Editor: Li Gong • li.gong@sun.com

# Post-Javaism

**Andrew P. Black** • *OGI School of Science & Engineering, Oregon Health & Science University*

The Java programming language has been a phenomenal success. It's a significant improvement over C and C++, and its libraries for network and GUI programming have introduced large numbers of programmers to previously esoteric disciplines. But Java isn't the end of programming language history. What language will we use 10 or 20 years from now?

Although good news for most programmers, Java has been a mixed blessing for programming language researchers. On the positive side, Java has demonstrated that a quality language can make complex programming tasks much simpler: it shows that language design is still a relevant discipline. On the negative side, Java's success has made obtaining support for research into new languages — and publishing the results of that research — more difficult. The effect is that object-oriented language research focuses on fixing Java's trouble spots or extending it to provide missing functionality. These are not bad activities, but they are necessarily limited to Java-like languages: statically typed, class-based, single-inheritance languages with a weak notion of interface.

The recent European Conference for Object-Oriented Programming (ECOOP) included the Workshop on Object-Oriented Language Engineering for the Post-Java Era at which researchers gathered to examine languages that diverge from this model.

## What is Post-Javaism?

The prefix "post" is used in art and architecture to indicate not just chronology — that one period or movement follows another in time — but also a change in direction or philosophy. Postmodernism, for example, is a reaction against modernism's established principles. Similarly, the call for participation to the ECOOP workshop effectively defines post-Javaism as a reaction against Java's established principles.

In architecture, the *modernists* used scientific principles to create stark, functional designs that flouted convention. Architectural modernism's defining moment was the 1958 construction of the Seagram building in New York (see Figure 1). With its functional, clean, and simple lines, it quickly became one of the US's most architecturally influential office buildings. With its ethos captured in phrases that have passed into our cultural heritage (think "less is more" and "form follows function"), modernism became the dominant style of the 1960s and 1970s.

In *Complexity and Contradiction in Architecture* (1966), Robert Venturi responded with a definition of postmodernism. He celebrated the rich mix of historic styles found in great cities such as Rome and observed that many people find this complex, sometimes contradictory, mix more comfortable than stark minimalism. That said, postmodernism is not a return to classical architecture; rather, it takes classical elements and uses them in incongruous ways. The AT&T building in New York, which incorporates classical arcades and a pediment reminiscent of a Chippendale highboy, is a defining example (see Figure 2).

## Modernism in Programming Language Design

It is Smalltalk, not Java, that provides an analog to architectural modernism. Smalltalk was a sparse, simple language with few unnecessary features; its form was unconventional but functional.

*continued on p. 93*


*Figure 1. Seagram building, New York. Modernism: less is more.*

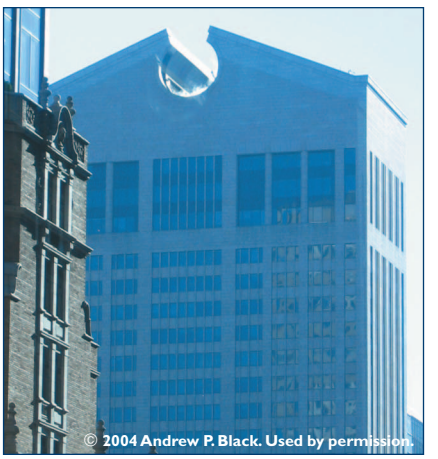


*Figure 2. AT&T building (now Sony Plaza). Postmodernism: using classical elements in inconguous ways.*

continued from p. 96

Indeed, "form follows function" describes Smalltalk well. Smalltalk rejected established conventions, such as storing programs in files, using lots of keywords, declaring types for identifiers, and sprinkling everything with parentheses regardless of necessity. Instead, Smalltalk embraced an architectural model of amazing purity: everything was an object, from classes to closures, dictionaries to methods, messages to numbers. Everything that supported the language — compilers, browsers, execution contexts, debuggers — was an object, too. Less was more.

