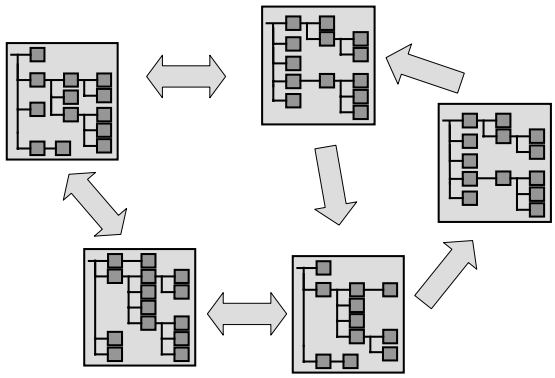

Figure 2. Moving between multiple views

class variables) or references to fields. This simplifies not only the implementation of traits but also their use: because field access can occur only in a method supplied by a class, there is never any ambiguity about what an access means. However, some applications require state. For example, one of our "customers" has expressed interested in using traits to add transactional behavior to arbitrary objects: this requires adding a field to store the transaction id as well as the appropriate methods.

We believe that we will be able to find a way to implement "state traits", that is, traits that add various kinds of fields to a composition. However, we are concerned that doing so may destroy the pleasant conceptual properties that make traits more friendly to programmers than mixins or multiple inheritance [17].

Given state traits, it will be possible to refactor inheritance into a combination of a behavior trait (containing the methods of the superclass), a state trait (containing its fields), and some local definitions. The ability to add aliases to methods from the "superclass trait" can be used to simulate **super**. This suggests the question: is this refactoring useful as an aid to understanding a program using inheritance? Or does it provide only a clumsy way of expressing something that inheritance itself captures more clearly and simply? Put another way: we need to consider whether inheritance itself is a beneficial language feature.

Neither traits, nor the Smalltalk language in which they have been implemented, enforce any controls on the visibility of methods. However, traits exacerbate this shortcoming of Smalltalk, because in Smalltalk all fields are private to the object that contains them, whereas using traits frequently requires that fields have accessor methods, and are thus effectively made public. Because visibility control can be an important tool for program understanding, we seek a solution that is more complete than visibility declarations in Java. We are exploring using interface declarations to control visibility, which will enable different clients to see different parts of an object at run-time [18].

5.3. Integrating Traits into the Multiple-view Browser

Our third research activity is to develop a traits model for Java-like languages and to integrate it into the multiple-view browser. Because of the way that Java was designed, this requires a lot more than mere translation from one syntax to another.

Because Java uses the name of a class explicitly within the body of that class, the way that traits are viewed must be changed. The simplest example arises with "constructors," which do not have names of their own, but overload the name of the class. We plan to use a keyword to identify constructors in a trait. The class name is also used in declarations of parameters, fields and local variables within the class. For example, the method copy in the class Set may declare a local variable of type Set. If copy is abstracted into a trait, the word Set must be replaced by something more generic. The obvious solution is to generalize the language by adding a notation for **thisType**, but this generalization may also complicate the type system.

The type of a method in a trait is parametric: it will change depending on the class into which the trait is ultimately incorporated. We might simply ignore this problem and duplicate the trait method code whenever it is used so that it can be type-checked in the appropriate context. Apart from being conceptually inelegant, this approach also has some practical problems, including object code bloat and the possibility of confronting the programmer with type errors in generated code. It would be preferable to type-check trait methods once, in a type system general enough to ensure that they are correct wherever the trait is used. It is not yet clear to us whether the GJ type system [4] is sufficiently expressive to deal with trait methods, or whether it must be extended further.

Studies of Java code have also suggested that there are other opportunities for reuse that traits could capture if they were to be suitably generalized. For example, identical method bodies appear in different classes, but with different visibility or synchronization declarations. The duplication could be avoided if these declarations were made "first class," so that they could be applied to a method obtained from a trait.

The criterion for success in all these activities is not conceptual elegance, but making real programs easier to understand and reuse. This can only be assessed by implementing traits in the multi-view browser, applying it in the field, and evaluating how well the browser helps programmers to accomplish their tasks.

6. Impact of our Research

Understanding large legacy programs well enough for them to be modified safely is *the* critical problem in software maintenance and re-engineering, and one to which several conferences and workshops are devoted. However, collaborations between the re-engineering and the programming language research communities are uncommon. We hope to fill that gap.

