Perspectives On Software

Andrew P. Black and Mark P. Jones
{black, jones}@cse.ogi.edu

Department of Computer Science & Engineering
Oregon Graduate Institute of Science & Technology
Beaverton, Oregon, USA

A Position Paper submitted to the OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-oriented Systems

Introduction

Since the dawn of creation, which, for the purposes of discussing computer programming we will take as 1950, programming has been a been a linear activity, in the sense that a program is a linear sequence of statements. When we use indentation to group statements, we are attempting to add some two dimensional structure to a linear artifact, in order to make the program easier to understand.

Instead, imagine a program at a higher level of abstraction: rather than dealing with program text, treat the program as a much richer abstract program structure (APS) that captures all of the semantics, but is independent of any syntax. Conventional one and two dimensional syntax, abstract syntax trees, class diagrams, and other common representations of a program are all different “views” on this rich abstraction.

“Perspectives” is a new approach to software development that uses an APS to describe programs. In this setting, programmers move between different views of a program to help them understand the original code, and to isolate relevant dimensions when changes are required. They create new views that collect together all of the code pertaining to a particular aspect of concern, so that this aspect can be understood in isolation; code irrelevant to the task at hand is out of sight. Such a system has the potential to support more principled and more reliable evolution of software artifacts, reducing the risks and costs that result from being constrained to a single view of a program. We also believe that Perspectives has great potential as an educational tool, since it will enable a complex program, for example, a compiler, to be presented to a class of students incrementally. At any time, the current view can focus on the topic of the current lecture, and extraneous detail can be hidden

We believe that elevating the program above the syntax and semantics of a single linear representation is a liberating idea, but at the same time a controversial one. Like prisoners released from Plato's cave, we are at first unable and then unwilling to grasp that there is a reality for programs that is more substantial than the two-dimensional shadows that we have been seeing for all of our lives.

Abstract Program Structures

Many refactorings, rather than being thought of as semantics-preserving source to source transformations, can instead be thought of defining an equivalence class of programs. The APS may be thought of as being that equivalence class, or alternatively as being a representative from which the members of the class can be derived. For example, consider the *Encapsulate Field* refactoring[5, p206], in which a public field is made private and all uses of and assignments to that field are replaced with sends of *get* and *set* messages. This refactoring is usually thought of as defining an equivalence between two programs, one with direct access to the public field, and one with all access through message sends. Instead, think of the program as a single more abstract structure, in which the field is inherently neither public nor private. However, both of these concrete representations can be obtained by projecting *views* from the abstract structure.

Figure 1 is a conceptual view of Perspectives. The cube is an abstract program struc-

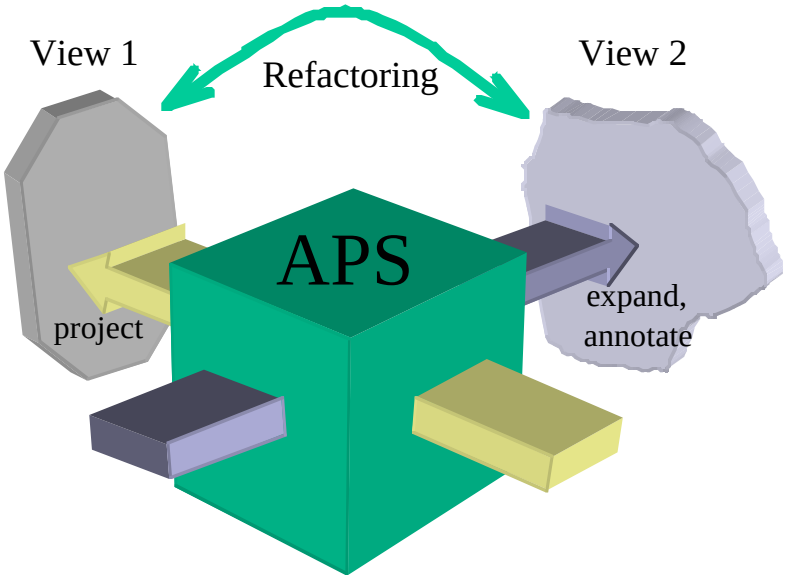

Figure 1: A conceptual view of Perspectives

ture (APS). The arrows represent two different views of the APS. In the case that two views 1 and 2 are both complete representations of the APS, they must be equivalent. The curved arrow then represents a (possibly compound) refactoring, which witnesses the equivalence of the views. Of course, the views may represent completely different aspects of the program, such as synchronization and memory management. In this case, they are in no sense equivalent and there is no refactoring that will take us from one to the other.

Taking the idea of abstracting away from the concrete program even further, the rows and columns dichotomy between abstract data types and objects disappears: ADTs and objects are just two different perspectives on the same APS, which contains the individual method bodies as basic units. The object-oriented view supplies clustering into classes; the ADT view supplies clustering into procedures that pattern-match by constructor.