The result was breathtaking: a palace that seemed to defy gravity and certainly challenged conventional wisdom. Smalltalk provided the seed that blossomed into many of the past 30 years' innovations, including Java, graphical interfaces, programming in the debugger, and contextual menus. However, most of the tourists who came to admire this modernist palace went back home to their Victorian row houses. Only a few visionaries felt they could live there.

### Java as Postmodernism

Taking the analogy to architecture one step further, we can say that Java defined postmodernism. It celebrated the mix of styles in historic bodies of code and blended classical elements such as bytes, Bools, and curly braces with the functional modernist forms of objects and instance variables. It gave people back the conventions that made them feel comfortable — along with a healthy dose of complexity and contradiction.

The result was that people liked Java. They didn't just visit on Sunday afternoon tours: they gave up their Victorian row houses and moved in. Java represented not innovation, but consolidation.

## The Forces of Evolution

I don't have enough space to pursue the architectural analogy too much further, but I'm constantly surprised by its strength. Is this just coincidence, or do similar forces shape the evolution of both architectural styles and programming languages?

### Evolution in Architecture

Four major forces have shaped architecture's evolution: technology, economics, function, and fashion.

New technology leads to structures that were never before possible: dressed stone arches, cast-iron facades, reinforced concrete, steel girders, plate glass, mechanical ventilation, insulation, engineered wood products — the list goes on. Simultaneously, economic changes help one technology advance as another recedes. Large plate-glass windows, for example, were once the prerogative of the fabulously wealthy; the rest of us had to be content with arrays of small panes divided by mullions. The situation today is reversed: large expanses of glass are inexpensive, and we must pay extra for true divided lights.

New functions are another major source of innovation. The Romans did not build railway stations, and the Victorians did not build multistory parking structures. Each new function catalyzed the creation of new forms, and often of the technology that made these forms possible.

Finally, we can't underestimate fashion's importance. The powerful display their wealth by creating new fashions; the rest of us often have the urge to follow them. I won't try to explain fashion, but denying that it's a powerful force for change would be foolish.

The result of these forces has pushed architecture toward larger, more open buildings, with such huge oscillations about the mean that this trend becomes apparent only decades or centuries later. Will the same be true for programming languages?

### Evolution in Programming Languages

The forces that effect programming languages are similar to those in architecture. New technology is the most obvious: type inference, parser generators, garbage collection, large and inexpensive memories, peephole optimizers, just-in-time compilers, and advances in raw hardware speed have changed languages in ways that Fortran's designers could not conceive.

Layered over this is the effect of economics. Computation is a young field, but we've already seen it progress from a dozen programmers sharing a $100,000, room-sized computer to dozens of smaller computers competing for the attention of a $100,000 programmer.

Although each of those smaller computers is a thousand times more powerful than one of the old room-sized machines, our languages are still designed as if the computer's time were more valuable than the programmer's. A similar revolution has occurred in the economics of memory, but perhaps with a more obvious effect: most of us program as if memory were free, which indeed it is — until it runs out.

These changes have made garbage collection and virtual machines the norm, instead of esoterica of the research laboratory. Technology and economics have enabled us to take one step away from the raw hardware — but only one step.

The demand for increased functionality has forced a minority of language designers to look seriously at declarative notations like SQL, used for defining searches over databases, and HTML, used for defining Web pages. Others work with languages that don't even have names: their sentences are sequences of gestures and clicks in a user interface. Approximately 95 percent of all the people who program computers today use these or other domain-specific languages. Of course, the remaining 5 percent — people, like us, who read IEEE Computer Society publications — realize that these aren't "real" languages at all, and that the programmers who use them aren't

"real" programmers. We've even coined a separate name for them: users (or sometimes, lusers).[1] This categorization is similar to the way that "real" architects recognize that food processing plants, fabs, bridges, and municipal housing projects aren't architecture at all, but mere engineering. Yet it has led to a situation in which "real" programming languages are very poor at describing 95 percent of what computers do.

Because of our propensity to ignore new technology and to keep out domain-specific function, fashion has had a disproportionately large effect on programming language design. Because fashion tends to be cyclic, we can reliably predict that the trend will swing away from postmodernism and back to something that looks a lot more like modernism. In other words, the next post-Java language will look a lot like Smalltalk.

