Software [R]evolution: A Roundtable

Software improvements continue to pervade the growing interconnected web of computers and communications. But are these improvements merely evolutionary or responsible for what some have been calling a software revolution? *Computer* turned to some of the very brightest in the field to find out.

Kirk L. Kroeker
Computer

Software improvements continue to pervade the growing interconnected web of computers and communications. Some might argue, though, that while hardware developments have typically been able to keep the pace set by Moore's law, software has lagged, making only incremental evolutionary changes instead of revolutionary ones. Others would argue that software technology has come of age and that we have entered a new era where improvements in development strategies, languages, architectures, and interoperability will no longer be linear, but exponential.

It is with these thoughts in mind that *Computer* turned to some of the very brightest in the field to ask their opinions about the course of software development. The format of the roundtable was fairly straightforward. We asked the participants about some of the recent developments in their areas of expertise and invited their opinions about the industry as a whole. And of course we asked them to do a little soothsaying on the near future of technology.

It is with great pleasure that we bring to *Computer*'s readers the ideas of

- **Larry Wall**, creator of Perl,
- **David A. Taylor**, CEO of Enterprise Engines and creator of the *business engine* idea,
- **Chris Horn**, CEO of IONA Technologies and the creator of Orbix,
- **Paul Bassett**, senior vice president of Netron and creator of frame technology,
- **John K. Ousterhout**, CEO of Scriptics and the creator of Tcl,
- **Martin L. Griss**, principal lab scientist in the computer research center at Hewlett-Packard Laboratories,
- **Richard Mark Soley**, chairman and CEO of the Object Management Group,
- **Jim Waldo**, distinguished engineer at Sun Microsystems and lead architect for Jini, and
- **Charles Simonyi**, chief architect at Microsoft Research.

The answers we received in response to our questions varied widely, with participants commenting on everything from scripting languages and integration technologies to frameworks and distributed computing. As always, we welcome your feedback. Direct your responses to computer@computer.org. ❖

Kirk L. Kroeker is staff editor of Computer. *Contact him at k.kroeker@computer.org.*

An Ongoing Revolution

Larry Wall
O'Reilly and Associates

I think software design has been revolutionary all along—to the extent that anything in this business is revolutionary. Hardware design *seems* to have been progressing faster, but if you factor out the science of miniaturization (from which both hardware and software have been getting a free ride), hardware design hasn't been doing any better than software design. And, while we tend to think of hardware as more basic than software, a lot of our progress in hardware design is really progress in the software used to design the hardware. Plus, I just don't buy the notion that doubling the number of address lines constitutes revolutionary progress in hardware design.

Even if we factor in miniaturization, hardware design still gets a free ride from the fact that putting twice as many lines on a chip gives you four times as many tran-

0018-9162/99/$10.00 © 1999 IEEE

sistors—two dimensions for the price of one. But even on a more abstract level, the problem space of hardware tends to be of low dimensionality. We hold hardware to a lower standard than software: When we make a computer run twice as fast with the same instruction set (or maybe even a reduced instruction set), we say it's twice as good. But when we make software twice as good in any particular dimension, we just throw more dimensions into the problem space and then blame the software for not keeping up. It's a bum rap.

On the flip side, programmers too often feel like they're responsible for all the sins of the world. Programmers should spend less time trying to kick butt and more time trying to kick back. If programming isn't fun, we're not going to convince more people to become programmers.

SCALING BETTER FOR GROWING PROBLEMS

The most revolutionary thing about language design these days is that we're putting more effort into larger languages. This is a direct result of the ubiquity of computers. People don't want to learn a new little language every time they run into a new interface. The successful languages of today aren't the little languages but rather the larger languages that can be *used* as if they were little languages. Then you not only leverage your previous knowledge, but your language can scale better as your problem grows. And you know it will.

Real languages evolve, they don't revolve. Truly revolutionary computer languages tend not to catch on. The most interesting recent developments have to do with cross-cultural communication of every sort. The whole world is becoming a connectionist computer. Certainly XML is going to help us translate between various data formats, but we're also working on making programming easier for the billions of potential programmers in the rest of the world.

Development versions of Perl now support Unicode, not just as data but also as program. We're thinking about how to add more training wheels to Perl. Perl already lets you choose your level of discipline, but I see that we can do a lot more to help both the novice and the expert. But especially the novice who wants to become an expert.

NOT TECHNOLOGICAL, BUT SOCIOLOGICAL

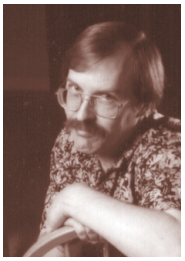
I hope we're not looking for some groundbreaking developments. That'd be like asking for groundbreaking developments in the English language. Our software problem is not primarily technological, but sociological. Innovation happens. But unless innovation is connected to our cultural heritage, it belongs to the problem set rather than the solution set. Ubiquitous computing requires ubiquitous programming. Ubiquitous programming is of necessity popular programming. And the populace is not yet with the program. Most people can't even program their VCRs.