Notice that our notion of *view* is not a pure projection: in addition to discarding information irrelevant to the concerns of that view, a view can also *add* information. For example, a view that adds an abstract superclass to a class hierarchy will also

provide a name for that superclass; a view that extracts a method from the midst of another method must provide a name for the new method and must define exactly which statements should be abstracted to form its body. The additional information provided by a view does not change the semantics of the program, but does change its presentation, and can be vital to its comprehensibility. Neither is there any assumption that all views must be orthogonal, or that a particular feature in the APS will be manifested in exactly one view.

Aspects as Views

We do not yet have a full understanding of the relationships between views of an APS and Aspects; indeed, both concepts are as yet only loosely defined. However, based on our (limited) experience with aspects, it seems that some aspects can be represented as views in Perspectives.

To illustrate the idea, we will use as an example the toy text editor that forms part of the JBuilder tutorial [1] which we have previously programmed using AspectJ [2]. Step 10 of the JBuilder tutorial introduces a “dirty bit” to record whether the file being edited has changed since it was last read from or written to disk. In Step 11, this bit is used to modify the behaviour of several other methods.

The aspect-oriented version of the program localizes all of this functionality as the *ChangeMonitor* aspect. This aspect introduces the boolean instance variable *changed*, and then provides **before** or **after** advice on five methods, whose behaviour is made to depend on *changed*. It also introduces a new method, *okToAbandon*, which is used in two of the pieces of advice.

How might this change monitoring aspect be represented in Perspectives? First, imagine that all of the relevant code has already been added to the APS. That is, the programmer has determined where the change monitoring code needs to go, and has inserted it; the process is conceptually identical to what happened in the JBuilder tutorial, or in the construction of the *ChangeMonitor* aspect.

The important question is: how can the interdependence of these code fragments be recognized weeks or months later, when a bug is found or the functionality needs to be enhanced? The answer that AspectJ offers is simple: all of the fragments are in one syntactic unit (the aspect). Perspectives offers a different answer: define a view from the APS that contains exactly the code that deals with change management.

Program slicing [3] provides the technology to build this view. The forward slices of the program using assignments to the *changed* instance variable as the slicing criterion will include three of the code fragments that were inserted as **before** or **after** advice on the editor's methods. It will also include the body of the introduced method *okToAbandon*, since the whole of the body is conditional on *changed*. The other two pieces of advice depend on the result of *okToAbandon*, which in turn depends on *changed*, so they too will be included in the slice.

Outline of Work

Perspectives does not yet exist. We have described a vision, but many issues must be addressed before this vision can be made a reality.

At the very foundation of Perspectives is the idea that a program *can* be adequately represented by a high-level Abstract Program Structure. This is true if and only if refactorings really are semantics-preserving: if the source and destination of a refactoring are not equivalent, then there are no equivalence classes to be represented in the APS.

Consider a simple refactoring: adding an empty abstract superclass to the class hierarchy. In a fully reflective language like Smalltalk, when any object can, for example, count the number of classes between its own class and Object, even such a benign change is not, in general, semantics-preserving. Indeed, even changing the name of a variable or a method may change the semantics of a program in the presence of reflection. The same is true for Java, except that the design of the Java class libraries is such that a program that never refers to the reflection classes should not be affected by refactoring.

But the Java libraries are large and complex: how can we tell if they have been designed so that this property in fact holds? Can we devise a type system that segregates meta-level “reflective” properties from basic operations? If so, then a program free of certain types may be provably invariant under refactoring. Alternatively, Ungar’s Self language [4] and others have used mirrors to access all meta-level information, so that a mirror-free program is guaranteed to be non-reflective.

This brings us to the question of language-independence. The APS should be independent of any particular syntactic representation, and could be used as a way of translating from one language to another. However, it is should also be clear that certain language features, such as unrestricted reflection, may make it impossible to apply Perspectives to a particular language. Are these features ones that would be missed—or is it perhaps the case that the features that make it impossible to fully abstract an APS from a concrete program also make it hard for a human reader to understand what the program seeks to accomplish?

It is also unclear what views will actually be useful when maintaining a real program. We have suggested that program slicing on appropriately chosen criteria could be used to create views that are like aspects. We do not yet know whether all aspects of interest can be captured in this way, for example, aspects that deal with concurrency and synchronization, or with purity (absence of effects). Other properties, like the immutability of an object, which depend on the *absence* of certain statements, rather than on their presence, may prove hard to capture as a view.

We plan to build Perspectives incrementally, initially ignoring some of these issues while we explore others, and learning from our experiences in applying the early prototypes to real programs.

References

[1] The TextEdit tutorial from the Inprise JBuilder documentation, http://www.borland.com/techpubs/jbuilder/jbuilder3-xplat/pg/tetutorial.html,

[2] 3 Examples for the use of AspectJ, originally part of the tutorial material for a prior version of AspectJ. http://www.cse.ogi.edu/~black/3AspectExamples/textedit.html

[3] F. Tip, “A Survey of Program Slicing Techniques”, Journal of Programming Languages, Vol.3, No.3, pp.121-189, September, 1995.

[4] *The SELF 4.0 Programmer’s Reference Manual*. Ole Agesen Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar and Mario Wolczko. Sun Microsystems and Stanford University, 1995.

[5] Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley (Object Technology Series), 1999.