Towards a New Model of Abstraction in Software Engineering

Gregor Kiczales
Xerox Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA 94304

Abstract

Existing abstraction techniques lead to implementations which overly constrain their users. This happens because these techniques force the implementor to make all the decisions about how to implement the abstraction including some which would better be made by the user. We have developed a new model of abstraction, which we call the "two-view" approach; the first view is the traditional one, it provides the functionality of the abstraction, the second view allows the user to participate in some implementation decisions. This approach promises to be general enough to handle a wide range of applications.

Problems With the Existing Model

The source of the problem with existing abstraction mechanisms is that not only do they shield users from having to know the details of how an artifact is implemented, they also prevent users from altering those details even when that might be desirable. That is, those details become fixed, implicit properties of the implementation. As such, they constitute immutable decisions, made by the implementor, about the use profile towards which the artifact is tuned. No matter how well designed the implementation is, there will inevitably be many user programs for which it is not optimally tuned.

As we have said earlier, the typical object-oriented programming language abstraction hides the internal representation of instances. As part of this hiding, it is typical for implementors to use a packed array format for instances. This has the effect of biasing the implementation towards programs in which most instances use most of their slots. If on the other hand, a user's application defines classes whose instances use only a few of their slots, this implementation strategy is inappropriate; a sparse array strategy would be better suited. Notice that in this case, and other cases like it, it is the implementation of the abstraction which is failing; the abstraction itself is perfectly capable of expressing the intended behavior.

This example has a property that is common to such cases and which suggests a possible solution as well. The property is that even though the dense instance representation may not be appropriate for the specific application in question, most other aspects of the implementation (e.g. method dispatch) may well be appropriate. The solution which is suggested is to find a way to open the abstraction's implementation so that the user can alter those aspects they need to while leaving the rest of it intact.
The Two-View Model

This suggestion leads to a new model of abstraction in software engineering in which two side-by-side views are taken of each piece of functionality. The first, which we call the base view, is the traditional one; it presents an interface that allows the programmer access to useful functionality. The user implements their functionality on top of the base view in the traditional sense. In the case of an object-oriented programming language this view provides classes, inheritance, methods and all the other typical behavior.

The second view allows the user to control certain aspects of how the first view is implemented. It controls the implementation of the base view, and is called the implementation view. The implementation view is used to reach inside the base view and incrementally adjust how certain aspects of its behavior are implemented. In the case of an object-oriented programming language, this view makes it possible to adjust the implementation of instances, slot access, method dispatch etc.

While many systems allow user extension, the key point of our two-view framework is that it separates user code which builds on top of the underlying functionality from user code which alters the implementation of the underlying functionality. In addition, there is the critical notion of incrementality: the user of the implementation view uses it to adjust the existing implementation, not to create a new implementation from scratch.

Our work in this area has been on three fronts: the two-view model of open abstraction itself, its application to programming language design and its application to other kinds of systems (i.e. window systems).

We have developed a new approach to programming language design, which we call metaobject protocols, that directly supports this two view approach. The base view is just a normal programming language, the implementation allows the user to adjust the behavior and implementation of that language. Our initial work with metaobject protocols has been in the context of object-oriented programming languages; in these cases the implementation provides access to the runtime of these languages rather than the workings of the compiler itself. But, we believe it is possible to develop a metaobject protocol based compiler and are pursuing that in our present work. The current problem with high-level languages is that for any given program the typical compiler is too general purpose—that is, it isn’t biased enough towards that particular program. Adding an implementation view would effectively allow the user to select the appropriate compilation strategy for any given part of their program. We believe this approach has the potential to make high-level languages as efficient as C and other assembly languages.

In other work, we have explored the use of the two view approach in the implementation of other complex abstractions. Silica[2] is a window system kernel with an explicit implementation view. We believe this approach can be applied to other kinds of systems, including operating systems. In fact, there are a number of existing systems which are open in the ways we have mentioned. Many of these systems can be profitably re-understood in terms of our two view model. For example, the Mach paging system can be described as having a base level which provides paging functionality and an implementation level which makes it possible to adjust the base level functionality. Reconsidering Mach in these terms can help to make it more clear what functionality should be placed in which interface and what role each piece of user code is playing.

References
