Objects Have Failed

Notes for a Debate

Richard P. Gabriel

Narrative

"Objects have failed."

What can it mean for a programming paradigm to fail? A paradigm fails when the narrative it embodies fails to speak truth or when its proponents embrace it beyond reason. The failure to speak truth centers around the changing needs of software in the 21st century and around the so-called improvements on OO that have obliterated its original benefits. Obsessive embrace has spawned a search for purity that has become an ideological weapon, promoting an incremental advance as the ultimate solution to our software problems. The effect has been to brainwash people on the street. The statement "everything is an object" says that OO is universal, and the statement "objects model the real world" says that OO has a privileged position. These are very seductive invitations to a totalizing viewpoint. The result is to starve research and development on alternative paradigms.

Someday, the software we have already written will be a set of measure 0. We have lived through three ages of computing—the first was machine coding; the second was symbolic assemblers, interpreter routines, and early compilers; and the third was imperative, procedural, and functional programming, and compiler-based languages. Now we are in the fourth: object-oriented programming. These first four ages featured single-machine applications. Even though such systems will remain important, increasingly our systems will be made up of dozens, hundreds, thousands, or millions of disparate components, partial applications, services, sensors, and actuators on a variety of hardware, written by a variegated set of developers, and it won't be incorrect to say that no one knows how it all works. In the old world, we focussed on efficiency, resource limitations, performance, monolithic programs, standalone systems, single author programs, and mathematical approaches. In the new world we will foreground robustness, flexibility, adaptation, distributed systems, multiple-author programs, and biological metaphors for computing.

Needless to say, object-orientation provides an important lens through which to understand and fashion systems in the new world, but it simply cannot be the only lens. In future systems, unreliability will be common, complexity will be out of sight, and anything like carefully crafted precision code will be unrealistic. It's like a city: Bricks are important for building part of some buildings, but the complexity and complicated way a variety of building materials and components come together under the control of a multitude of actors with different cultures and goals, talents and proclivities means that the kind of thinking that goes into bricks will not work at the scale of the city. Bricks are just too limited, and the circumstances where they make sense are too constrained to serve as a model for building something as diverse and unpredictable as a city. And further, the city itself is not the end goal, because the city must also—in the best case—be a humane structure for human activity, which requires a second set of levels of complexity and concerns. Using this metaphor to talk about future computing systems, it's fair to say that OO addresses concerns at the level of bricks.

The modernist tendency in computing is to engage in totalizing discourse in which one paradigm or one story is expected to supply all in every situation. Try as they might, OO's promoters cannot provide a believable modernist grand narrative to the exclusion of all others. OO holds no privileged position. So instead of Java for example embracing all the components developed elsewhere, its proponents decided to develop their own versions so that all computing would be embraced within the Java narrative.

Objects, as envisioned by the designers of languages like Smalltalk and Actors—long before C++ and Java came around— were for modeling and building complex, dynamic worlds. Programming environments for languages like Smalltalk were written in those languages and were extensible by developers. Because the phi-

losophy of dynamic change was part of the post-Simula OO worldview, languages and environments of that era were highly dynamic.

But with C++ and Java, the dynamic thinking fostered by object-oriented languages was nearly fatally assaulted by the theology of static thinking inherited from our mathematical heritage and the assumptions built into our views of computing by Charles Babbage whose factory-building worldview was dominated by omniscience and omnipotence.

And as a result we find that object-oriented languages have succumb to static thinkers who worship perfect planning over runtime adaptability, early decisions over late ones, and the wisdom of compilers over the cleverness of failure detection and repair.

Beyond static types, precise interfaces, and mathematical reasoning, we need self-healing and self-organizing mechanisms, checking for and responding to failures, and managing systems whose overall complexity is beyond the ken of any single person.

One might think that such a postmodern move would have good consequences, but unlike Perl, the combination was not additive but subtractive—as if by undercutting what OO was, OO could be made more powerful. This may work as a literary or artistic device, but the idea in programming is not to teach but to build.

The apparent commercial success of objects and our love affair with business during the past decade have combined to stifle research and exploration of alternative language approaches and paradigms of computing. University and industrial research communities retreated from innovating in programming languages in order to harvest the easy pickings from the OO tree. The business frenzy at the end of the last century blinded researchers to diversity of ideas, and they were into going with what was hot, what was uncontroversial. If ever there was a time when Kuhn's normal science dominated computing, it was during this period.

My own experience bears this out. Until 1995, when I went back to school to study poetry, my research career centered on the programming language, Lisp. When I returned in 1998, I found that my research area had been eliminated. I was forced to find new ways to earn a living within the ecology created by Java, which was busily recreating the computing world in its own image.

Smalltalk, Lisp, Haskell, ML, and other languages languish while C++, Java, and their near-clone C# are the only languages getting attention. Small languages like Tcl, Perl, and Python are gathering adherents, but are making no progress in language and system design at all.

Our arguments come in several flavors:

1. The object-oriented approach does not adequately address the computing requirements of the future.
2. Object-oriented languages have lost the simplicity—some would say purity—that made them special and which were the source of their expressive and development power.
3. Powerful concepts like encapsulation were supposed to save people from themselves while developing software, but encapsulation fails for global properties or when software evolution and wholesale changes are needed. Open Source handles this better. It's likely that modularity—keeping things local so people can understand them—is what's really important about encapsulation.
4. Objects promised reuse, and we have not seen much success.

5. Despite the early clear understanding of the nature of software development by OO pioneers, the current caretakers of the ideas have reverted to the incumbent philosophy of perfect planning, grand design, and omniscience inherited from Babbage's theology.
6. The over-optimism spawned by objects in the late 1990s led businesses to expect miracles that might have been possible with objects unpolluted by static thinking, and when software developers could not deliver, the outrageous business plans of those businesses fell apart, and the result was our current recession.
7. Objects require programming by creating communicating entities, which means that programming is accomplished by building structures rather than by linguistic expression and description through form, and this often leads to a mismatch of language to problem domain.
8. Object design is like creating a story in which objects talk and interact with each other, leading people to expect that learning object-oriented programming is easy, when in fact it is as hard as ever. Again, business was misled.
9. People enthused by objects hogged the road, would not get out of the way, would not allow alternatives to be explored—not through malice but through exuberance—and now resources that could be used to move ahead are drying up. But sometimes this exuberance was out-and-out lying to push others out of the way.

But in the end, we don't advocate changing the way we work on and with objects and object-oriented languages. Instead, we argue for diversity, for work on new paradigms, for letting a thousand flowers bloom. Self-healing, self-repair, massive and complex systems, self-organization, adaptation, flexibility, piecemeal growth, statistical behavior, evolution, emergence, and maybe dozens of other ideas and approaches we haven't thought of—including new physical manifestations of non-physical action—should be allowed and encouraged to move ahead.

This is a time for paradigm definition and shifting. It won't always look like science, won't always even appear to be rational; papers and talks explaining and advocating new ideas might sound like propaganda or fiction or even poetry; narrative will play a larger role than theorems and hard results. This will not be normal science.

In the face of all this, it's fair to say that objects have failed.

Failure to Embrace Failure

L. Peter Deutsch outlined "Seven Fallacies of Distributed Computing" sometime during the 1990s.

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero

Interestingly, these fallacies also apply to monolithic, single-computer systems. Here's how:

- The network is reliable—within a single program, calling a procedure or invoking a method may not work because the interface wasn't understood by the developers, the procedure or method is incorrect, data is semantically bad, etc.
- Latency is zero—a procedure might not return due to failure. In multithreaded programs, a thread may die or take a long time, etc.
- Bandwidth is infinite—long data copy times, infinite loops, loops going too long based on bad data.
- The network is secure—if some library is written by someone else...
- Topology doesn't change—data-driven programming, dynamic method dispatch...
- There is one administrator—code can be written by lots of different people and installed haphazardly or by mistake.
- Transport cost is zero—using data can cost.

But the real point is that the Seven Fallacies point out that failures can happen in a network, and they can happen in monolithic code too. The thrust of much of the last 30 years of computer science has been to make it so that failures can't happen—by using more and more static notions, by making programming languages more strict, and by heavyweight methodologies, to name a few.

When we program a distributed system, we need to embrace failure by writing code that is resilient in the face of failures and that can repair itself when it notices things are going wrong. To do this well in a distributed setting requires practice, but we are taught that it is bad to practice this in our regular programming.

Failures are called exceptions, while in the real world failures are the rule. Although one view of objects is that they are autonomous entities with identity and even a sense of self-preservation, in fact the realm of object-oriented programming has been taken over by the static thinkers who have imposed types and static notions on an initially dynamic approach to software construction.

Failure to Embrace Self-Healing

Self-healing is necessary for the future where applications will be increasingly distributed on the net. Pieces of functionality will be damaged or removed, possibly by people making mistakes while doing sys admin work or by failures during upgrades or the development of new or upgraded code.

