

Has Object-Oriented Programming Delivered?

Greg Goth

Nearly 20 years after Bjarne Stroustrup unveiled the first workings of C++, the debate over the bottom-line benefits of object-oriented programming remains as heated as ever. Has OO programming delivered on its promises?

For the software engineer charged with maintaining an OO system hiding numerous bugs under layers of inheritance, probably not. However, for a merchant whose Java-enabled Web site has led to greatly increased sales and revenue, it certainly has. So context must be considered, but the discussion must continue.

The debate over the bottom line benefits of OO programming remains as heated as ever.

Fitting in between

"I think there is a lot of confusion, and the fact that you have to ask that question says on the one hand that it clearly hasn't been a total success—a clear win—and on the other, that some projects have been a success, so it's not a clear loss. It really does fit in between," says Vic Basili, professor of computer science at the University of Maryland and principal investigator for the Center for Empirically Based Software Engineering. "I think the issues are, what are the variables and the parameters that give us a clue about how and when it works?"

Basili contends that the intended users' thought processes must be considered when looking at OO programming's usefulness.

"It's not clear that OO is the right way to think in all domains. It's clearly good for

some, but we haven't had, and still don't have, enough empirical evidence one way or the other, and haven't done enough to identify key variables under which it's worth the transition and under which it's not."

Erik Stensrud, associate professor of information management at the Norwegian School of Management, went so far as to design an experiment meant to measure the productivity of OO language against a procedurally oriented language. Stensrud never ran the experiment, but did enough OO programming while working for the consulting firm Accenture that he pondered the question with some practical application.

Stensrud starts with the fundamental question, "What is good design?" His general answer is that it depends. According to him, "If your aim is testability, then OO dynamic binding is not necessarily the optimal choice. If your aim is program understandability and readability, then OO adds value by letting you express stuff on a higher abstraction level such as class hierarchies, but following program flow is no fun. If your aim is modularity, then I believe inheritance is good at the conceptual level—for example, in requirements analysis models—but not at the physical level because you break the encapsulation idea. So OO physical design should always 'use' instead of 'inherit.'"

"The crucial question is, does this logical modification require code changes to a single module or to many modules? If single, your design is right—if many, it isn't. Now, whether an OO design, a functional design, or a design separating data and operations is the right design is not obvious before you do

this test. I did a sensitivity analysis of a design once while I was with Accenture five years ago. I found out that parts of the design ought to be OO and other parts not, in order to make future modifications in as fast and cheap a way as possible.”

Lionel Briand, associate professor of systems and computer engineering at Carleton University in Ottawa, Canada, agrees that thinking through analysis and design is the crucial step many programmers do not do well when beginning an OO project.

“Many people use object-oriented languages and have very little idea about how to design the systems or even specify them. They may know the syntax of the language, but many of the systems have terrible designs. It’s improving over time—slowly, but it’s improving.”

Briand also says innovations such as design patterns have been successful because they provide programmers with sound recipes that teach the proper uses of OO concepts such as inheritance, polymorphism, and aggregation.

Problems with C++

For many observers, the bottlenecks might not be the responsibility of the OO concept, but rather that the dominant language in OO projects, C++, is not the best language for the job. The low-level roots of C++ in C are ill suited, some say, for the high-level demands of OO programming.

Consultant Bruce Webster, author of *The Pitfalls of Object-Oriented Development* (Hungry Minds, 1995), one the industry’s standard reference works on the subject, says, “I actually had two pitfalls surrounding C++. One was using C++ and the other wasn’t, and that pretty much sums it up.

“It is a brittle, complex language—largely the work of one person, Bjarne Stroustrup, working at Bell Labs trying to build C with classes. He was very keenly focused on issues of performance and size, which is what C was known for. The problem is, it’s a language on which a lot of programmers have impaled

themselves because it’s possible to go down an alley and reach a point where you’re not sure how to get out. In essence, what you have is ongoing revisions and extensions to the language, which seem to address some of the core fundamental flaws.”

Webster also says that despite the inroads other languages such as Java, Eiffel, and Smalltalk have made, C++ will remain a powerful force.

“It got out of the starting gate first, and entrenchment is a tremendous factor in IT,” he says.

Barry Boehm, TRW Professor of computer science at the University of Southern California, says, “There are a lot of different attributes you can get under the general umbrella of object-oriented technology,” he says. “Encapsulation and information hiding is generally a good thing, and classes and inheritance are generally a good thing unless you take them to excesses. Various languages support various combinations of these things. C++ added extra features to cover most of those, but it wasn’t really designed for modularity and information hiding. Languages like Ada and Java do a better job of that.”

Rahul Sharma, senior staff engineer at Sun Microsystems, says focusing strictly on language as an indicator of whether OO technology is delivering is an incomplete approach.

