How the Object-Oriented Revolution Was Won

Allen Stix, Associate Professor
School of Computer Science & Information Systems
Pace University
Pleasantville, NY 10570-2799

Mary F. Courtney, Associate Professor
School of Computer Science & Information Systems
Pace University
Pleasantville, NY 10570-2799

Abstract

The authors conduct a whimsical interview with an historian of computing at ISECON 2050 and learn why it took Java to vault mainstream systems construction over the barriers to objects.

The historian explains that for object-oriented systems analysis and design to feel natural, a good amount of direct experience with objects is requisite. Coding is the only activity that provides actual experience with the nature and properties of objects. Java, much more than C++, expedites this because: (i) Java's libraries supply enforced demonstrations, and (ii) Java, because it disallows free functions, requires verbs to be nouns.

The serious intent of this paper is to explain why the switch to Java, even from C++, is worth the effort. Programming is the place for acquainting students with objects. This is one of the chief reasons for including programming in the curriculum for Information Systems.

Keywords: Java, Object-Oriented Programming, Curriculum

In comparison to the road to structured analysis, design, and coding; the road to objects has been long and slow. Remember Tim Rentsch's predictions, made in 1982? He got things right, except that it took into the 1990's to get where he thought we'd be in the 1980's:

> What is object-oriented programming? My guess is that object-oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.
>
> [Tim Rentsch; "Object-Oriented Programming" *SIGPLAN Notices*, Sept. 1982, v. 17, no. 9, pp. 51-57]

Stories of success with object technology have become common, and it is clear that objects will one day dominate the profession of computer information systems. But what will it take for the transition to be pervasive?

To find out, we obtained special authorization for time travel and non-participatory attendance at ISECON 2050. Thinking "...Intel inside, idiots outside...," the Dean agreed to foot the bill if we could verify, beyond all doubt, that we had actually gotten there and accomplished our investigative goal. We thought that ironclad proof might lie in becoming the first individuals in the year 2000 to know what superseded the object-oriented paradigm.

The following is a loose reconstruction of our interview with an historian of computing on how it came to be that objects prevailed. We had to find an historian because none of the other Johnny-come-lately educators and practitioners could believe that systems had ever been built without objects. We submit the following to verify our trip to 2050 and get the Dean to pay-up. I did the talking and the more serious co-author took notes.

* * *

Me: Hey guy! What thrills await Y2K information technologists? We do objects you know. What's next?

Historian: Objects started in the arena of programming, and it took the longest time for Twentieth-Century programmers to understand and use them. The conceptual underpinning was in place well before 1975. The first programming language with objects, Simula, appeared in 1967. The first theoretical treatise appeared in a slim, 1972 book by three giants, Dahl, Dijkstra, and Hoare. They described all the essentials of object-based and object-oriented programming. Their work was of a formal nature. Their formal examination of programming did not uncover techniques beyond abstraction and derivation; that is, they foresaw nothing beyond objects. It is hard to call the stark reality exposed through formal inquiry prescience, but they were right. The final paradigm for software, as far as we know, is the object-oriented paradigm.

Me: You say it took us a long time to understand objects. You've got to remember that serious work was hampered by daily struggles with Windows. Every fifteen minutes the program I'm trying to build hangs with another General Protection Failure.

Historian: The program YOU are building, hummm. Two obstacles made objects difficult for early practitioners understand. First of all, like that proverbial elephant described differently by the blind men standing at different places, object technology has different facets. Some descriptions focused on objects as abstractions or as abstract data types. Others would focus on objects in terms of inheritance. Thus, people were hearing different things from different quarters. The object proselytizers in the 1980's were not presenting a coherent picture. That their explanations relied upon analogies made things worse. No analogy was quite right. But speaking literally was no help either. Saying that an object is "a record with functions" did not help non-coders; and coders could not visualize how or why such operatively dissimilar constructs might be fused.

Looking back, it seems that concrete code was the only effective way to communicate: (i) how classes put instance variables and functions together, (ii) that classes are data types for use as type specifiers, (iii) that objects are instantiations of classes, (iv) how objects access their members, and (v) how one class inherits variables and functions from another class. C++ was very helpful in this regard. More than any other language, especially Smalltalk, it made the fundamentals of object-oriented programming accessible to professional programmers and to students. Of course, it was a nasty language in which to build real systems. Too many traps. That's why it was abandoned so quickly. But, as I said, it was of inestimable value in displaying an object as a thing that encapsulates both data and functions and in displaying how objects are put to work. Incidentally, functions in object parlance are termed methods.

Me: My colleague and I co-authored an asynchronous course in C++ that ran many times. We also delivered a paper on our observations pertaining to learning over the Internet. It's a lie that we tried to get a grant to research the Boolean, anti-binary, least squares approach.

Historian: That the fabric of C++ was not object-oriented was both a benefit and a drawback. Procedural programmers felt right at home with C++, however the language itself neither exemplified the use of objects nor had rules compelling coders to create classes of their own. The next major language, Java, did both of these.