Consider this: If a Java API is altered so that it changes packages, there is no Java-approved way of making the upgrade in applications except for recoding. The way that Java is defined, packages that are "official" are part of the Java package. But, for some domains, there needs to be experimentation to get the APIs right, and perhaps several experiments. And sometimes those experiments need to take place in the world of real applications. Ah, but those applications cannot be easily altered when the API becomes official and changes packages. This, therefore, hinders experimentation.

Self-healing requires, perhaps, diversity and redundancy, if not a statistical basis. Note that in some key biological systems, structures that carry information are also food, and the mechanism that moves information from one place to another is actually the one that moves food. We need to be looking at other paradigms to find mechanisms to help us understand and make self-healing systems.

Failure to Fight Off the Static Thinkers

- OO is dynamic—what's with these static guys taking over?
- "Dijkstra is renowned for the insight that mathematical logic is and must be the basis for sensible computer program construction"—it can be argued that sometime around 1970–1980, the pseudo-mathematicians within CS took over. This started with Dijkstra and Parnas, who wrote later, in 1986:

  > *Ideally, we would like to derive our programs from a statement of requirements in the same sense that theorems are derived from axioms in a published proof. All of the methodologies that can be considered "top down" are the result of our desire to have a rational systematic way of designing software.*
  >
  > –A Rational Design Process: How and Why to Fake It, IEEE Trans. on Software Engineering, Feb. 1986

  This is based on a fundamental error: that the structure of the process for creating something matches the structure of the thing created. In this case, there is a mistaken hidden belief that a proof is constructed by starting with the axioms and moving forward through the steps of the proof. This error can be traced back to the Greek philosopher Pappus.
- "The strong typing of object-oriented languages encourages narrowly defined packages that are hard to reuse. Each package requires objects of a specific type; if two packages are to work together, conversion code must be written to translate between the types required by the packages." [John K. Ousterhout]
- What the static types are trying to do is to make sure that a program cannot fail at runtime. Yet, for living systems to be created, they must be able to sustain a failure and repair itself. This is part of a doomed attempt to eliminate errors and write perfect systems. OO people understood this but allowed—though how could they have stopped it?—the static types (meaning, a certain breed of computer scientist) to take over their languages, starting with C++, Eiffel, and Java.

❧ ❧ ❧

> *The key question is whether typing has reduced the expressiveness of languages. I guess that my experience says "yes." This is because type systems are not generally able to describe the sophisticated abstractions that we want to build. Although these abstractions may be sophisticated, that does not mean they are impossible to understand—they are often quite intuitive. But by their very nature, type systems are required to be able to efficiently and automatically prove assertions about programs, and this will tend to impose limitations on what kinds of things a type system can do. We also do not want our types to be larger than our programs, so this imposes limits on their complexity. One thing that is not recognized enough is that types are partial specifications, and it might be nice to allow different levels of detail in your types, without having to resort to "object" (the type with no detail).*
>
> *The second question is whether the loss in expressiveness is worth the gain in "safety." I would say right now that the answer is no, in part because Java and C# even lack any form of generic types (type parameters). This forces you do cast everywhere, which defeats the purpose of the type system, or build things like .NET CollectionGen (http://www.sellsbrothers.com/tools/) to generate wrappers that do the casts for you.*
>
> –William Cook

> . . . all *marshaling-based type checking is done at run-time.*
>
> –Chris Sells, http://www.sellsbrothers.com

As you point out, static typing is tied to the notion of designing up front. It is also connected with fear. I have to say that I really like types, in general. But I don't like them when the prevent me from expressing myself. Or, as is more likely but much more subtle, when they cause a language designer to distort a language to suit the needs of the type system.

–William Cook

Failure to Fight Off the Syntax Freaks

The early OO languages—Smalltalk and let's say the Lisp-based ones—had very simple syntaxes, but later ones like C++ and Java became very heavy with syntax.

- Where is simple, user-level programming?
- Where is there respect for form and not only structure? One could argue that making some part of the language unassailable—operators for example—renders the language too brittle and inflexible in many ways that limit its ability to be used, and contributes to making the language more suited for structural programming rather than expressive programming.

Failure to Tell the Truth About Reuse

Reuse is largely a failure. The primary reason is that making things reusable requires extra work, and there is no real incentive to do it. Moreover, people seem to think that reuse comes for free with OO languages, but this is a mistake from reasoning about implementation inheritance.

> *Every manager learns that reuse requires a process of reuse or at least a policy. First, you need to have a central repository of code. It doesn't help if developers have to go around to other developers to locate code you might be able to use. Some organizations are small enough that the developers can have group meetings to discuss needs and supplies of code.*
>
> *Second, there has to be a means of locating the right piece of code, which usually requires a good classification scheme. It does no good to have the right piece of code if no one can find it. Classification in the world of books, reports, magazines, and the like is a profession, called cataloging. Librarians help people find the book. But few software organizations can afford a software cataloger, let alone a librarian to help find the software for its developers. This is because when a development manager has the choice of hiring another developer or a software librarian, the manager will always hire the developer.*
>
> *Third, there must be good documentation of what each piece of code in the repository does. This includes not only the interface and its purpose but also enough about the innards of the code—its performance and resource use—to enable a developer to use it wisely. A developer must know these other things, for example, in order to meet performance goals. In many cases such documentation is just the code itself, but this information could be better provided by ordinary documentation; but again, a development manager would prefer to hire a developer rather than a documentation person.*
>
> *For a lot of pieces of code it is just plain simpler to write it yourself than to go through the process of finding and understanding reusable code. Therefore, what development managers have discovered is that the process-oriented world of reuse has too many barriers for effective use.*
>
> –Patterns of Software, rpg

> *However, the form of reuse in object-oriented languages hardly satisfies the broad goals of software development. What I want to suggest is a better word than reuse and maybe a better concept for the reuse-like property of object-oriented languages.*
>
> *The word (and concept) is compression. Compression is the characteristic of a piece of text that the meaning of any part of it is "larger" than that piece has by itself. This is accomplished by the context being rich and each part of the text drawing on that context—each word draws part of its meaning from its surroundings. A familiar example outside programming is poetry whose heavily layered meanings can seem dense because of the multiple images it generates and the way each new image or phrase draws from several of the others. Poetry uses compressed language.*
>
> –Patterns of Software, rpg

The Failure of Reuse

By Jack Ganssle
Embedded.com
(12/14/01, 07:15:51 PM EDT)

Reuse is the holy grail of software engineering, one that is so entrenched in our belief system no one dares to question its virtue. The quest for reusable components is one of the foundations of object-oriented programming and all of the tools and languages that OOP has spawned.

Yet, my observations suggest that at least in the embedded space reuse has been a dismal failure.

First, let me define "reuse." We can talk about carrying-over code, or salvaged code, both of which imply grabbing big source files and beating them into submission till a new product appears. Not to knock that; it's much better than starting from scratch each time. But real reuse, though, means leaving the freakin' source code alone. We're in reuse nirvana when we're able to pluck a module from the virtual shelf and drop it in, unchanged.

Martin Griss and others have observed that a module isn't really reuseable till it's been reused three times. No matter how good our intentions, the first time we try to reuse something we discover a facet of the new problem the old module just can't manage. So we tune it. This happens a couple of times till the thing is generally reusable. That's not because we're stupid; it's simply because domain analysis is hard. No one is smart enough to understand how a function might get used in other apps.

It's expensive to do a forward-looking design of a function or module. You'll always save money in the short term solving today's very specific problem, ignoring the anticipated demands of future projects. If you elect to pursue a careful program of reuse your projects will initially come in late and over-budget.

Reuse is hard. It's like a savings account. My kids complain that if they stick a few bucks in the bank then that's money they can't use -- and it's just a piddling sum anyway. Sure. The value of savings comes after making regular deposits. Ditto for reuse. The cost is all up-front; the benefits come from withdrawals made in later years.

Sure, we all know the long-term outweighs today's concerns, but I'll betcha most bosses won't agree. They'll usually buy into the idea of the benefits of reuse, without being willing to stand the pain of creating the reuseable components. And that's the rub. When the ship date looms closer, most bosses will tell us to toss out any sort of discipline that has long-term benefits in pursuit of a near-term release. So, in practice, reuse often fails since schedules generally dominate over any other parameter.

I also suspect most of us don't want to reuse code. Seventy percent of RTOSes are homebrew, despite at least 100 commercial -- free and otherwise -- products on the market. Nothing is easier to beg, borrow, or buy than an RTOS, yet most of us still refuse to practice even this most simple of all reuse strategies. Why is this? I've heard many specious arguments (too expensive—yet there are plenty of freebies; or we don't trust the code—but some are safety certified, etc), plus a few really good arguments (it's all legacy code; no one is willing to make risky changes).