“When we say development, it’s never just about the language,” Sharma says. “It’s about the platform and libraries you have to build around that particular language. Defining the roles of the developer, the deployer, the tool vendor, and so on, as those roles are clearly specified in the J2EE platform, for example, becomes a significant advantage if each of those roles are meeting their responsibilities.”

Veteran programmer Thomas Niemann, who teaches computer science at Portland Community College in Oregon, says some problems lie with C++, but he also has reservations with some of OO programming’s key concepts. “There is a basic problem with the object-oriented methodol-

ogy and it has to do with excessive information hiding,” he says.

Empirical quagmire

It might be simple to explore the ultimate payoff of OO technology if definitive empirical studies existed. Alas, Basili says, this work is still ongoing, still incomplete and might be beside the point.

“I think it’s all piecemeal,” he says. “People continue to look at what’s useful and what measures might be valid, but that’s not the main issue here. I think the main issue isn’t whether we have the right measures to help us understand what we just did, but whether we can think differently, whether that thought model is appropriate for all problems, and whether people are ready, willing, and able to almost engage in the empirical paradigm.”

“As long as it’s my opinion versus your opinion, and we’ve had success and you haven’t, and we don’t know why we did and why you haven’t, we’re not going to get very far.”

Sharma believes the old ways of studying, quantifying, and improving development and productivity have forever changed because of the Internet’s birth as the dominant conduit for communications and that OO technology is tailor-made for this era.

“In terms of software engineering, prior to the time of C++ and Java becoming popular, there was more focus on the waterfall model. Projects went through certain stages, and the process was very structured around those vertical stages,” he says.

However, Sharma also says market demands have changed the way software must be conceived and developed.

“If you’re looking at the kind of projects that happened in the dot-com environment, time to market was very critical. You had to build functionality incrementally. You didn’t have to address all the requirements in one go. In that case, you needed to tailor the process to the rapidly evolving technology and it’s fairly lightweight.”

OO programming’s modularity, extensibility, and reusability attributes

are well suited for these quickly developed products, Sharma says.

"You can use those characteristics and apply them so you develop each of these new technologies in a faster turnaround time. You could do it with a non-object-oriented language, but you get minimum reusability, extensibility is constrained, and modularity cannot be expressed the same way as in an object-oriented language."

Niemann's concern with OO technology lies mainly in the concepts of inheritance and polymorphism, which can introduce extra levels of information hiding that make testing and maintenance more painstaking than it should.

"In the final analysis, you want to write code that is easy to read and easy to maintain. However, if you excessively hide information it will actually be more difficult to find and isolate bugs. This is the main problem I have with object-oriented programming. In fact, when I maintain code somebody else has written with this method, I must frequently view several different files to arrive at the place I need to go, whereas in procedural programming, which has information hiding but to a lesser degree, locating information is much more straightforward.

"I'm not advocating total abstinence from object-oriented programming; there are places it works and works well. Its just that the whole-

sale abandoning of procedural techniques is not the direction we should be going."

Several empirical studies partially bear out Niemann's observations. *An Empirical Study Evaluating Depth of Inheritance on the Maintainability of Object-Oriented Software*, published by researchers at the University of Strathclyde in Great Britain in 1996, concluded that an experiment with three levels of inheritance was easier to maintain than one with none or with five levels. A 1999 paper by researchers at the University of Southampton, also in the United Kingdom, *Experimental Assessment on the Effect of Inheritance on the Maintainability of Object-Oriented Systems*, concluded that systems without inheritance were easier to maintain than systems with either three or five levels of inheritance.

Yet Carleton University's Briand says the issue isn't whether to use inheritance, but rather to go back to sound design. "The reason there's an argument depends on whether you use inheritance properly for the right problem," he says.

Many design patterns, for example, use inheritance to anticipate change to lower the future costs. "Design patterns are designed to anticipate certain kinds of changes. It's true that for those kinds of changes, they facilitate maintenance. Of course, if you misuse

inheritance, you'll probably create more inconvenience and difficulty than advantages during maintenance," Briand says.

What's a procedural language?

The arguments over OO programming might fade as the generation of programmers who started with procedural languages leaves the workforce, replaced by graduates of collegiate programs that have emphasized OO languages.

At Carleton, students learn OO analysis, design, coding, and testing.

"It's partly a generational thing," Briand says. "The change in paradigm is so huge, it's very hard to learn on the job and most companies invest very little in training. It's not like you can learn object-oriented design and analysis in one day in a consultant's course. It's not possible. You learn the buzzwords."

At USC, Boehm says courses are taught largely in Java, and are increasingly using less C++. Niemann says a similar situation exists in Portland.

"We are teaching several classes in C++ and Java. There's only one course in C, and it is actually not a requirement for any of the other courses. It's just a service class for students who want to learn a bit of programming. There's a strong trend toward object-oriented programming in industry, and I feel colleges, to serve industry, should teach this technique. However, I feel we're throwing the baby out with the bath water, and there's a lot to be said for procedural programming. I've seen it work in large-scale applications and work well. Unfortunately, it would be career suicide to champion procedural techniques."