Computer science still seems to be looking for the magic bullet that will cause people to write correct programs without having to think. Instead, we need to teach people how to think. And there's where computer culture is really falling down. We've forgotten the importance of teaching computer phonics. We've fallen into the whole-language learning fallacy that our elementary schools are only just now recovering from (and some people will never recover from). The problem with whole-language learning is that the people who would learn anyway can learn it that way, but the people who need to learn via phonics are abandoned. So it is with computers. There are millions of potential programmers out there who are not capable of learning C++ or Java merely by soaking in it. They need to muddle around with simple concepts first. They need to babble baby talk before they defend their dissertations before the local OO fanatic. They need to sing in the shower before they get voice lessons.

It's the *culture* of computing that needs to have a developmental revolution. Ontogeny needs to recapitulate phylogeny. That is, individual programmers need to go through the same learning process that computer science as a whole has gone through. Programmers need to learn symbolic programming before they learn structured programming, structured programming before modular programming, modular programming before object-oriented programming.

I realize this view may be sacrilege to today's revolutionaries. Call me a reactionary, but Perl didn't succeed by discarding the lessons of the past. One of those lessons is simply that computers should help people learn when they're ready to learn and otherwise stay out of their faces. Most programming is not mission-critical. ❖

Larry Wall



Larry Wall created the Perl programming language, the rn newsreader, and the ubiquitous patch program. He was the recipient of the System Administrator Guild's 1994 Outstanding Achievement Award and the Free Software Foundation's 1998 Free Software Award. He is currently employed by his publisher, O'Reilly and Associates, who pay him to take care of Perl and the Perl community, and to show that there's more on the open source bandwagon than just the band. He is currently working on the third edition of Programming Perl *(also known as the Camel book), but finds himself spending an inordinate amount of time working up new comedy routines for keynote addresses, interviews, and roundtable discussions like this one. He received a BA in natural and artificial languages from Seattle Pacific University. Contact him at larry@wall.org.*

Programming for Everyone

David A. Taylor
Enterprise Engines

I don't see anything very exciting going on in scripting languages. We have VBA in the Windows world, JavaScript from the platform-independent side of the industry, and a few other big yawns. They all seem to be stuck in that awkward compromise—too hard for nonprogrammers to do much with and too wimpy to be real programming languages.

What really bothers me about this situation is that we now have the ability to do something spectacular on this front. Object technology introduced all the lexical innovations we need to allow nonprogrammers to give simple instructions to computers. Think about it for a moment. A message is just an imperative sentence. The receiver is the subject of the sentence (technically, the vocative), the method name is the verb, and the parameters are the direct and indirect objects. Consider the sentence, "Accounting, give Smith & Co. a credit of $420 on Invoice 1234." That sentence maps word for word onto a corresponding Smalltalk message, with values substituting for parameter names.

We created objects in our own image, and they interact just like we do. After going to all the trouble to do that, we should leverage what we've built and allow business people to give simple instructions to their computers, preferably through a speech-recognition interface. Then we'd be out of that awkward compromise. We'd have a scripting language that business people could actually use, and it would be so remote from what we think of as programming that professional programmers wouldn't even view it as relevant.

OBJECT TECHNOLOGY NO LONGER A BUZZWORD

The most exciting development in object technology is that it's no longer a buzzword, which must mean it's good for something now. You aren't with it these days unless you talk about component software or agent technology and explain why these things are better than objects. Java, which is one of the best object languages ever written, plays to the Web and to platform independence, with hardly a mention of the O word. Even *Object Magazine* changed its name to *Component Software* to maintain its cutting-edge image.

When I first realized this was happening, I was ticked off because I thought the software industry was moving past objects just when they were finally ready for prime time. Then I remembered a prediction I made about 15 years ago, that by the year 2000 no one would talk about objects any more because the technology would be so thoroughly absorbed into the mainstream that no one would think to mention it. I guess maybe my prediction came true.

RELEASING THE POWER OF THE COMPUTER

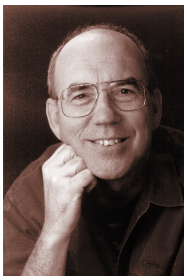
There is one big breakthrough that I have been working toward most of my career, and I think we're getting close. I'm talking about releasing the power of computer technology to ordinary business people. We've put a computer on just about every manager's desk, and we allow them to tinker a little with databases and spreadsheets. But if they want to make even the slightest change in the way they do business, they have to go to IS and stand in line for a couple of years.