I saw numbers recently that suggested 20% of embedded TCP/IP stacks are homemade. Surely some very few embedded apps do need a highly customized protocol stack. But for the rest of us writing our own must be one of the most incredibly irresponsible wastes of talent and dollars imaginable.

I think aggressive reuse is our only hope of salvation from the morass of expensive and unreliable code. Building correct firmware is so difficult that we along with our bosses have a fiduciary responsibility to find ways to make it reuseable.

But we're not doing it. Sure it's hard. Yes, it's initially expensive. And of course we cannot reuse everything; a lot of what we build will always be inherently unreusable, like hardware drivers that get tossed with each new spin of the design.

Why is reuse such a failure?

In IEEE Transactions on Software Engineering, Volume 28, Issue 4 (April 2002):

> *Failures were due to not introducing reuse processes, not modifying non-reuse processes and not considering human factors. The root cause was the lack of commitment by top management, or nonawareness of the importance of these factors, often coupled with the belief that using the object-oriented approach or setting up a repository would automatically lead to success in reuse.*
>
> –Success and Failure Factors in Software Reuse, Maurizio Morisio, Michel Ezran, Colin Tully

> *I think you need to make a stronger argument for OOP=reuse. I think it really is, but it needs to be clear from the above. And there's reuse over time (by the same project team), and reuse of a stable OO subsystem by different teams (e.g. MFC). I think those are different, and they've both failed. The former because OO hierarchies don't tend to remain stable—they're constantly having methods added, changed, etc. It's rarely a case of creating new interfaces for implementations. And in the latter case (using something like MFC) there's better success, but the reality is that unless the OO hierarchy has a lot of users and mileage behind it, it typically isn't that usable except for very narrow domains. Also there's <William Cook's> argument about OO systems accumulating a lot of mechanism that works at runtime rather than compile time.*
>
> –Warren Harris

Failure of Encapsulation

Encapsulation is hiding implementation behind an interface. This enables an implementor to monkey with the implementation without clients knowing. In OO, objects provide the ultimate in encapsulation, since either the method signature or the message is all that someone using the object can know.

It fails when there are global properties that need to be maintained by a group of encapsulations, and when the realities of evolution where wholesale changes need to be made.

> ***Encapsulation:*** *the problem is that encapsulation is fantastic in places where it is needed, but it is terrible when applied in places where it isn't needed. Since OOP enforced encapsulation no matter what, you are stuck. For example, there are many properties of objects that are non-local, for example, any kind of global consistency. What tends to happen in OOP is that every object has to encode its view of the global consistency condition, and do its part to help maintain the right global properties. This can be fun if you really need the encapsulation, to allow alternative implementations. But if you don't need it, you end up writing lots of very tricky code in multiple places that basically does the same thing. Everything seems encapsulated, but is in fact completely interdependent. This is related to the notion of aspects.*
>
> –William Cook

> *Interfaces certainly provide encapsulation due to their level of abstraction, but I don't really think this is their primary function. They're all about gaining leverage through generalization and abstraction. They're the OO equivalent of higher-order functional programming. This generalization is something you would want within a running program, and it is not focused on the evolution of the program over time.*
>
> *I believe that* ***encapsulation has failed because it is all about program evolution****, and in real life, evolution doesn't take place in the ways we expect it to. A very good programmer might write a very good class (with public, protected and private state and methods spelled out just so) only to find that the requirements have changed drastically in the next release, and the class need to be further factored, generalized, parameterized or virtualized to accommodate the new requirements. Over a number of these cycles, the class may stabilize, but the very nature of the dynamics of OOP seems to be making these sorts of refactorizations. Encapsulation fails because it ends up protecting the programmer from themselves -- a lot of extra typing, code movement, etc. -- without anything valuable coming from it. Moreover, even if encapsulation is preserved (as with the polar/rectangular example), there may be other dimensions of the code that are not "encapsulated" such as the performance implication on the overall program (imagine the performance impact on changing your window system to deal with polar points with all sorts of trig operations nicely encapsulated under the covers).*
>
> ***The truth of the dynamics/evolutionary situation is that you need all the code in front of you so that you can massage it into the new form you want it to take*** *(which is why open source succeeds where encapsulation does not). Minimizing outside dependencies is a good thing, but doing this at the class level is usually just too fine grained. I think that's why many people still resort to standard C functions when making a public interface for their C++ program (or they use IDL or some other interface formalism to minimize the collection of outward-facing methods). But when it comes to massaging the internals, encapsulation just gets in the way.*
>
> –Warren Harris

Abstraction is layering ignorance on top of reality.

–rpg, at the Debate

Failure to Improve Software Development

From Lisp and Smalltalk development environments we've generally taken a step backward. GUI development has improved quite a bit, but what goes on under the covers is still hard. Most developers use just some form of text editor and a basic compiler-like system.

Patterns have sprung up as an attempt to capture some design and domain knowledge, but they are not widely used (sadly). Further, frameworks seemed at first to hold a lot of promise, but in practice they have proven too brittle and awkward to use. In the original Smalltalk (and Lisp) environment(s), the environment itself was the basic framework along with a variety of smaller internal ones. Thus the process of development itself was one of essentially redecorating one's development environment. This is much more dynamic than a build process, which is what C++ and Java require.

With Agile Methods, the OO crowd has begun to talk about what development is really about, but overloading OO languages with static notions has reverted the mainstream of OO to the debilitating style it has been for many years. There are no good development environments for C++ and Java, though there are for Smalltalk and Lisp.

Failure to Tell the Truth About Design

- It's not just characters talking to each other, there are things like law of physics, etc

Model-driven programming is a little like the use of macros in Common Lisp in which the form of the language is adapted to the domain rather than having to dream up configurations of objects and patterns of communications. To Lisp/Scheme people, the GoF book is a joke because the patterns there are simply covering for lack of programming and abstractional power.

William Cook: I want to bring the programs closer to the design....

❧ ❧ ❧

Peter Norvig:

Design Patterns in Dylan or Lisp

- 16 of 23 patterns are either invisible or simpler, due to:
- First-class types (6): Abstract-Factory, Flyweight, Factory-Method, State, Proxy, Chain-Of-Responsibility
- First-class functions (4): Command, Strategy, Template-Method, Visitor
- Macros (2): Interpreter, Iterator
- Method Combination (2): Mediator, Observer
- Multimethods (1): Builder
- Modules (1): Facade

One can argue that the popularity of design patterns means that programmers need to write code where in other languages they could abstract over more things, use first-class types, etc.

❧ ❧ ❧

Again, Agile methods are finally trying to tell the truth about design: That when designing a new thing, you need to be building it too. Here is why:

Agile methods seem to all be solving a pair of simultaneous problems: creating in an artistic mode while at the same time doing careful engineering. Developing software systems is a creative activity requiring the techniques of science, engineering, and art; and software, unlike art, is also required to perform via execution on (networked) computer hardware. Let's look at these two problems a bit.

Unless a particular system is being implemented afresh after many successful implementations, its actual requirements cannot be determined until its creators are in the midst of its creation, and users of the software cannot know what they desire until they can see what is possible and what it is like to use it. In most cases, interesting new software presents numerous creative challenges, and there are no special ways to handle them in software. Creative activity requires identifying some triggers—facts, thoughts, initial approaches, metaphors, actors, roles, snippets of code, snippets of design, reminders, half-remembered thoughts while listening to customers—doing some construction—be it writing lines or sentences, making some brush strokes,

performing some initial cuts—and then finding more or better triggers and refining what is already there. Every act of creation requires triggers.

> *...art is simply making things in one's own way, guided by the skills and inclinations at hand, experiences, the materials at hand, and the triggers that present themselves. Triggers play the role that most see as creativity or self-expression. A trigger is any thing, place, person, rhythm, or image that presents itself, or metaphor that comes to mind that leads the maker to make a work; often the trigger appears in the final work, and if the work contains a lot of private triggers, it is sometimes considered hermetic.*
>
> –Richard P. Gabriel, **Writers' Workshops & the Work of Making Things**

The poet, Richard Hugo, says it this way:

> *A poem can be said to have two subjects, the initiating or triggering subject, which starts the poem or "causes" the poem to be written, and the real or generated subject, which the poem comes to say or mean, and which is generated or discovered in the poem during the writing. That's not quite right because it suggests that the poet recognizes the real subject. The poet may not be aware of what the real subject is but only [has] some instinctive feeling that the poem is done.*
>
> –Richard Hugo, **The Triggering Town**

But just as importantly, the completion of an act of creation consists of a response to what has already been created along the way. The work itself supplies further triggers, and seeing it manifested in the real world, one can see whether one's imagination before construction was sufficient. Hugo, speaking of poetry, says it this way:

> *The poet's relation to the triggering subject should never be as strong as (must be weaker than) his relation to his words. The words should not serve the subject. The subject should serve the words.*
>
> –Richard Hugo, **The Triggering Town**

But a computer program is not like a poem or sculpture: A computer program must run properly and with adequate response for its needs. When building something that's never been built before, the only way to get it right is to get it right every step of the way, by repairing anything that is new and broken as soon as it's written and by making sure that nothing that is already ok becomes broken. In writing, this means proofreading each sentence as soon as it's complete, if not sooner. When a paragraph is done, proofread it. If possible, engage other eyes and minds to read what has been written already, and if it makes sense, use a writers' workshop to test the final product.