## The Smalltalk Revolution

Rather than just repeating history, we should study it; rather than just going back to Smalltalk, we should learn from it. How can we repair Smalltalk's weak points without diluting its strengths?

### Small is Beautiful

One of the great strengths of Smalltalk is that it's small — really small. The Blue Book's syntax diagrams define 27 syntactic categories for Smalltalk, compared to 31 for Pascal.[2] This includes cascades, symbols, and array constants, which I think I would take out if I were designing Smalltalk today. Small is beautiful.

One thing I used to consider Smalltalk's greatest shortcoming, I now believe to be one of its strengths: the absence of type declarations. I think the principal reason for this omission was not that the designers were philosophically opposed to getting early warnings of programming errors, but rather that the technology to check them didn't exist in the 1970s. I can't find anything in Ingall's design principles paper that argues against type declarations.[3] In fact, when we designed Emerald in the early 1980s, we had to invent a type system that was adequate for letting programmers define their own List.of[something] data types.[4]

So why do I now believe that type declarations are a mistake? Let's look at a fragment from a Java program that plays a card game:[5]

```
public DiscardPile discardPile;
...
discardPile =
      new DiscardPile(268, 30);
```

Now consider the following conceptually equivalent Smalltalk code:

```
instanceVariableNames:
          '... discardPile ...'
...
discardPile :=
  DiscardPile locatedAt: 268@30.
```

The redundancy in type name, class name, and instance variable name in the Java just clutters the code without adding any extra information. We now have the technology to infer most variables' types and present them unobtrusively in a programming tool when requested. In fact, Haskell and ML have done this for years. Let's admit that — when supported by proper browsing tools — inference is a better technology, adopt it and move on.

### Reflection

Another great thing about Smalltalk, which the Java folks seemed to miss at first, is that the programming environment is fully accessible from the language. Smalltalk is, in fact, a metaprogramming system: through the wonders of computational reflection, you can access and change the system's internals from within the system itself.[6] Smalltalk had the advantage of casting off the boat anchor of a file-based representation early in its development; Java has not been able to do this yet, despite the valiant efforts of Object Technology International and VisualAge for Java. Because Smalltalk represents programs as objects instead of files, tools are amazingly easy to build. As good tools become available to the community, they become stepping-stones to even better tools, and the process accelerates.

The disadvantage of Smalltalk's easy access to metaprogramming is that it's too easy to use reflective operations "by accident." Java's solution is to label some interfaces as part of a *reflect* package. The Strongtalk language did something more sophisticated: to perform a reflective operation on an object, it required you to first obtain a mirror on that object and then reflect on the mirror.[7] This makes the distinction between reflective and ordinary operations quite clear. I think we can take this idea a bit further and use colored types to distinguish between the results of reflective and ordinary operations. For example, the result of counting the number of classes between a given class and the root of the hierarchy might be a red integer, as opposed to an ordinary white integer. The sum of a red and a white integer would also be red; we could use this scheme to ascertain, by examining their colors, which results depend on reflection.

You might ask why it's important to be able to delimit the effects of reflective operations in this way. The answer is that refactoring is one of our most powerful tools in the constant battle against software entropy.[8] In general, refactorings are semantics-preserving only in the absence of reflection. To see this, consider a program that uses reflection to obtain the name of an instance variable, extracts the fifth and the seventh letters, converts them to ASCII codes, and prints their product. Clearly, changing the name of that instance variable will change the behavior of the program, or even cause

it to crash if the new name is only six characters long. Thus, we see that it is important for the program maintainer to know when it is safe to make what is usually a harmless change, such as renaming a variable, and when it might have unexpected consequences.

### Distribution

I've argued that although programs are called on to perform vastly more, and more varied, functions than anyone could have dreamed of 20 years ago, many of these functions are now the province of domain-specific languages. One functional category that isn't is distributed programming. This is because it is really a metacategory.

