Schizoid Classes

Rodney Bates, Wichita State University

Java also calls a receiver the “object for which the instance method was invoked,” and C++ also calls it the “object for which the function is called.”

The Smalltalk project, carried out during the 1970s at Xerox PARC (Palo Alto Research Center), is most famous for introducing the GUI (graphical user interface). It also brought us a series of three programming languages, culminating in Smalltalk-80, which is a pure object-oriented language in the sense that every type is a class type, and every operation is an overridable method. This applies to the built-in types and operations of the language, as well as those defined by the programmer.

Smalltalk-80 was an important and enlightening experiment in just how far object-orientation can be taken in a programming language. It is simple, compact, and shows a rare and refreshing integrity of concept. To accomplish its goals, it introduces the idea that the variables of a class can be either *class variables* or *instance variables*, and the methods can be either *class methods* or *instance methods*. This turns the class into a mixture of two fundamentally different concepts—type and module—with very different semantics. Smalltalk manages to do this relatively cleanly.

Unfortunately, two more recent languages, C++ and Java, have taken this same distinction and turned it into a gratuitous mess. Let’s look just at these two languages for a moment, then come back to Smalltalk. First, a bit of review of some concepts. Different languages often use different terms for the same or similar concepts. To avoid a terminological explosion, I will use terms from Smalltalk, which are mostly the same as Java terms (see Table 1).

A *type* defines a set of possible values (and maybe a set of operations on them, as well). It can be built in to a language (for example, integer or Boolean) or programmer-defined (for example, an array, record, or class type). There can be many *instances* of a type in a program, or none at all. Simply declaring a type doesn’t create any instances. They are usually created by declaring local or global variables, or by executing allocators for heap-allocated objects.

A *module*, in contrast, always exists in exactly one instance in a complete program, and this instance is cre-

Of class,

TYPE, AND METHOD

ated by the module declaration. It groups a bunch of declarations, showing that they are logically related.

It generally introduces a

space of names and allows qualified names to be used to make the grouping explicit and to avoid name conflicts. It frequently corresponds to a separate compilation or source file. Modules raise several additional, important programming issues, but the rest are incidental to this discussion.

An *object type* is an object-oriented, programmer-defined type that contains variables and methods. Each instance has a complete set of the variables. Each method is mostly like a function, but with some significant additional properties. It has a special parameter, called the *receiver* parameter, that is an instance of the type.

In a method call, there is a special argument called the *receiver* argument that plays three roles. First, its static type gives the object type where the search for the method name is to begin. Second, the dynamic type of the object it denotes is used at runtime to *dispatch* to one of possibly several method bodies, all with the same or compatible parameter lists. Third, it is passed as a special parameter to the method in addition to the parameters found between parentheses.

In C++ and Java, the construct called a *class* is neither a module nor a type, but a confusion of both. With respect to the class variables, however, it is a module, in that there is always exactly one instance of each of these, and the class declaration alone creates it. With respect to the instance variables, however, it is an object type. There is no instance of these created by the class declaration, but each declared local or global variable and each heap-allocated object that identifies the class as its “type” has a complete set of instance variables.

A class method is not really a method at all. It is just a plain function in strange dress. It has no receiver parameter, and calls on it don’t provide one and don’t dispatch. With respect to the class “method,” the class behaves just like a module, as it does for a class variable. An instance method, in contrast, is a real method. It has a receiver parameter, and it is invoked using a receiver argument

that plays all three receiver roles. With respect to an instance method, the class is a real type.

All this greatly confuses calls. A call can be either a plain function call or a true method call, with a receiver. This is a very important semantic difference to a programmer who is writing or reading the call and thinking about what it means. This might be reasonable if the distinction were syntactically explicit, but in both Java and C++, the absence or presence of a “receiver” in the source code is orthogonal to whether the semantics of the call actually involve one.

Consider this simple C++ call: `r -> f()`. In the consistent case, this is a real method call, on an instance method named `f`. The receiver is `r` and plays all three roles. But it is also possible that `f` is actually a class method. The syntax has a misleading “receiver” `r`, but semantically there is none. In fact, `r`’s only function is to provide a type that is used to look up the method `f`. There is no dispatching, `r` is not passed—and can even be nil.

Or, consider a syntactically different call, `f()`. The syntax omits the receiver, and in the consistent case, there is none. This could be a traditional call, if `f` is an ordinary function. If the call occurs inside the body of a method (class or instance), named `g`, of some class `C`, then `f` could be a method of `C`. If it is a class method, there is also no receiver. But if it is an instance method, there is an implied receiver, the same object that was the receiver in the call to `g`.

The absence or presence in the syntax of a “receiver” is independent of whether one actually exists. Two of the four cases are misleading. In the other two cases, the syntax of the call is consistent with its semantics, but this doesn’t help because a reader has to go through an elaborate process to figure out whether a given call is one of the contradictory cases. In general, finding the right declaration for `f`, which will answer the question, involves searching a complex, multi-branching tree of declarative regions. The possibilities for Java are similar, though slightly less complicated in some ways.