This possible mass lack of historical perspective worries industry veteran Webster.

"Luke Hohmann, who wrote *The Journey of the Software Professional*, (Prentice Hall, 1996) told me a few years ago he was hiring for a software company he was at and interviewing somebody who had a computer science degree. The applicant said she had never programmed in

Useful URLs

Lionel Briand's publications, including many papers on object-oriented programming: www.sce.carleton.ca/faculty/briand/briand_pub.shtml

Thomas Niemann's article detailing some shortcomings of object-oriented programming in *Embedded Systems* magazine: www.embedded.com/1999/9908/9908feat1.htm

Links to object-oriented programming forums, papers, and arguments: www.geocities.com/tablizer/oopbad.htm

The International Software Engineering Research Network bibliography of technical works, including experiments on object-oriented software: www.iese.fhg.de/network/ISERN/pub/isern_biblio_tech.html

anything but Java. We both found that scary because she had no basis to compare what she was doing with anything else.”

Yet Briand says these worries might be for naught.

“Most new systems will be object-oriented, no question about it. There

will be well-designed object-oriented systems and poorly designed object-oriented systems. Who’s developing new systems today in C? Nobody.”

Aviation Software Guidelines

Kelly J. Hayhurst and C. Michael Holloway

Regulatory authorities in the US and Europe have received two documents crucial to aviation software developers. One clarifies existing software guidelines for airborne systems and equipment certification, and the other provides new guidance for nonairborne communication and navigation systems.

The first document is the *Final Report for Clarification of DO-178B Software Considerations in Airborne Systems and Equipment Certification*. The RTCA (formerly an abbreviation for Radio Technical Commission for Aeronautics) and the European Organization for Civil Aviation Equipment (EUROCAE) approved it in October 2001 as DO-248B (EUROCAE document ED-94B is the European equivalent).

DO-248B is intended to clarify, not add to, existing DO-178B guidelines. The document contains typographical corrections, minor wording changes, frequently asked questions with brief responses, and discussion papers that provide longer clarifications. The FAQs and discussion papers address a range of issues, including safety, previously developed and commercial off-the-shelf software, and structural coverage.

The second document, *Guidelines for Communication, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance*, contains guidance for nonairborne CNS/ATM software as a counterpart to DO-178B. It discusses system aspects relevant to CNS/ATM systems and is intended to be an interpretive guide for applying DO-178B to these systems. Also, the

document provides direction for areas peculiar to CNS/ATM systems. For example, it has guidelines for adaptation data used to tailor elements of a CNS/ATM system at a particular site. The document also expands on the COTS software guidelines in DO-178B because COTS software is often used extensively in CNS/ATM systems.

The RTCA approved the document as DO-278 in March 2002 (ED-109 is the EUROCAE equivalent). The FAA is working on a strategy to implement DO-278 as part of the FAA’s acquisition process for CNS/ATM systems.

Evolving guidelines

RTCA and EUROCAE developed the original DO-178 guidelines in 1982 and added additional details in 1985 to produce DO-178A. The December 1992 revision, DO-178B, is the de facto standard for development and assurance of software for commercial transport aircraft or engines.

Since 1992, software developers and certification authorities have raised a number of questions about DO-178B content and application. In response, the RTCA and EUROCAE established a joint committee—known as RTCA Special Committee 190/ EUROCAE Working Group 52, or SC-190—to answer these questions and ensure that users consistently apply the guideline’s original intent. The committee was directed to develop position papers clarifying unclear sections of DO-178B and to prepare a list of issues to consider in future guideline revisions.

From 1996 to 2001, SC-190 met twice yearly, alternating locations be-


tween the US and Europe. Committee membership included representatives from the US, Canada, Europe, India, and Australia.

In addition to the two new documents, SC-190 compiled a list of issues to be considered when DO-178B is revised to produce DO-178C. The FAA anticipates that a DO-178C committee will be formed in 2004.

Who makes the rules?

The Code of Federal Regulations specifies the rules governing aircraft certification in the US, including rules requiring that aircraft systems meet their intended function, do not negatively affect other aircraft systems or functions, and are safe to operate.

To meet these rules, nongovernment aviation groups such as the RTCA and EUROCAE provide a forum through which government and industry develop consensus-based recommendations for aviation issues. Recommendations often take the form of proposed guidelines or performance specifications. The recommendations become official guidance if regulatory authorities, the FAA in the US and the Joint Aviation Authorities in Europe, accept them.

Copies of DO-248B and DO-278 are available for purchase through the RTCA (www.rtca.org) or EUROCAE (www.eurocae.org). 

Kelly J. Hayhurst is a senior research scientist at the NASA Langley Research Center. Contact her at k.j.hayhurst@larc.nasa.gov.

C. Michael Holloway is a senior research engineer at NASA Langley. Contact him at c.m.holloway@larc.nasa.gov.