And in a situation where the users of a software system are not its developers, the system needs to be "proofread" every step of the way by those users. Therefore, not only should the software be subject to continual early testing of small additions, those additions should be given to the users.

By making small releases, the creators are also able to gather more and, as well, more accurate triggers, thereby enabling them to get closer to a good piece than if they were to rely on fewer triggers or triggers gathered over a short, initial period. And the course corrections done by small releases enables more eyes and minds to supply triggers—those of the users.

Agile methodologies don't usually describe what they are for this way, and different methodologies address these points a little differently, but if you look at those methodologies, they are trying to gather more triggers through spreading out the gathering period and through engaging crowds or mobs, on one hand, and to be

careful to build correctly and simply all through the process on the other. The magic comes from also being able to factor in a response to what has been already created—not only is this further source of triggers but the work itself to teach its creators about itself.

But what is missing from the whole process is what happens next. The agile methodologies are able to create very good first drafts, ones that might also be good shippable drafts. In the written arts, a first draft is followed by numerous revisions, workshops, private readings and discussions, copyediting, etc. This is part of completing a work of art. The agile methodologies are sometimes silent on this part of the job.

Nevertheless, I believe this way of looking at the creation of software—as gathering triggers, responding to existing work, and building correctly all along the way—will illuminate our understanding of agile methods and how to apply and evolve them.

Participatory Design (http://hci.stanford.edu/bds/14-p-partic.html)

> *The field of participatory design grew out of work beginning in the early 1970s in Norway, when computer professionals worked with members of the Iron and Metalworkers Union to enable the workers to have more influence on the design and introduction of computer systems into the workplace. Kristen Nygaard—who was well known for his computer-science research as codeveloper of SIMULA, the first object-oriented language—collaborated with union leaders and members, to create a national codetermination agreement, which specified the rights of unions to participate in the design and deployment decisions around new workplace technology.*
>
> *In the following decades, several projects in Scandinavia set out to find the most effective ways for computer-system designers to collaborate with worker organizations to develop systems that most effectively promoted the quality of work life. The DEMOS project, conducted in Sweden in the second half of the 1970s, involved an interdisciplinary team of researchers from the fields of computer science, sociology, economics, and engineering. Sponsored by the Swedish Trade Union Federation, its focus was "trade unions, industrial democracy, and computers" (Ehn, 1992, p. 107). Researchers worked with union members at a locomotive repair shop, a daily newspaper, a metalworking plant, and a department store.*
>
> *In the locomotive repair shop, DEMOS participants were brought in because union members were unhappy with a computer-based planning system being introduced by management. Originally, the call for assistance was motivated by controversy over the amount of time assigned to different work tasks; after working together, however, union members and researchers saw that the overall assumptions of the system (that work could be deskilled, and that all planning was a management prerogative) formed the chief issue. As a result, the union conducted its own investigation into production planning, and called attention to significant problems with materials organization, job design, and overall planning that were hindering production efficiency. Insight into the production process and its relationship to computer-system design and job design led the union to formulate a series of principles and positions that it could then use as a basis for bargaining with management (Ehn, 1992).*
>
> *The UTOPIA project was a collaboration between Swedish and Danish researchers and the Nordic Graphic Workers' Union. It developed and applied a work-oriented approach to the design of computer-based tools for skilled workers. The project team explicitly sought to reinforce and enhance skilled work-*

ers' control over process and methods, focusing on computer assistance for page makeup and image processing for newspapers.

Pelle Ehn, a primary participant in the UTOPIA project, describes its design philosophy, which they called the tool perspective:

The tool perspective was deeply influenced by the way the design of tools takes place within traditional crafts... new computer-based tools should be designed as an extension of the traditional practical understanding of tools and materials used within a given craft of profession. Design must therefore be carried out by the common efforts of skilled, experienced users and design professionals. Users possess the needed practical understanding but lack insight into new technical possibilities. The designer must understand the specific labor process that uses a tool. (Ehn, 1992, p. 112)

Good systems cannot be built by design experts who proceed with only limited input from users. Even when designers and prospective users have unlimited time for conversation, there are many aspects of a work process—such as how a particular tool is held, or what it is for something to "look right"—that reside in the complex, often tacit, domain of context. The UTOPIA researchers needed to invent new methods for achieving mutual understanding, so that they could more fully understand the work world of graphics workers.

> Requirement specifications and systems descriptions based on information from interviews were not very successful. Improvements came when we made joint visits to interesting plants, trade shows, and vendors and had discussions with other users; when we dedicated considerably more time to learning from each other, designers from graphics workers and graphics workers from designers; when we started to use design-by-doing methods and descriptions such as mockups and work organization games; and when we started to understand and use traditional tools as a design ideal for computer-based tools.
>
> –Ehn, 1992, p. 117

The UTOPIA project applied innovative design techniques, such as the use of role-playing scenarios with low-fidelity mockups to give the workers a feel for what their work might be like with new technology. In the end, UTOPIA produced a working system, called TIPS, that was tested at several newspapers, and was eventually sold to a company that developed image-processing systems.

There has been some participatory design in the United States in the Scandinavian style (see, e.g., Sachs, 1995), and widespread use of design techniques that are based on participatory design. Greenbaum and Kyng (1991, p. 4) identify four issues for design:

- *The need for designers to take work practice seriously—to see the current ways that work is done as an evolved solution to a complex work situation that the designer only partially understands*
- *The fact that we are dealing with human actors, rather than cut-and-dried human factors—systems need to deal with users' concerns, treating them as people, rather than as performers of functions in a defined work role.*
- *The idea that work tasks must be seen within their context and are therefore situated actions, whose meaning and effectiveness cannot be evaluated in isolation from the context*

- *The recognition that work is fundamentally social, involving extensive cooperation and communication*

These principles apply in all workplaces, regardless of the specific interactions between workers and management. They are at the root of design approaches that have been developed with names such as contextual inquiry (Holtzblatt, 1993), situated activity (Suchman, 1987), work-oriented design (Ehn, 1988), design for learnability (Brown and Duguid, 1992) situated design (Greenbaum and Kyng, 1991). An ongoing series of conferences on participatory design, organized by Computer Professionals for Social Responsibility (see Schuler and Namioka, 1993), has provided an opportunity for participatory-design concepts and practices to move beyond their original settings to a larger community of software designers.

Today, some of the concepts of participatory design are becoming standard practice in the computing industry. The emerging common wisdom in the major software-development companies is that it is important to design with the user, rather than to design for the user (as highlighted in De Young's account in Chapter 13). Participatory-design researchers have devised a variety of techniques to facilitate the communication of new technology possibilities to workers—to give the ultimate users insight into what it would be like to work with an envisioned system. These techniques include the low-fidelity mockups and role-playing activities of UTOPIA, as well as technology-aided methods such as the use of quick-and-dirty video animation to simulate the patterns of interaction with a new interface (see Muller et al., 1993; Muller, 1993).

In a panel at the 1994 Participatory Design Conference, Tom Erickson of Apple Computer set out four dimensions along which participation by users could be measured:

- *Directness of interaction with the designers*
- *Length of involvement in the design process*
- *Scope of participation in the overall system being designed*
- *Degree of control over the design decisions*

The original participatory-design movement was at the high end of all these scales. Designers worked over the full development cycle with a highly involved group of worker representatives. These representatives considered every aspect of the computer system being developed and of the deployment planned for it. They were in a setting where their labor–management agreements guaranteed that they had significant control over the outcome. As Kuhn describes in Chapter 14, there was a focus on issues of industrial democracy.

In many software-design settings, the degree of participation along these dimensions may not be uniformly high. The overall principles of participatory design, however, are relevant: The conceptual approach and its repertoire of techniques are applicable across a wide range of products and design settings.

Suggested Readings

Susanne Bodker. Through the Interface: A Human Activity Approach to User Interface Design. Hillsdale, NJ: Erlbaum, 1991.

Joan Greenbaum and Morten Kyng. Design at Work. Hillsdale, NJ: Erlbaum, 1991.

Michael Muller and Sarah Kuhn (eds). Special Issue on Participatory Design, CACM 36:4 (June, 1993).

Douglas Schuler and Aki Namioka, (eds). Participatory Design: Principles and Practices. Hillsdale, NJ: Lawrence Erlbaum Associates, 1993.