This gets a whole lot muddier in both these languages because of programmer-defined overloading. The multiple candidate meanings of a single method name can include a mixture of class and instance methods. In Java, they can also come from several different class definitions. This leads to a language complexity explosion that few working programmers come close to understanding, let alone working though for every call they write or read. Java requires 16 pages to explain the rules for determining what a call actually means, and C++ takes 15, counting six additional function-like constructs. The entire Oberon-2 language is defined in only this many pages.

Let’s return to Smalltalk. It manages to introduce class variables and methods without such a mess. As a small point, it syntactically groups all class attributes together and all instance attributes together, which helps make this important semantic distinction more explicit.

Moreover, Smalltalk creates at runtime, for every class, an actual object, which I will call the *metaobject*. This holds the class attributes. There is exactly one of these, the language explicitly identifies it, and most important, the “class” attributes of the original class are actually instance attributes of the metaobject. In fact, you can think of the notation for defining class attributes as just syntactic sugar for declaring instance attributes of the metaobject. The language creates the metaobject (as well as a class for it, called the *metaclass*) and ensures that there is exactly one instance, thus giving it a module-like property.

Most important of all, accessing a class variable is actually done by accessing that instance variable of the metaobject, and the metaobject is syntactically explicit. Similarly, invoking a class method is done by making a normal method call on the corresponding instance method of the metaobject. As a result, there is only one call construct in the language, with one set of semantic rules. It always has a receiver, the receiver is always syntactically explicit in the call, and it always plays all three of the semantic roles of a true receiver.

Java’s and C++’s approach to “class” variables and

methods is an example of single-tool mentality gone overboard. The poor class is hopelessly schizoid. It doesn’t know if it is a type or a module, and is often a confused mix of both. The confusion extends to many programmers, especially when trying to figure out just what an

TABLE 1 Equivalent Terminology in Smalltalk, Java, and C++

Smalltalk	Java	C++
class variable	class variable	static data member
instance variable	instance variable	nonstatic data member
class method	class method	static member function
instance method	instance method	nonstatic member function
receiver	target reference	implicit object parameter

innocent-looking call actually means.

Putting class variables and methods into a class is attempting to make an object type into both a type and a module. Compared with separate constructs, it is harder to use for either purpose alone or for both purposes simultaneously. It adds huge and gratuitous complexity to a programming language, in a world where most working programmers don't understand their language, anyway.

When you find out your hammer is ineffective for cutting, the solution is to add a saw to your toolbox. If you operate under the preconceived notion that one tool should be used for everything, you will have to put the saw teeth on the handle of the hammer, which makes the one tool less effective than separate purpose-designed tools, both for driving nails and for sawing.

The best-designed languages give you two abstraction tools—a module and an object type—each of which serves its own purpose reasonably well. A module can contain any declarable entity, including variables (instead of class variables) and functions (instead of class methods). If you need to associate a type with a single-instance abstraction, you can put that type in the module, too. If you need several intimately related types, you can put them all there—for example, a graph module, which needs a type for graphs, nodes, and arcs. This is another

case that is poorly handled by a class as the only abstraction construct.

Smalltalk pays a high price elsewhere for taking object orientation to the extreme, notably in complete loss of static typing and serious runtime efficiency penalties. Special, one-instance forms of classes are, for many programming problems, not as good a conceptual match as modules. But at least it provides a single, consistent, and syntactically explicit call mechanism. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

RODNEY BATES (rodney.bates@wichita.edu) has never done a paid day's work in electrical engineering, for which he was trained. Instead, he has worked 33 years in computer software, two-thirds in industry, the rest in academia. He has been involved mostly with operating systems, compilers, and as a resident programming language lawyer. He has contributed to popular computer magazines, trade and research conferences, and academic journals. His current big project is a semantic editor for Modula-3 and other languages. He is an assistant professor and graduate coordinator in the computer science department at Wichita State University.

© 2004 ACM 1542-7730/04/0900 \$5.00

ACM TRANSACTIONS ON INTERNET TECHNOLOGY

Transactions on Internet Technology (TOIT). This quarterly publication encompasses many disciplines in computing—including computer software engineering, middleware, database management, security, knowledge discovery and data mining, networking and distributed systems, communications, and performance and scalability—all under one roof. *TOIT* brings a sharper focus on the results and roles of the individual disciplines and the relationship among them. Extensive multi-disciplinary coverage is placed on the new application technologies, social issues, and public policies shaping Internet development. **Subscribe Today!**

PRODUCT INFORMATION

ISSN: 1533-5399

Order Code: (108)

Price: \$37 Professional Member

\$32 Student Member

\$150 Non-Member

\$13 Air Service (for residents
outside North America only)

TO PLACE AN ORDER

Contact ACM Member Services

Phone: 1.800.342.6626 (U.S. and Canada)
+1.212.626.0500 (Global)

Fax: +1.212.944.1318

(Hours: 8:30am—4:30pm, Eastern Time)

Email: acmhelp@acm.org

Mail: ACM Member Services

PO Box 11414

New York, NY 10286-1414 USA

Association for Computing Machinery
The First Society in Computing
www.acm.org

AD24

www.acm.org/pubs/periodicals/toit

AD22