This goes back to my earlier comments on scripting languages. We have the ability to let people give simple spoken instructions to their computers. If a manager dictates a series of instructions and then indicates when those instructions should be carried out in the future, she has just programmed her computer. Once she can make a business change on the spot like that, do you think she'll ever go back to the IS department?

Yeah, I see you grimacing. Believe me, I know all the reasons why this is a really bad idea. It takes years of training to write correct programs. End users create havoc even without this ability. Security would be an even worse nightmare if we ever did this. And so on. None of these are adequate reasons not to take the obvious step forward. If you look at the history of technological innovation, our most life-changing technologies have started out being restricted to trained operators, then they really caught on once they were released to the unwashed masses. Computers may seem different, but it is a difference of degree, not kind.

The revolution that was started with the introduction of PCs won't be complete until the people who own those PCs can use them to change the way their business works. This is one of those things that's hard to

David A. Taylor



David A. Taylor is the founder and CEO of Enterprise Engines. His chosen mission is to create a new generation of software he calls business engines, which he defines as executable business models that can be dynamically modified to adapt to changing conditions. After five years of development and seven prototypes, his vision is now being realized in his company's flagship product, the Enterprise Engine. He is the author of six books, including Object Technology: A Manager's Guide *(Addison Wesley Longman, 1998). Before entering the business arena, he was a professor of psychology at the University of Rochester in New York and was one of the founders of the field of cognitive science. Taylor received a PhD in cognitive psychology from the University of California at Irvine. Contact him at dtaylor@engines.com.*

see now but is going to be really obvious in retrospect. I think when we look back on this period, we will wonder why it took us so long to figure it out. ❖

Making Software Work Together

Chris Horn
IONA Technologies

Middleware is solving complex integration challenges and enabling the delivery of exciting new applications and solutions. Many of the pieces are in place in terms of the infrastructure and the supporting services. The next great technological direction for middleware is ease of use, which will be driven by two factors: the continuing popularity of component development models such as CORBA, COM, and EJB and the growing need for access to this technology by a wider section of the development community.

But there is a blizzard of competing technologies—such as MOM, ORB, OTM, TPM, EAI, EJB, DCOM, COM+, application server, and Web application server—that all use different approaches but promise to solve the same problems. Because of this, there is currently a risk of complete confusion for the developers and their organizations in the short term. What approach do they use? What tools do they need? What standards do they support? What is the end goal?

THREE CORE AREAS

In order to determine where we are, we need to think of middleware as made up of containers, components, and connectors. In this landscape, the container provides an environment where application components can be deployed. These components can be based on standards such as CORBA, EJB, or COM, or they can represent existing monolithic systems—not legacy systems because I believe there's no such thing as legacy, just working systems. The container supports a range of services including cross-platform coverage, asynchronous messaging, security, transactionality, manageability, deployment support, and extensibility.

Within the component segment, it's important to distinguish between application-level components and system-level components. The former are plugged together to build application systems and are governed by a number of standards such as CORBA, EJB, and COM, while system-level components are hidden from the user and are part of the architecture of the container. The component layer offers great advances in ease of use and development.

Finally, connectors provide bridges between the container and components and other systems and environments, such as third-party packaged applications. These three areas provide a context within which to view and evaluate the different elements involved in any middleware solution, which hopefully provides a respite from the profusion of different terms and approaches.

BRINGING SYSTEMS TOGETHER

Developers and organizations are realizing the inherent value in bringing their IT systems together. The core value of middleware is that it offers an evolutionary approach to advancing IT. It preserves existing investment in systems, skills, and tools and enables the extension of those resources over time. Middleware and "Making Software Work Together"—IONA's motto—are being driven by the understanding that it makes sense to bring different departmental systems together.

You need the ability to extend your applications across the network and to change those applications and systems to meet changing market demands. Evolution meets revolution in the approach to developing these systems. In tandem with leveraging existing skills and technologies, the world has arrived at components. The widespread adoption of CORBA, EJB, and COM underlines middleware's success, and the use of these different models even across the same project highlights why it's growing in popularity.

In addition to tying together new and existing systems and applications, organizations are faced with accommodating these new models. This clearly illustrates the value middleware brings to the equation: It offers support for the technology you are using today and provides a path to future developments.