Failure to Tell the Truth About Design

OO focuses on perfecting each object instead of looking outward to the interaction of objects, creating whole, reliable, and flexible systems, and working with people trying to accomplish something in the real world.

–Ron Goldman

The most grievous fault of OO has been the inward focus it has engendered—a focus on making each object perfect, efficient, mathematically provable, etc. —as opposed to looking outward to the human world. The basic "object's attitude" tries to encapsulate & isolate every problem into the static world, sucking all the messy, human, living juice out of it.

–Ron Goldman

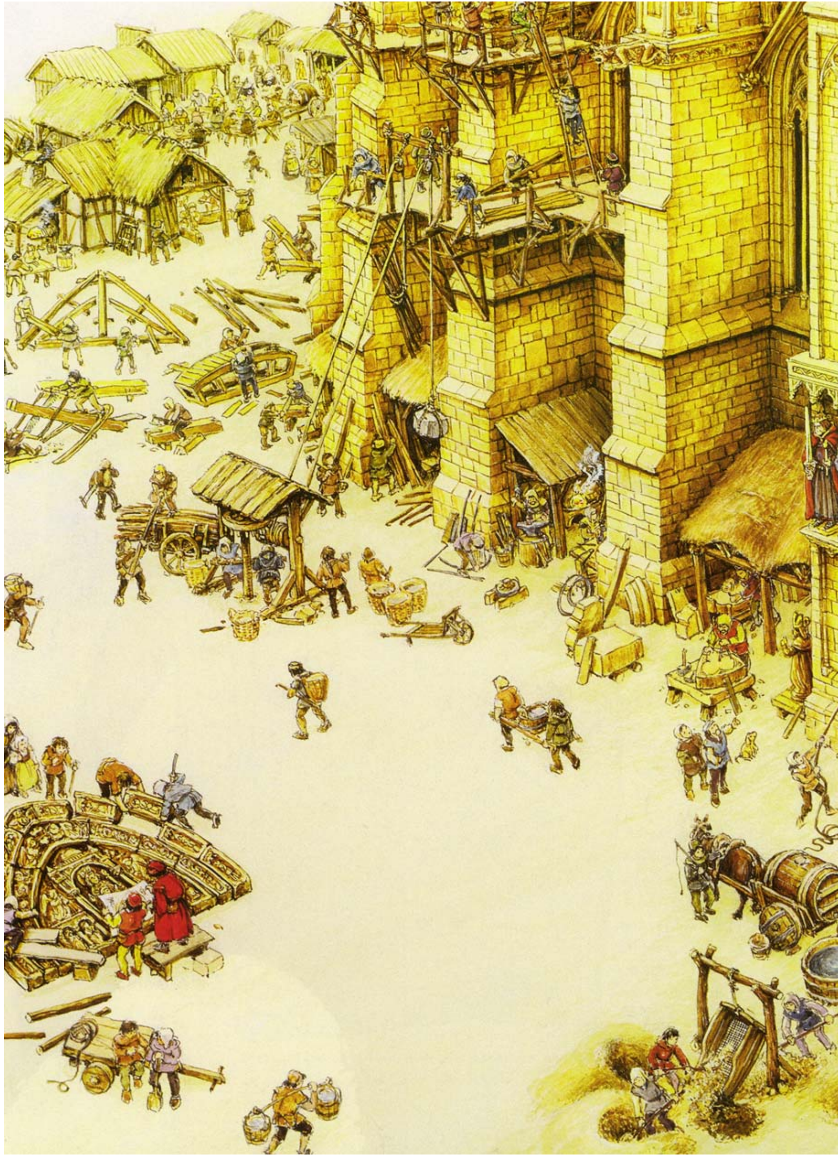
A first principle of construction: on no account allow the engineering to dictate the building's form . . . . never modify the social spaces to conform to the engineering structure of the building.

–Christopher Alexander

Failure to Tell the Truth About Design

Failure to Tell the Truth About Design

Failure to Tell the Truth About Design



Babbage on Design

> *It can never be too strongly impressed upon the minds of those who are devising new machines, that to make the most perfect drawings of every part tends essentially both to the success of the trial, and to economy in arriving at the result.*
>
> –**On Contriving Machinery**, Charles Babbage

However, for more complex machinery where performance will depend heavily upon "physical or chemical properties" (p. 261), optimum design cannot be determined on paper alone, and testing and experimentation ("direct trial") will be unavoidable.

Babbage worked during the heyday of the Industrial Revolution, and many of his ideas were informed by the transition from the medieval engineering worldview to the worldview of the modern factory. Babbage considered himself in the line of early economists like Adam Smith. For the medieval engineer, the purpose of engineering was to bring about God's perfect order on earth, and thereby replace the chaos of the physical and social world with something akin to the crystalline rationality of heaven. In this view, a designer has perfect knowledge of the task that the machine will perform and the environment in which the machine will operate. The designer of a machine has the power to implement the design and to secure the cooperation of all of the parties who will interact with it. And a factory is a self-sufficient world, wholly apart from the rest of the world except for the flows of material inputs and outputs, which can be characterized simply and completely.

–Adapted from Phil Agre, http://commons.somewhere.com/rre/2001/RRE.The.Fall.of.Babbage..html

Failure to Limit the Grand Narrative, Resulting in the Dot-Com Meltdown

> *Apparently we are trained to expect a software crisis, and to ascribe to software failures all the ills of society: the collapse of the dot-com bubble [27, 30], the bankruptcy of Enron [49], and the millennial end of the world [76].*
>
> *This corrosive scepticism about the achievements of programming is unfounded. Few doom-laden prophesies have come to pass: the world did not end with fireworks over the Sydney harbour bridge, and few modern disasters are due to software. To consider just two examples: the space shuttle crash was not caused by software—indeed, Feynman praises the shuttle software practices as exemplary engineering [23]; and the Dot-Com Boom (like the South Sea Bubble) was not caused by failure of technology, but the over-enthusiasm of global stock markets.*
>
> –Notes on Postmodern Programming, James Noble & Robert Biddle

In fact, the failure of the dot-com bubble was the fact that a bubble was created by the grand narratives of the OO world. When some of the promises of that grand narrative appeared on the Web in the form of home-grown (aka, postmodern) activities and some stores that seemed to make the experience of purchasing devoid of the expense of traveling to a store and walking around. If the energy required to shop could be eliminated, the narrative asserted, then shopping would become uncontrolled with uncontrolled profits. The beauty of some parts of the computer experience was confused with what could be done, and the technologists, faced with the possibility of unfathomable riches and with the belief that their god-like powers could rise to the occasion, decided the truth was not worth cracking open. That is, the grand narrative was left to stand and the global stock markets responded to it.

The truth of the software associated with the Web is indeed a success story, just as the stories of countless men and women who live in moderate comfort without grand ambitions nor with even noticeable achievements are success stories—simply because they play out without devastation and ambition.

- eCommerce requires adaptable software to handle changing business conditions and models
- From January 2000–May 2001:
  - 374 companies were delisted from NASDAQ
  - on average $5–$10m was spent on computer infrastructure
  - of that, $1–$5m was spent on software development
  - in many cases, the software was not suitable and not adaptable enough for real business situations

ZoZa.com is typical: the first proprietary apparel brand launched online selling high-fashion sportswear—hop off your mountain bike, pop into the Porsche, and off to the Pops.

- ATG Dynamo running on Solaris—eBusiness platform
- Oracle 8i
- Verity search tools
- A lot of custom, stand-alone Java glued everything together
- The bulk of the ATG work was outsourced to Xuma—an application infrastructure provider
- The production site:

- 2 Sun Netras doing web services via Apache/Stronghold (web server/secure web server)
- 4 Sun Netras providing an application layer and running Dynamo
- 1 Verity server
- 1 Sun Netra providing gateway services to fulfillment partners
- 1 Sun Enterprise 250 running Oracle as a production database
- Staging: 2 web boxes, 2 logic boxes, 1 Oracle box, 1 Verity box
- Development: one big Sun Ultra 2

The CTO of ZoZa says:

> *We built a good e-commerce platform, but unfortunately sales were slowly building just as the dot com economy collapsed. We had built a company to handle the promised phenomenal sales based on the Ziegler's self-promoted public profile. That never happened. The* ***costs of building our sales and fulfillment capabilities****, combined with ZoZa's lack of credit in the apparel manufacturing world caused us to go through SoftBank's $17 million quite quickly.*
>
> *Sizing was a terrible problem for ZoZa. Not only did the clothing get designed for ever smaller people, but even then the sizing was highly variable. Sometimes only a specific color of a product would be whacked out.*
>
> *We ended up building a separate database table for sizing anomalies.*
>
> [The Zieglers were] *disappointed by the Web. Initially, they planned to use virtual-reality technology so customers could mix-and-match items and feel like they were trying on clothes. They also wanted to provide a personal assistant to shoppers who could recommend items based on an individual's coloring. But the Zieglers scrapped all that when they found that* ***the technology ruined the shopping experience*** *because it took too long to download. "****The medium is far more rigid than we imagined****," says Mel* [Ziegler].
>
> *Patricia* [Ziegler], *a former newspaper illustrator who designs many of the clothes, was* ***put off by the poor quality of colors on the Web****. And* ***she was really bummed to learn how complex it was to swap out items that weren't selling well****. "We have to change 27 to 32 different links to swap out just one style—from the fabric to sizing to color," she says.*
>
> *So the Zieglers have gone back to basics.* ***Although it cost roughly $7 million to build, their Web site is, well, stark****—a picture of Zen minimalism. Navigation is simple and uncluttered. Nothing exists on the site that can't be optimally used with a standard 56k modem. The one concession to flash comes in the form of so-called mind crackers, which are Zen sayings about life hidden behind little snowflakes that have been sprinkled throughout the site.*

Failure to Give Form a Chance Over Structure

Design patterns in OO exist because the only way to do any sort of abstraction in OO languages is to define communicating and cooperating objects. In fact, one cannot even use ordinary functional or procedural abstraction in the usual way in some languages because everything that is like a procedure or a function must be associated with a class. This causes the designer to think exclusively in terms of a structure of objects.