One approach to distributed programming is to do what Emerald did: identify a particular programming paradigm suitable for distribution (in this case, mobile objects) and support it in the language. This works well if that paradigm matches the application, but badly otherwise.

Another approach is to provide libraries that support a variety of paradigms; this works particularly well if the language is sufficiently reflective. For example, suppose we want to implement an object-oriented message send to remote objects. Suppose further that rather than stepping outside of the language and resorting to a preprocessor, we prefer to implement this feature in the language itself. This means that the language's metalevel features must be sufficient not just to reify messages and argument lists, but also to intercede to change their semantics.

Mobile objects present a more complex example: implementing mobility as a library requires access not only to the implementation of message send but also to the underpinnings of the object storage subsystem. For efficiency, mobility also requires that the distinction between mutable and immutable objects be manifest in the language, and that the language does not provide an object-identity operation as a primitive.

### Failure

The defining characteristic of distributed computation on the Internet is partial failure: part of the program fails to satisfy its specification, while other parts continue to function. Failures are distinguished from exceptions in that the latter are part of the interface specification. Thus, a procedure that generates an exception condition might be fulfilling its specification.

Because a failure's exact nature is outside the scope of the specification, recovery requires us to pick over the failed computation's debris to see what, if anything, we can salvage and what cleanup is necessary. Only a program that has not failed can accomplish this task. Thus, we need some kind of watertight compartment that gives us a dry place to stand while trying to pump the water out of the failure. Operating systems provide such compartments — address spaces — but language designers have heretofore ignored them. The Internet programming language of the 21st century would do well to provide for failure recovery.

## Post-Post-Java

The only way to produce a language over which one programmer can have intellectual mastery is to start with a really tiny kernel and add on what 30 years of experience has shown to be absolutely necessary. Even then, we must be prepared to learn that our experience was inapplicable to the changing environment in which we work. We know that the opposite approach does not work: it is not possible to start with a large language and identify features to remove, because the features are invariably interdependent.

In addition to the fairly well-understood additions I have described here — failure-handling, distribution, reflection, and immutability — other additions that address serious issues in Internet computing are certainly worthy of consideration. Today, software quality and security are two of the biggest problem areas, and language designers are indeed working to create language features, such as universally quantified assertions and security types, to help address these problems. Perhaps, just as code safety in the Internet was the killer problem that lead to Java's adoption, the next language for the Internet might be driven by the need to improve software quality and security.

### Acknowledgments

I am indebted to Richard Staehli, for fruitful discussions about programming languages, and Jessica Black, for invaluable information about architecture. This article is a revised and extended version of a position paper originally submitted to the ECOOP Workshop on Object-Oriented Language Engineering for the Post-Java Era.

### References

1. E.S. Raymond, ed., *The New Hacker's Dictionary*, 3rd ed., MIT Press, 1996.
2. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.
3. D.H. Ingalls, "Design Principles Behind Smalltalk," *Byte*, vol. 6, no. 8, 1981, pp. 286–298.
4. R.K. Raj et al., "Emerald: A General Purpose Programming Language," *Software-Practice & Experience*, vol. 21, no. 1, 1991, pp. 91–118.
5. T. Budd, *Understanding Object-Oriented Programming Using Java*, Addison-Wesley, 2000.
6. F. Rivard, "Smalltalk: A Reflective Language," *Proc. Int'l Conf. Metalevel Architectures and Reflection* (Reflection '96), 1996, pp. 21-38.
7. L. Bak and G. Bracha, "Mixins in Strongtalk," presented at the Inheritance Workshop, European Conf. Object-Oriented Programming (ECOOP), 2002; www.cs.ucsb.edu/projects/strongtalk/pages/documents.html.
8. A. Hunt and D. Thomas, *The Pragmatic Programmer*, Addison-Wesley, 2000.

**Andrew P. Black** is a professor of computer science at the OGI School of Science & Engineering, Oregon Health & Science University. His research interests include distributed systems, programming languages, and programming methodology. He received his D. Phil from the University of Oxford. He is a member of the ACM. Contact him at black@cse.ogi.edu.