Although we have already made great advances in terms of making the design, development, and deployment of middleware solutions easier,

Chris Horn

Chris Horn is chairman and CEO of IONA Technologies, which he cofounded in 1991. He was the initial developer of Orbix, has headed the company since its inception, and has guided its growth to the top of the middleware market. He chairs the new Expert Group on Future Skills, part of Ireland's Education Technology Investment Fund, which plans to invest IR£350 million into education in Ireland. Prior to founding IONA, he taught computer science at Trinity College in Dublin, where he participated in many pan-European IT research projects involving distributed computing. He also worked in Brussels for the European Commission and was an integral part of the 10-year Esprit program designed to improve the continent's technology industry. Horn received a PhD in computer science from Trinity College. Contact him at chorn@iona.com.

there is still some room to go. I believe the next 12 months will see the delivery of server-side tools that bring the same ease-of-use to enterprise server-side development that developers enjoy today on the client side with tools like Visual Basic and Visual Café. The delivery of these tools will drastically reduce the barriers to building high-end application services like transactions, and it opens middleware to a new level of market acceptance. This is a core priority at IONA.

The other trend is the increasing use of different component models across organizations. Middleware vendors need to be aware of the need for native support of these models in any middleware solution, and their advocacy underlines why middleware will continue to be a core technology for the future. ❖

Extracting Useful Patterns

Paul Bassett
Netron

Exponential improvements are already old hat in many domains. They happen whenever you can automate the slow, error-prone aspects of programming, such as when high-level specifications—like which database schema to use—logically entail their implementation codes. Whereas Moore's law ultimately fails due to immutable physical laws, software technology improvements are limited only by our capacity to extract useful patterns from apparent novelty, which itself can be improved by software.

MOVING FROM INVENTION TO CONVENTION

A new paradigm can solve problems you think are impossible in the current paradigm. An example is framework technology, which routinely achieves 90 to 95 percent reuse. But a new paradigm takes a generation or more to go from invention to convention. OO is more than 30 years old! For most of that time, the paradigm is all but invisible: "How come I never heard of you?" Then comes the impossibility stage: "I've heard of you but don't believe you." Then there's the quizzical effect, where we see the new way but interpret it in old ways. Finally, with real understanding comes mainstream acceptance.

Today I'm seeing mostly stage three—the quizzical effect—so things are getting there. To finish the job we must replace the craft mentality with the engineer's ethos. In software shops and houses, the ideas of asset management and flexible manufacturing processes don't compute. On the other hand, the size and complexity of systems are increasingly beyond what older paradigms can handle. Having invested heavily in reusable assets, managers will insist on a sustainable return. That demand will turn the tide.

CLASSIFYING LEGACY CODE

One potentially revolutionary development is the means to automatically classify legacy code. The classifier finds similar functions wherever their code fragments happen to be and determines their input, output, and local variables.

Finding novel patterns of similarity is thought to be computationally hard—compute time goes up exponentially with input volume. The breakthrough came with the invention of a way to fingerprint Cobol code so that the patterns emerge in linear time. Our tool, called HotRod, classifies hundreds of thousands of lines of code per hour. And if all you want to do is find the business rules, it is even faster. Because the important bits of old code are otherwise so hard to find, the advent of this tool makes the migration of large legacy systems a viable proposition.

Trying to predict breakthroughs is not this fool's game. And we already have more innovation than we can usefully absorb. The real issue is sifting the wheat from the chaff.

This already difficult problem is exacerbated by a belief in silver bullets (needed to win so-called religious wars) and magic pills (otherwise known as vaporware). Yet ignorance and skepticism also abound to the point where—at the risk of mixing too many metaphors—some really innovative babies get tossed out with their bathwater.

Cost-effective systems, delivered on time and on budget have never gone out of style. Just so, they have never become routine either. You want genuine innovation? Aye, there's the rub. ❖

Paul Bassett

Paul Bassett cofounded Netron and is the company's senior vice president of research, where he helps IT organizations shift to a flexible software manufacturing paradigm. For his invention of frame technology, he received the Information Technology Innovation Award from the Canadian Information Processing Society. His book, Framing Software Reuse: Lessons From the Real World *(Prentice Hall, 1997), was described by Ed Yourdon as "the best book about reuse I've seen in my career." He is also a coauthor of the IEEE's forthcoming Standard P1517: Software Reuse Life Cycle Processes. He is a member of* Component Strategies' *editorial board and a lecturer in the Computer Society's Distinguished Visitor Programme. He received an MSc in computer science from the University of Toronto and the ISP designation from CIPS. Contact him at pbassett@netron.com.*

Integration: A New Style of Programming

John K. Ousterhout
Scriptics

A revolutionary change is occurring in the way people build computer software. It used to be that every application was a monolithic entity built from scratch, but today more and more applications are built by integrating existing resources.

The infrastructure of a modern business contains countless applications, devices, protocols, data sources, and frameworks. For an organization to run efficiently, all of its resources must be coordinated, and it must be easy to create new applications that leverage the capabilities of existing resources. Furthermore, for a business to introduce new technologies, it must be able to integrate them with existing systems.