Some problems require creating a language to express concepts, actions, constraints, etc. If you look at AI in the 1970s and 1980s, a lot of it was defining a language in which to express the programmatic solution and then expressing it. You can't do that with OO languages (except for some).

⁂

If you give someone Fortran, he has Fortran. If you give someone Lisp, he has any language he pleases.

–Guy L. Steele Jr

This is the nub of what I want to say. A language design can no longer be a thing. It must be a pattern—a pattern for growth—a pattern for growing the pattern for designing the patterns that programmers can use for their real work and their main goal.

My point is that a good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but by building on the frame of a base language.

–Guy L. Steele Jr

Modeling: *One of the great things to be driven by OOP is the idea of software modeling languages, like UML. People have noticed that a lot of the information in these models is structural, concerning the large-scale structure of information, presentation, and workflow. Within this structure is a more complex layer of information about behavior, detailed operational specifications, etc. But just setting up the structure involves huge amounts of typing in OOP. For example, a typical enterprise application might have 500 tables and 10,000 attributes. That translates to something like 200,000 lines of code before you have even done any real work. A similar thing happens with the user interface. People are working on round-trip template generation to produce all this code automatically, but it begs the question of why it needs to be generated at all? While some people dream of automatically generating full applications from specifications, this is not likely. But what will happen is that the structural aspects of the specifications will be compiled automatically, leaving the true complex behaviors to be coded and plugged into this structure. How will these plug-ins be coded? Probably using a variety of paradigms: functional, logic-based, and object-oriented.*

–William Cook

Failure to be Truthful about Learning

There are 24 books with titles like "Teach yourself <something related to> Java" or "Learn Java." 15 of them have phrases like "in 21 days," "in 24 hours," or "over the weekend." There there the ones that say "In Web time," "in the least amount of time," "the quickest way," and "now."

From Peter Norvig:

> *Researchers (Hayes, Bloom) have shown it takes about ten years to develop expertise in any of a wide variety of areas, including chess playing, music composition, painting, piano playing, swimming, tennis, and research in neuropsychology and topology. There appear to be no real shortcuts: even Mozart, who was a musical prodigy at age 4, took 13 more years before he began to produce world-class music. In another genre, the Beatles seemed to burst onto the scene, appearing on the Ed Sullivan show in 1964. But they had been playing since 1957, and while they had mass appeal early on, their first great critical success, Sgt. Peppers, was released in 1967. Samuel Johnson thought it took longer than ten years: "Excellence in any department can be attained only by the labor of a lifetime; it is not to be purchased at a lesser price." And Chaucer complained "the lyf so short, the craft so long to lerne."*
>
> *Here's my recipe for programming success:*
>
> - *Get interested in programming, and do some because it is fun. Make sure that it keeps being enough fun so that you will be willing to put in ten years.*
> - *Talk to other programmers; read other programs. This is more important than any book or training course.*
> - *Program. The best kind of learning is learning by doing. To put it more technically, "the maximal level of performance for individuals in a given domain is not attained automatically as a function of extended experience, but the level of performance can be increased even by highly experienced individuals as a result of deliberate efforts to improve." (p. 366) and "the most effective learning requires a well-defined task with an appropriate difficulty level for the particular individual, informative feedback, and opportunities for repetition and corrections of errors." (p. 20-21) The book Cognition in Practice: Mind, Mathematics, and Culture in Everyday Life is an interesting reference for this viewpoint.*
> - *If you want, put in four years at a college (or more at a graduate school). This will give you access to some jobs that require credentials, and it will give you a deeper understanding of the field, but if you don't enjoy school, you can (with some dedication) get similar experience on the job. In any case, book learning alone won't be enough. "Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter" says Eric Raymond, author of The New Hacker's Dictionary. One of the best programmers I ever hired had only a High School degree; he's produced a lot of great software, has his own news group, and through stock options is no doubt much richer than I'll ever be.*
> - *Work on projects with other programmers. Be the best programmer on some projects; be the worst on some others. When you're the best, you get to test your abilities to lead a project, and to inspire others with your vision. When you're the worst, you learn what the masters do, and you learn what they don't like to do (because they make you do it for them).*

- *Work on projects after other programmers. Be involved in understanding a program written by someone else. See what it takes to understand and fix it when the original programmers are not around. Think about how to design your programs to make it easier for those who will maintain it after you.*
- *Learn at least a half dozen programming languages. Include one language that supports class abstractions (like Java or C++), one that supports functional abstraction (like Lisp or ML), one that supports syntactic abstraction (like Lisp), one that supports declarative specifications (like Prolog or C++ templates), one that supports coroutines (like Icon or Scheme), and one that supports parallelism (like Sisal).*
- *Remember that there is a "computer" in "computer science". Know how long it takes your computer to execute an instruction, fetch a word from memory (with and without a cache miss), read consecutive words from disk, and seek to a new location on disk. (Answers here.)*
- *Get involved in a language standardization effort. It could be the ANSI C++ committee, or it could be deciding if your local coding style will have 2 or 4 space indentation levels. Either way, you learn about what other people like in a language, how deeply they feel so, and perhaps even a little about why they feel so.*
- *Have the good sense to get off the language standardization effort as quickly as possible.*

❧ ❧ ❧

On the other hand, because OO dominates everything, it makes it even harder for lesser skilled people to program.

> *OO has meant that the skill level required to program continues to rise, limiting who can be a programmer. Again this stems from the monolithic viewpoint of everything being an object. Complexity requires multiple levels of expression, which OO has neglected.*
>
> –Ron Goldman

❧ ❧ ❧

> *OO as it is with its focus on the small is incapable of conceiving of "Software as Literature"—the exclusive focus on the object makes it impossible to see the larger whole.*
>
> –Ron Goldman

Failure to Get Out of the Way

Redefining Computing

While it is perhaps natural and inevitable that languages like Fortran and its successors should have developed out of the concept of the von Neumann computer as they did, the fact that such languages have dominated our thinking for twenty years is unfortunate. It is unfortunate because their long-standing familiarity will make it hard for us to understand and adopt new programming styles which one day will offer far greater intellectual and computational power.

—John Backus, 1981

Programming Languages

Millions for compilers but hardly a penny for understanding human programming language use. Now, programming languages are obviously symmetrical, the computer on one side, the programmer on the other. In an appropriate science of computer languages, one would expect that half the effort would be on the computer side, understanding how to translate the languages into executable form, and half on the human side, understanding how to design languages that are easy or productive to use.... The human and computer parts of programming languages have developed in radical asymmetry.

—Alan Newell & Stu Card, 1985

Computing Paradigms

...the current paradigm is so thoroughly established that the only way to change is to start over again.

—Donald Norman, The Invisible Computer

Deep Trouble

Computer Science is in deep trouble. Structured design is a failure. Systems, as currently engineered, are brittle and fragile. They cannot be easily adapted to new situations. Small changes in requirements entail large changes in the structure and configuration. Small errors in the programs that prescribe the behavior of the system can lead to large errors in the desired behavior. Indeed, current computational systems are unreasonably dependent on the correctness of the implementation, and they cannot be easily modified to account for errors in the design, errors in the specifications, or the inevitable evolution of the requirements for which the design was commissioned. (Just imagine what happens if you cut a random wire in your computer!) This problem is structural. This is not a complexity problem. It will not be solved by some form of modularity. We need new ideas. We need a new set of engineering principles that can be applied to effectively build flexible, robust, evolvable, and efficient systems.