We also have hopes that we can influence the way in which industrial programming tools are built. We have demonstrated the traits browser to Cincom, the leading commercial Smalltalk vendor, which has expressed interest in incorporating this technology into a future version of VisualWorks. The Squeak community has also been enthusiastic in its reception of traits, even at this early prototype stage. It seems that the theoretical benefits of traits do pay off in practice: many Smalltalkers who refused to accept either multiple inheritance or mixins are surprisingly open to the idea of adopting traits. We believe that this is because the method-level syntax and semantics of the language remain unchanged, and the extra structure and reuse

opportunities provided by traits are therefore always optional. The practical influence of the multiple-view browser will be magnified because we plan to target it on Java and build it as a plugin to the Eclipse framework (see section 5.1). Nevertheless, we believe that many of our results will be applicable to programming languages in general, and not just to Java or Smalltalk.

References

[1] Andrew P. Black, Nathanael Schärli and Stéphane Ducasse. Applying Traits to the Smalltalk Collection Hierarchy. In ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03), October 2003, pp. 47–64, Anaheim, CA: ACM Press.

[2] Andrew P. Black and Nathanael Schärli. Traits: Tools and Methodology. In International Conference on Software Engineering (ICSE), May 2004, Edinburgh, Scotland.

[3] Gilad Bracha and William Cook. Mixin-based Inheritance. In ECOOP/OOPSLA'90, Ottawa, Canada, October 1990, pp. 303-311: ACM Press.

[4] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In OOPSLA'98, October 1988, pp. 183–200, Vancouver, Canada: ACM Press.

[5] Martin Fowler. *Refactoring: improving the design of existing code*. The Addison-Wesley object technology series, ed. G. Booch, I. Jacobson, and J. Rumbaugh. 2000: Addison-Wesley. xxi+431 p.

[6] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley professional computing series. 1995: Addison-Wesley. iv+395 p

[7] Andreas Kacofegitis and Neville Churcher. "Theme-Based Literate Programming." In Ninth Asia-Pacific Software Engineering Conference (APSEC'02). 2002.

[8] Shriram Krishnamurthi, Matthias Felleisen and Daniel P. Friedman. "Synthesizing Object-Oriented and Functional Design to Promote Re-use." In ECOOP 1998, pp. 91–113: Springer Verlag, Lecture Notes in Computer Science, vol. 1445.

[9] Michele Lanza and Stéphane Ducasse. "A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint." In Object-Oriented Programming Syst., Lang. and Applications 2001: ACM Press, pp 300–311.

[10] Loren Larsen and Mary Jean Harrold. "Slicing Object-Oriented Software." In Proceedings of the 18th International Conference on Software Engineering (ICSE'96), March 1996, pp. 495-505.

[11] Scott Meyers. "Difficulties in integrating multiview development systems." IEEE Software **8**(*1*) 1991: 49–57.

[12] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification." IEEE Transactions on Software Engineering **20**(*10*) 1994: 760-773.

[13] William F. Opdyke. *Refactoring Object-Oriented Frameworks* [PhD thesis]. University of Illinois at Urbana-Champaign, Urbana, Illinois: 1992. 151 p.

[14] Steven P. Reiss, "Simplifying data integration: the design of the Desert software development environment." In Proc. 18th International Conference on Software Engineering. (ICSE'96)1996, pp 398–407.

[15] Steven P. Reiss, "The Desert environment." ACM Transactions on Software Engineering and Methodology (TOSEM), 1999. **8**(*4*): pp 297–342.

[16] Don Roberts, John Brant and Ralph Johnson, "A Refactoring Tool for Smalltalk." Journal of Theory and Practice of Object Systems 1997. **3**(*4*): pp 253-263.

[17] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Andrew P. Black. "Traits: Composable Units of Behavior." In Proceedings ECOOP, July 2003, Darmstadt, Germany: Springer Verlag, Lecture Notes in Computer Science, vol. 2743, pp. 248-274.

[18] Nathanael Schärli, Andrew P. Black and Stéphane Ducasse. "Object Encapsulation for Dynamically Typed Languages." Technical Report CSE 04-002, OGI School of Science & Engineering, Oregon Health & Science University, Beaverton, OR. March 2004.

[19] Peri Tarr, Harold Ossher, William Harrison and Stanley M. Sutton, Jr. "N degrees of Separation: Multi-Dimensional Separation of Concerns." In International Conference on Software Engineering (ICSE) 1999, pp. 107–119, Los Angeles, CA: ACM Press.