THE RISE OF SCRIPTING LANGUAGES

As the importance of integration applications has increased, so has the use of scripting languages such as Tcl. Traditional system programming languages such as C, C++, and Java are poorly suited to the integration task. Their emphasis on compilation and strong typing makes it difficult for them to accommodate the tremendous variety and rapid evolution that characterize integration applications.

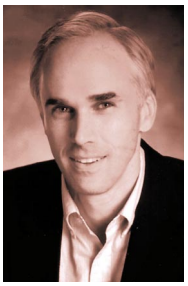
In contrast, scripting languages are interpreted and weakly typed, which allows integration applications to be developed five to 10 times faster than with system programming languages. As a result, scripting languages will be used for a larger fraction of application development in the years ahead.

INCREASING SOPHISTICATION

Although they have existed for several decades, scripting languages in recent years have seen dramatic improvements in their level of sophistication. They were originally used for less critical applications, such as text processing and report generation, but modern scripting languages such as Tcl are sophisticated enough to support much larger and more mission-critical applications.

New developments over the next year or two will make scripting languages more enterprise-ready than ever before. For example, the latest release of Tcl supports Unicode for international applications, and it is thread-safe to allow usage in high-performance server applications. Tcl also has support for XML, ActiveX, CORBA, and other key enterprise protocols. Although these improvements are evolutionary, the cumulative effect is that scripting languages can be used for a huge new class of applications.

John K. Ousterhout



John K. Ousterhout is CEO of Scriptics, which he founded in 1998 to promote Tcl as a platform for integration. He is the creator of the Tcl scripting language and the Tk toolkit and spends much of his time raising awareness of this new style of programming. His interests include scripting languages, Internet programming, UIs, and OSs. Before founding Scriptics, he was a professor of computer science at the University of California at Berkeley and a distinguished engineer at Sun Microsystems. Ousterhout is a Fellow of the ACM and has received many awards, including the ACM Software Systems Award, the ACM Grace Murray Hopper Award, and the UC Berkeley Distinguished Teaching Award. He received a PhD in computer science from Carnegie Mellon University. Contact him at ouster@scriptics.com.

Upcoming developments in scripting languages will continue to be evolutionary rather than revolutionary. However, one development that is likely to be particularly significant is the integration of scripting languages with XML.

XML promises to revolutionize the interchange of information within and between organizations, but it provides only a static data representation. By itself, XML only describes things. However, a scripting language can provide the active component to bring XML to life. Scripting languages can easily extract data from XML, translate XML to other representations, interface XML to existing applications, and take actions based on the data in an XML document.

The string-oriented nature of scripting languages makes them a perfect match to the loosely structured string nature of XML. Open source Tcl extensions for manipulating XML documents are already available, and commercial tools will be coming from Scriptics soon. I believe that scripting languages will provide the vehicle of choice for manipulating XML, and the combination of scripting languages and XML will dramatically increase the use of both. ❖

Domain Engineering and Reuse

Martin L. Griss
Hewlett-Packard

The notion of components has certainly caught on. No longer do we simply talk about object libraries. Instead, we have discussions about COM, COM+, CORBA, JavaBeans, and EJBs. And UML has become a pervasive software blueprinting language with many vendors providing tools support—even

Microsoft provides a component-oriented subset in the form of Visual Modeler. Larger-grain components, in the form of business objects or business components, are being seriously defined by OMG and others.

So components and scripting are becoming the standard by which large-scale enterprise development will be judged. All these technologies, taken together, could radically change the way people do reuse, even though none is revolutionary by itself.

COMMONALITY, VARIABILITY, AND FLEXIBILITY

I now believe that agents combined with a workflow language will be the next step in increased flexibility and reuse. Just as JavaBeans have grown into enterprise JavaBeans, so business components will incorporate the notion of agents, providing specific interfaces and using services that enhance their autonomy and robustness. Furthermore, in order for collaborations of business components to provide well-understood support for business processes, they will interact according to a local and global workflow. Thus, to choreograph these collaborations explicitly and flexibly, workflow languages will become the next generation of scripting languages.

Also, we've come to realize that you can't simply take random things and reuse them in a context for which they weren't designed. And you can't just design something to make it reusable all by itself. You have to think about how a group of components will fit together, how they will collaborate, and how these components may be individually variable.

Domain engineering is about looking for commonality and variability. Express it somehow and then take advantage of it when you build the components. The work Ivar Jacobson and I did on reuse was to add some extensions to UML elements that we call variation points, which can be thought of as a generalization of extension points or "hot spots" that allow you to state that some models are variable without saying how they vary.