–Gerald Jay Sussman, MIT

Amorphous Computing Project, MIT

A colony of cells cooperates to form a multicellular organism under the direction of a genetic program shared by the members of the colony. A swarm of bees cooperates to construct a hive. Humans group

together to build towns, cities, and nations. These examples raise fundamental questions for the organization of computing systems:

- *How do we obtain coherent behavior from the cooperation of large numbers of unreliable parts that are interconnected in unknown, irregular, and time-varying ways?*
- *What are the methods for instructing myriads of programmable entities to cooperate to achieve particular goals?*

These questions have been recognized as fundamental for generations. Now is an opportune time to tackle the engineering of emergent order: to identify the engineering principles and languages that can be used to observe, control, organize, and exploit the behavior of programmable multitudes.

Amorphous Computing Project

The objective of this research is to create the system-architectural, algorithmic, and technological foundations for exploiting programmable materials. These are materials that incorporate vast numbers of programmable elements that react to each other and to their environment. Such materials can be fabricated economically, provided that the computing elements are amassed in bulk without arranging for precision interconnect and testing. In order to exploit programmable materials we must identify engineering principles for organizing and instructing myriad programmable entities to cooperate to achieve pre-established goals, even though the individual entities are unreliable and interconnected in unknown, irregular, and time-varying ways.

Autonomic Computing Project, IBM

Civilization advances by extending the number of important operations which we can perform without thinking about them.

–Alfred North Whitehead

[How to you make things simpler for administrators and users of IT?] . . . *we need to create more complex systems. How will this possibly help? By embedding the complexity in the system infrastructure itself—both hardware and software —then automating its management. For this approach we find inspiration in the massively complex systems of the human body. Think for a moment about one such system at work in our bodies, one so seamlessly embedded we barely notice it: the autonomic nervous system.*

It tells your heart how fast to beat, checks your blood's sugar and oxygen levels, and controls your pupils so the right amount of light reaches your eyes as you read these words. It monitors your temperature and adjusts your blood flow and skin functions to keep it at 98.6º F. It controls the digestion of your food and your reaction to stress—it can even make your hair stand on end if you're sufficiently frightened. It carries out these functions across a wide range of external conditions, always maintaining a steady internal state called homeostasis while readying your body for the task at hand.

Autonomic Computing Project

> *But most significantly, it does all this without any conscious recognition or effort on your part. This allows you to think about what you want to do, and not how you'll do it: you can make a mad dash for the train without having to calculate how much faster to breathe and pump your heart, or if you'll need that little dose of adrenaline to make it through the doors before they close.*
>
> *It's as if the autonomic nervous system says to you, Don't think about it—no need to. I've got it all covered. That's precisely how we need to build computing systems—an approach we propose as autonomic computing.*

Feyerabend Project

> *...one of the most striking features of recent discussions in the history and philosophy of science is the realization that events and developments ... occurred only because some thinkers either decided not to be bound by certain 'obvious' methodological rules, or because they unwittingly broke them.*
>
> *This liberal practice, I repeat, is not just a fact of the history of science. It is both reasonable and absolutely necessary for the growth of knowledge. More specifically, one can show the following: given any rule, however 'fundamental' or 'necessary' for science, there are always circumstances when it is advisable not only to ignore the rule, but to adopt its opposite.*
>
> —Paul Feyerabend, Against Method

Feyerabend Project

- Understand the limitations of our current computing paradigm
- Understand the limitations of our current development methodologies
- Bring users—that is, people—into the design process
- Make programming easier by making computers do more of the work
- Use deconstruction to uncover marginalized issues and concepts
- Looking to other metaphors
- Three workshops so far, four more planned—using a tipping-point approach

Feyerabend Project

- Homeostasis, immune systems, self-repair, and other biological framings
- Physical-world-like constraints—laws, contiguity
- Blackboards, Linda, and rule-systems—use compute-power
- Additive systems—functionality by accretion not by modification
- Non-linear system-definition entry—instead of linear text
- Non-mathematical programming languages
- Sharing customizations

- Language co-mingling and sustained interaction instead of one-shot procedure invocation in the form of questions/answers or commands
- Piecemeal growth, version skews, random failures
- Artists' understanding, ambiguous truth

Feyerabend: Biological Framings of Problems in Computing

- Goal is to come up with "Hilbert Problems" for computing
- Need for new metaphors both for computing and for biology

> *Every living organism is the outward physical manifestation of internally coded, inheritable, information.*
>
> –http://www.brooklyn.cuny.edu/bc/ahp/BioInfo/GP/Definition.html

Feyerabend Home Page:

http://www.dreamsongs.com/Feyerabend/Feyerabend.html

Lifetime Management

- An object should be able to participate in its growth and evolution

William Cook

> *Software Modeling is a new paradigm for creating software by modeling each aspect of a desired system using appropriate high-level modeling languages within an overall modeling architecture. Example aspects include user interfaces, data models, security, data mappings, transaction boundaries, queuing/distribution, exception handling, event models, algorithms, workflow, etc. The modeling languages may be declarative, equational, logic/relational, functional, rule-based, object-oriented, procedural, or simply structural - but must meet two criteria: 1) be an effective way to precisely describe an aspect of system behavior, and 2) fit together to form an overall system architecture. Although software models can be interpreted, compilation of the models allows for powerful optimizations that allow extremely configurable models to be resolved into efficient programs. The transformation can also adapt the system to different contexts, for example a PDA, GUI, or browser, and assist in generating documentation and test cases. To date, no system for Software Modeling exists, although many of the building blocks exist in active research areas, including domain-specific languages, partial evaluation, meta-programming, aspect-oriented programming, informal modeling notations, reusable frameworks, and pattern languages. Some open issues include modularity in modeling languages, debugging, and extensibility of architectures and modeling languages. If this vision of Software Modeling is achieved, it will introduce a new level of software reuse and program readability, by focusing on concise descriptions of what makes one program different from another (its fundamental models) while suppressing all the details that every program shares with other similar programs.*
>
> –William Cook

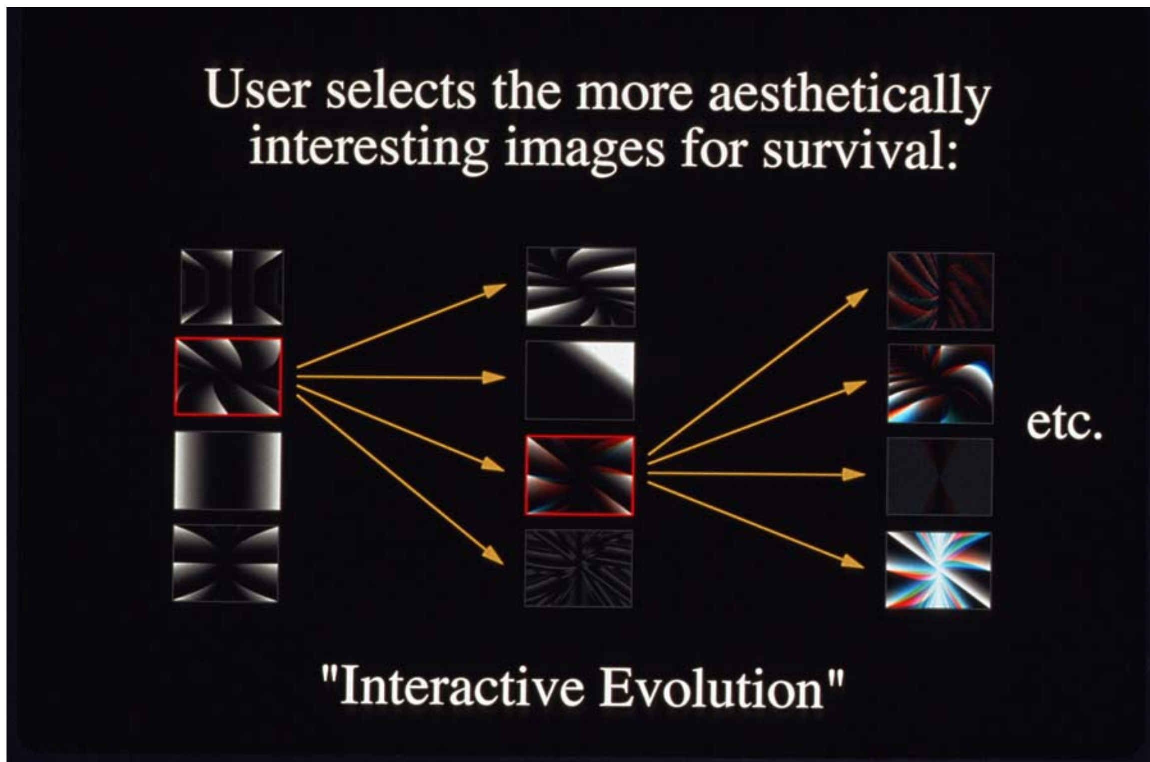
Failure to Embrace Other Paradigms

When OO became trendy, most of the other paradigms disappeared from serious consideration, though most of them continued in a more underground way. Diversity is the way we get innovation and invention, and OO cut that off. Whether we can blame the OO people for this is unclear. Nevertheless, the fact is that some important alternatives have been forced out of the picture, and we are stuck with a bastardized version of OO today.