But let me complete what I was saying about why objects were hard for early programmers to key into. Basically, early programmers were electrical engineering types or, at least, oriented toward bits and bytes. That, anyway, was the culture. The result was trouble dissociating the operation of the machine, at the hardware and assembler level, from the essence of a program as the symbolic manifestation of a process.

On the machine, computation takes place with instructions such as load, store, add, branch on zero, and the like. Notice that machine instructions are verbs; each instruction signifies an action. Values, on the other hand, are inert bit strings. They sit passively in registers, in memory, or in storage. They can neither modify themselves nor, like the machine, be asked to do things.

Early languages like FORTRAN built upon this paradigm with no explicit awareness that there could be other options. Subroutines operationalized the high level instructions one might have wished the machine had offered. Whatever a subroutine did, it was always to act: to compute a square root, to sort an array, to print a report. Data types were still inert, though the language managed them in particularized ways. A float, for instance, was stored as two separate bit strings, a value and a scaling factor. A character was an eight-bit integer that was displayed as a letter instead of as a number.

Later languages, like PL/I, Pascal, and C, offered more sophisticated subroutines and more sophisticated data types than FORTRAN; but they did not diverge from the machine language model of computation. Instructions acted; subroutines acted. Bit strings were acted upon; variables and records were acted upon.

Programmers internalized this model. So did system analysts and system designers. Software systems were built on the basis of the actions they would have to perform; and when an action entailed too much processing to be coded on a single page, it was decomposed into subordinate actions. Tens of thousands of otherwise fine minds were mutilated by this procedural model.

Me: With all due respect, the Y2K bug was the biggest non-event of the last millennium, right? If we were cognitively deficient, the culprit was Jolt. You know, "all the sugar and twice the caffeine." I am always wired.

Historian: The reverence for caffeine among computing professionals was immortalized by the name of the language that turned things around, Java. In much the way that C++ was "a better C," Java was an improved C++. Probably the greatest improvements were the simplifications owing to the fact that all objects were dynamically allocated and garbage was collected automatically. Along with this, all variables for objects were references (i.e. pointers that did not need to be dereferenced); assignment meant "copying an address"; and it was always the address of an object, passed by value, that was transmitted to actual arguments in methods. Also, templets were unnecessary because all classes were "Objects"; and by that I mean derived from the class at the top of the hierarchy. Apologies if I'm getting too technical.

While this is what attracted many C++ developers to Java, other aspects of the language made it attractive to forward-thinking educators.

Firstly, Java provided numerous illustrations of objects at work. To take apart a string, for example, students could instantiate a StringTokenizer object. The instantiated object would be dedicated to the string passed to its constructor, such as the three word string, "See Spot run." The programmer could access the object's method countTokens() which would return, in this case, the integer 3. Then, to capture the leftmost sub-string, the programmer could access the object's nextToken() method. This would return the string "See" and clip it from the string held within the object. After this, countTokens() would report that two strings remained. The next time that nextToken() was called on this object, "Spot" would be returned and removed from the contained data member.

More important than the StringTokenizer as a tool was the fact that it exemplified the nature of objects. A program could have any number of StringTokenizer objects. Each such object contained its own internal string about which it could answer the question, "How many 'tokens' does your string currently hold?" And each object could operate upon its internal string. Students were immersed in objects from early on. Through hands-on use, they intuitively understood that an object could be a software mechanism that held data (e.g. a string) and performed a specialized job (enabled its substrings to be culled for processing). Random was another class from which students instantiated objects. Each one of these could be asked for its nextDouble(), which it computed from the values it stored for its own sequence of pseudo-randoms.

Added to this, Java forced beginning students to create similar objects of their own. It did this by disallowing free functions. In FORTRAN, Pascal, or C where you'd have written a subroutine called sort(), in Java you'd have to design a class from which you'd declare a sorter object. This is how "verbs" became "nouns." Instead of thinking in terms of sort, getCustomerName, printReport you'd think in terms of sorters, nameGetters, and reportPrinters. Little software mechanisms that performed specialized jobs.

And the more of objects students experienced, the more apparent their versatility. An array in Java is an object. To adhere to tradition, arrays perform writes and reads from specified compartments with the bracket operator. But each array object supplements its sequence of elements with a scalar named length. Thus, when an array is passed to a method, its size goes right along as part of its corpus. This was an impressive application of encapsulation, that objects allow nouns to be better nouns.

Me: I, myself, would have named that language Szechuan, or possibly General Tso's Chicken. You're quite a Java proponent; mind telling me how you are related to James Gosling?!

Historian: Java had a tremendous impact. The age of modern computing dawned around the year 2000, when colleges and universities in large numbers began adopting Java as their backbone language. By the time students were sophomores, they were indoctrinated. That marvelous book by David Bellin and Susan Suchman Simone, *The CRC Card Book* (Addison-Wesley, 1997), was the bridge to OO analysis and design in the large. As the wave of Java-schooled students moved into industry, the software profession was transformed. Functional decomposition gave way to thinking about systems in terms of their parts, and catalogs of reusable and customizable parts became standard.

Me: Say no more! I'm getting tired of this, but our colleagues will be pleased to learn that in switching to Java from C++ they are doing the right thing.

****