INCREMENTAL INVESTMENT

There is clearly a rich stream of technology innovation, such as enhancements to Java, Web tools, and generators. Then there is aspect-oriented programming, new pattern and modeling tools, and so on. Yet success with large-scale reuse is not just technology; architecture and standards are a social process. The decision to reuse rather than invent from scratch involves culture, education, management, and professional incentives.

Many companies have a rather short-term silver-bullet view that limits investment in architecture, component development, domain engineering, and reuse. To change behavior like that, there must be some kind of publicized success that shows it really works and that it pays off. There *are* successes, but I don't think we publicize them well enough. You don't pick up an issue of *Computer* every month and see a discussion of the latest and the greatest success.

Another element is reducing the cost of entry. By having good components available to cover more of the space you don't have time to fill yourself, you can do all the up-front architecting and domain engineering pretty easily. Many people believe that it's an all or nothing thing. Either you do it properly with domain engineering or you don't bother. But that's not true. You can get incremental benefit for incremental investments.

More companies seem to be realizing that they can make domain-specific components according to one or another standards. But things are still very complex. I think, though, that we are able to follow a two-tier strategy. That is, our technology allows some fairly sophisticated components to be written and used with much less sophistication. I'm working on a paper now that is about moving from components and scripts to agents and workflow—in other words, more competent components, a richer framework, and a more stylized scripting language. ❖

Martin L. Griss

Martin L. Griss is principal laboratory scientist for software engineering at Hewlett-Packard Laboratories. He currently researches model-driven, agent-based application management, OO reuse, and domain-specific kits. Before that, he was an associate professor of computer science at the University of Utah, where he is currently an adjunct professor. With Ivar Jacobson and Patrik Jonsson, he coauthored Software Reuse *(Addison Wesley Longman, 1997). He has authored over 45 papers, an HP reuse handbook, and many technical reports. He is a member of the Sigsoft Executive Committee and the joint IEEE/ACM Software Engineering Education Project (SWEEP). He received a PhD in physics from the University of Illinois. Contact him at griss@hpl.hp.com.*

Thought Converging

Richard Mark Soley
Object Management Group

While it would be nice to say that there have been revolutionary changes in the computing milieu over the past decade, and there will be in the future, it's hard to believe such a thing. Most, and possibly all, advances in computing have stood on the shoulders of past giants. Or, if not stood on the shoulders, at least stepped on the toes.

THE SOFTWARE DEVELOPMENT EXPERIENCE

Nevertheless, I do believe there is a trend that will effect changes in the software industry. That trend is an

evolutionary convergence of several streams of thought—not just object technology, but the component concept, tools to support pervasive modeling, and round-trip design and implementation. All of these ideas add up to a software development experience that will—finally—better support reuse and integration, which are, after all, the same thing.

From the point of view of the Object Management Group, this is where we have been going for 10 years, from our infrastructure middleware (CORBA) and modeling language (UML) to standards targeting vertical markets from health care and telecommunications to life sciences, manufacturing, and transportation.

WELL-STRUCTURED GLUE

I could only claim the label "evolutionary" for the emerging component software marketplace. In fact, OMG wasn't formed to introduce standardized object software, but rather standardized software interfaces that happen to leverage distributed object technology. This is the evolutionary change: Object technology has become absolutely mainstream; distributed object technology to integrate heterogeneous systems is becoming an everyday phenomenon, which means that finally systems integrators can consider building systems from legacy components in finite time.

This has led, for example, to a whole new class of systems integrators, typically called "enterprise application integration" companies, which have leveraged the convergence of distributed objects and software modeling to deliver applications from legacy parts in reasonable time frames.

What was exotic middleware software a few years ago is showing up in really critical systems—hospital records, satellite trajectory control, cable set-top boxes, national air traffic control systems, worldwide retail management, high-availability international news feeds, and border-crossing immigration management. All of those settings, for example, feature deployed, mature CORBA solutions. This tells me the evolutionary concept of well-structured "glue" to paste together existing applications has caught on.

I really don't see any major groundbreaking achievements in the middleware area any time soon. I see more the integration of concepts. For example, I see Java's portability converging with CORBA's interoperability and UML's modeling capabilities to make systems integration a more common activity (rather than from-the-ground-up systems generation).

Standards in these areas—including repository integration standards and leveraging evolutionary ideas like XML—mean we can concentrate on getting the job done rather than generating yet more infrastructure. In the same way we saw "yacc" and "lex" allowing compiler writers to ignore the syntactic parsing part of the job and concentrate on language design and code generation, we will see standardized parts and converging processes that finally allow us to concentrate on so-called business logic—the functionality of the system we're building. ❖

Richard Mark Soley

Richard Mark Soley is chairman and CEO of OMG, where he focuses on a wide array of activities from management of an active corporate board to a very busy consensus-making process to the technical details of standards like CORBA, UML, and MOF. At OMG, he directs the OMG technical processes, including all subcommittees, task forces, and SIGs. Previously, he cofounded and was chairman and CEO of AI Architects, maker of the 386 HummingBoard and other PC and workstation hardware and software. Prior to that, he consulted for various technology companies and venture firms on matters pertaining to software investment opportunities. Soley received a PhD in computer science and engineering from the Massachusetts Institute of Technology. Contact him at soley@omg.org.

Portability Is Key

Jim Waldo
Sun Microsystems

The single most revolutionary trend that I've seen over the past few years has been toward languages that are portable. Not portable in the old sense, in which being portable meant that you could recompile on a new platform and start debugging immediately, but the idea that the binary form of a program can be run on multiple platforms. The most popular language with this property has been Java, but it has hardly been alone—Tcl has many of the same properties, and the Inferno platform also has portable code.

The revolutionary aspect of portability is that it breaks the link between the binary form of a program and a particular processor. This in turn means that code can be moved around a network of machines without worrying about the kind of processor or operating system on the machine where the code ends up. This changes all the rules of distributed computing. It was the one factor that allowed us to do the Jini system, but that is just a start to what can be done with mobile code.

REJECTING COMPLEXITY

A subtler form of revolution is also taking place. This is the revolution that rejects complexity in favor of simplicity. One mark of this revolution has been the move from C++ to Java. When I talk to people about Jini, I spend a lot of time describing how much time we spent

making the system as simple as possible. People are beginning to understand that simplicity is required for reliability. The big wads of software that we have grown used to might be replaced by small, simple components that do only what they need to do and can be combined together. It's a move from the Baroque to a Bauhaus style, and one I think we all need to embrace and encourage.

Part of this revolution is a trend toward language-based systems. It used to be that the language you used to develop a system or component was an accidental property, and everything was written assuming that it needed to talk with things written in other languages. This led to a great deal of complexity and often weakened the connections between systems to the passing of lowest-common-denominator data types. But we are now seeing systems that require a particular language. Because of that, we are able to use the rich type system of the language to describe interactions. This makes the overall system considerably more simple while increasing its expressive power.

COMBINING TECHNOLOGIES

The ability to move code around the network was the real revolutionary step that led to Jini. But there were other advances, both evolutionary and revolutionary, that helped make it possible. A revolutionary step was the notion of a safe language and a safe execution environment. Moving code is only part of the story; you need to have confidence that the code you move will not cause problems when it is run. This requires that the language not allow programmers to do things that can cause damage. The damage can be either planned or unintentional. A virus is just a programming error that was planned. Getting a safe language started was a way of making programs more reliable, but in the end it was an enabler of the kind of code motion we needed to do Jini.

An evolutionary step was the joining of a strongly typed language with a dynamic language. These two have generally not been combined in the past. There was no particular reason why they couldn't have been, but dynamic languages usually were weakly typed, and strongly typed languages were statically linked. Taking the step of combining the two allowed the kind of safety we needed with the plastic runtime environment that is usual in dynamic languages. For distributed systems that need stability—but which are also constantly changing—the combination was a natural.

The next couple of years could be very interesting. People are just beginning to grasp that they don't have to tie the code that runs on a processor tightly to that processor, which means that a computer can now use a network to get code rather than a local disk. This greatly increases the scope of what we think of as a computer. It is no longer just desktop systems or servers, but also includes personal digital assistants, cell phones, our cars, and very soon most of our appliances and home entertainment equipment.

Most of us haven't had to program for these things in the past, and we certainly haven't thought much about what to do with a network of such things. New ways of exchanging both information and behavior are going to be thought of, and new patterns of interaction will be tried out. The world has suddenly become much richer than what we have been used to, and our usual techniques aren't going to be adequate. I think it is going to be a lot of fun. ❖

Jim Waldo

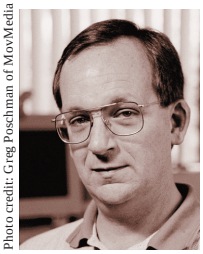


Photo credit: Greg Poschman of MovMedia

Jim Waldo is a distinguished engineer at Sun Microsystems, where he is the lead architect for Jini, a distributed programming system based on Java. Prior to Jini, he worked in JavaSoft and Sun Microsystems Laboratories, where he did research in OO programming and systems, distributed computing, and user environments. Before joining Sun, he spent eight years at Apollo Computer and Hewlett-Packard working in distributed object systems, UIs, class libraries, text, and internationalization. While at HP, he led the design and development of the first ORB and was instrumental in getting that technology incorporated into the first OMG CORBA specification. He is an adjunct faculty member at Harvard University, where he teaches distributed computing in the department of computer science. He received a PhD in philosophy from the University of Massachusetts at Amherst. Contact him at waldo@sun.com.

The Future Is Intentional

Charles Simonyi
Microsoft Research

Intentional Programming (IP) is the most exciting thing happening today in software engineering. IP is not a new language like Java or a new programming technique like OO. IP is simply an OS for abstractions, a new category of metatool that coordinates the cooperation of independently developed abstraction objects—called intentions—so the programmers' illusion will be a continually extending and improving integrated program development environment.

IP achieves this illusion by building the programs out of intention instances, just as a CAD system builds the plans for a complex artifact out of objects. The methods of the IP intentions are not executed at runtime—which would be just plain OO—but instead are called by the IP system at programming time whenever IP encounters an intention instance. The functionality of the intentions is expressed by the transformation

method of the intention, which translates an intention instance into the traditional computer science primitives. Through the action of additional methods, the intention can customize other aspects of program development, from source control to editing, source checks, browsing, testing, and debugging.

REPACKAGING PROGRAMMING

The effects of this repackaging of programming practices will be both evolutionary and revolutionary. On the evolutionary side, existing legacy investment in software can be preserved: The initial set of intentions can express the abstractions of the legacy languages, their *unparsers* can perfectly mimic the traditional notations, and their transformations can result in performance and compatibility characteristics that are identical to the legacy implementation. This is unlike previous revolutions in software engineering in which programmers were typically asked to distance themselves from legacy code and to accept new trade-offs.

The revolutionary aspects of IP are also unprecedented: IP can start a cycle of innovation by independent developers who, for the first time, will have a realistic chance to contribute a new abstraction, a new notation, or a new transformation to the software engineering marketplace. New intentions, or new methods for an existing intention, can be created, priced, distributed, used, and improved as independent artifacts.

In the past, cycles of innovation were frequently fueled by the separation and intermediation of system components.With the advent of personal computer OSs, the cycle of independent innovation starts. The results of the independent development have been great improvements over the starting point. To see that the separation of concerns was a necessary condition of the improvement one only needs to ask, "Could any organization focusing mainly on the word processing problem have accomplished what the entire PC industry has actually accomplished?"

INNOVATING INTENTION

IP creates an analogous separation and intermediation. From the point of view of the programmer-user, the decision whether to use an innovative intention can be made on the margin: The legacy investment will not be in jeopardy. Users will be able to experiment with radically new notations and views of their programs—without having to change the code—just by turning on the newly purchased unparsers. The great improvements in program performance reported by the transformational programming community would become available to every programmer.

Most importantly, new abstractions could be used without having to rethink or rewrite legacy code. Additionally, if improvement in existing code is desired, IP can be helpful there, too. The transformation algorithms can be used as editing tools: Legacy patterns can be found mechanically and replaced by new abstractions. Pieces of code can be mechanically lifted into new procedures. Implementations of new abstractions can be chosen to be compatible with the existing structures so the source improvements can be done incrementally.

From the point of view of programming innovators, there are two irresistible draws to IP: first, the potential of the mass market described above; second, the unprecedented room for innovation. Part of this freedom stems from the ability of the intentions' methods to execute arbitrary code. The other part is the richness of the methods' interface to IP. The range of notations that can be generated is unusually wide.

We can only speculate on what the cycle of innovation in software engineering might eventually bring. In the shorter term, we will probably see familiar abstractions and notations whose novelty will be just that they will be available to all. Pre- and postconditions, enumerators, or procedure specialization will be available to everyone, no matter what their legacy language might have been. Next, the emphasis will shift to domain-specific abstractions, which will greatly simplify problem expression in their respective domains.

Because most problems extend over multiple domains, IP's ability to mix abstractions will be critical to effective use of high-value abstractions that incorporate much domain knowledge and are therefore domain specific. IP will also help open up many new domains including the subdomains of the field of software engineering.

Without IP, the payback from any one of these domains by themselves would not have been sufficient to justify the development and use of a particular domain-specific language. ❖

Charles Simonyi

Charles Simonyi is chief architect at Microsoft Research, where he is responsible for new approaches in programming technology. He joined Microsoft in 1981 to start the development of microcomputer application programs and hired and managed teams who developed Excel, Word, and others. Before coming to Microsoft, he worked at Xerox PARC, developing Bravo, the first WYSIWYG editor. Simonyi received a PhD in computer science from Stanford University. In 1997, he was elected to be a member of the National Academy of Engineering for "his contributions to the development of widely used desktop productivity software." Simonyi has held endowed chairs for Public Understanding of Science at Oxford University, Theoretical Physics at the Institute for Advanced Study at Princeton, and Educational Technology at Stanford. Contact him at charless@microsoft.com.