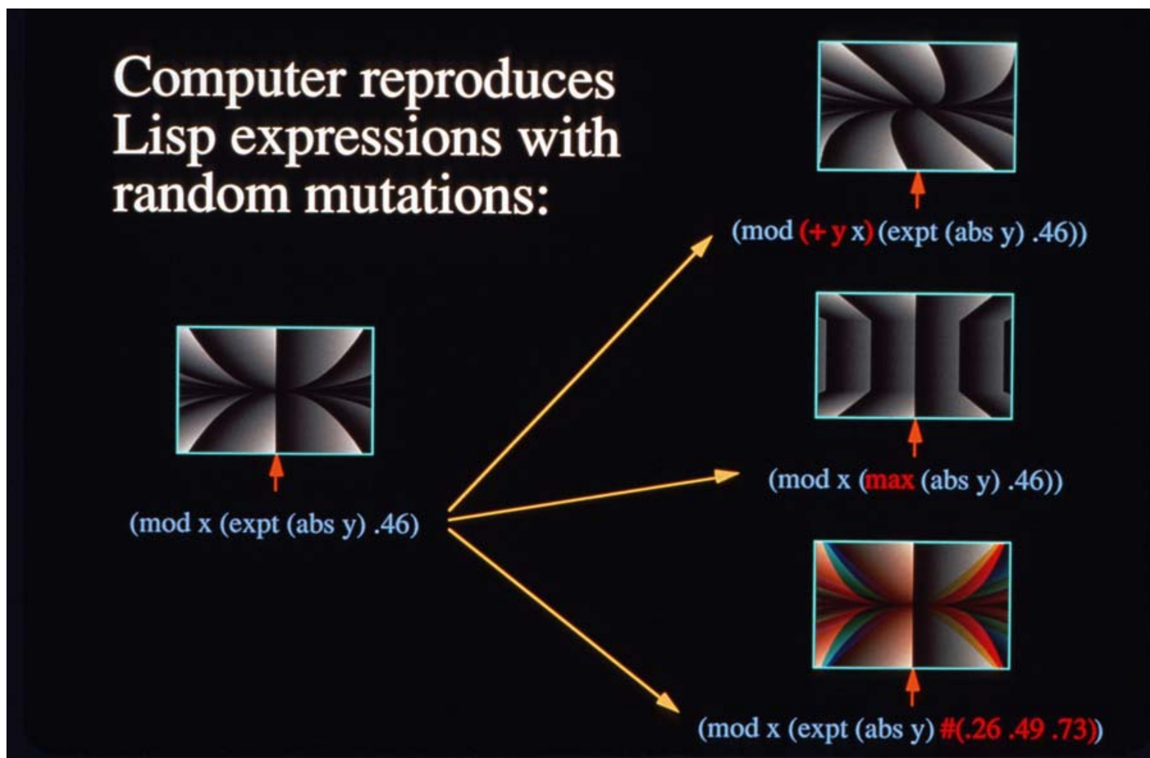
- Where is logic programming?
- Where are expert systems?
- Where is data-driven programming?
- Where is functional programming?
- The good has forced out the excellent

> ***Postmodernism:*** *the tendency toward totalizing discourse (can I use that phrase with an ironic tone?) has to stop. We need to give ourselves the freedom to use (or create) the most effective means of description for a given situation. We must find ways to let these different forms work together. Do not take a simplistic view of postmodernism and try to turn it into the ultimate paradigm. Don't fall into that trap...*
>
> –William Cook

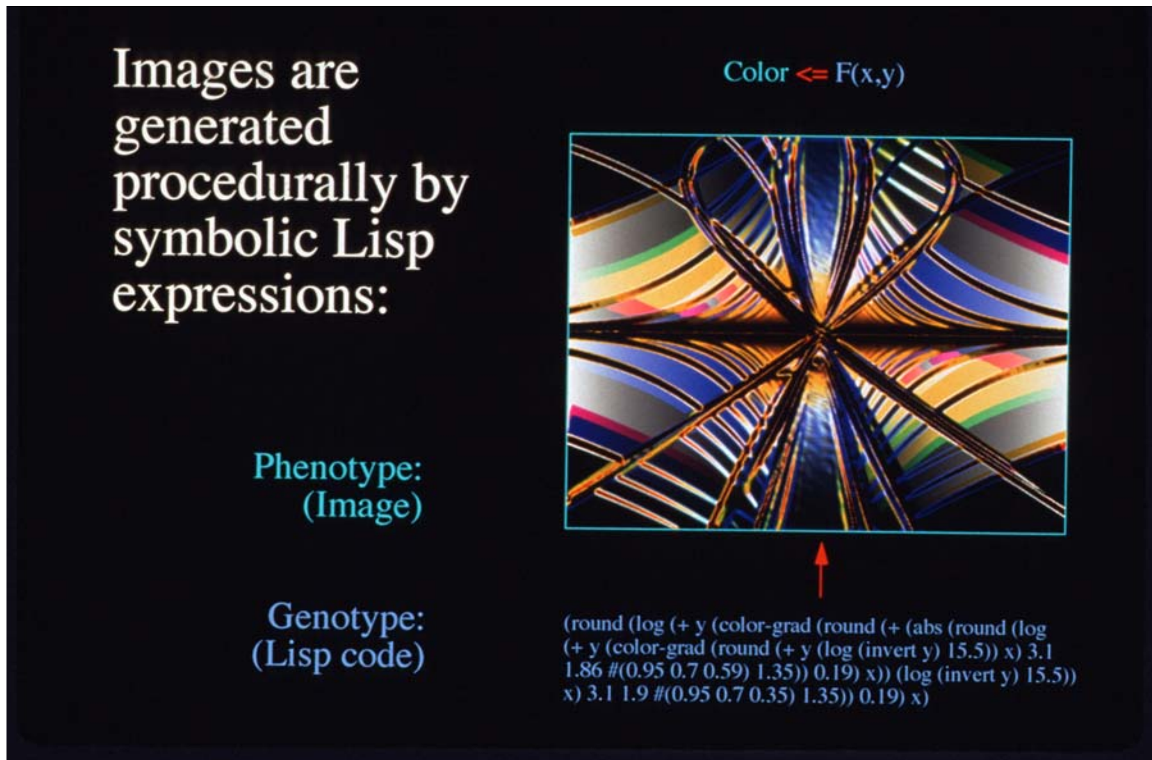
Other paradigms can bring lots of advantages:



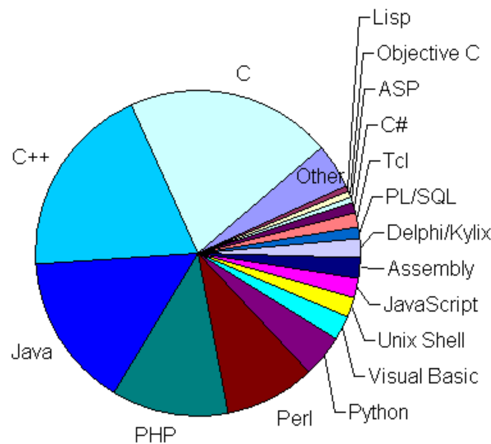
Here is a program easily written in Lisp to explore ideas of evolution but which would be much more difficult in C++, Java, or C#.



Notice that Lisp representations of expression along with Lisp's ability to compile and execute code in memory make this not only feasible but easy.



Here is the breakdown of programming languages in use on SourceForge:



Scheme-Based Web Server

(from Programming the Web with High-Level Programming Languages by Paul Graunke, Department of Computer Science, Rice University Shriram Krishnamurthi, Department of Computer Science, Brown University Steve Van Der Hoeven, ESSI, Université de Nice Matthias Felleisen, Department of Computer Science, Rice University)

A Web server provides operating system-style services. Like an operating system, a server runs programs (e.g., CGI scripts). Like an operating system, a server protects these programs from each other. And, like an operating system, a server manages resources (e.g., network connections) for the programs it runs.

Some existing Web servers rely on the underlying operating system to implement these services. Others fail to provide services due to shortcomings of the implementation languages. In this paper, we show that implementing a Web server in a suitably extended functional programming language is straightforward and satisfies three major properties. First, the server delivers static content at a performance level comparable to a conventional server. Second, the Web server delivers dynamic content at five times the rate of a conventional server. Considering the explosive growth of dynamically created Web pages [7], this performance improvement is important. Finally, our server provides programming mechanisms for the dynamic generation of Web content that are difficult to support in a conventional server architecture.

The basis of our experiment is MrEd [11], an extension of Scheme [15]. The implementation of the server heavily exploits four extensions: first-class modules, which help structure the server and represent server programs; preemptive threads; which are needed to execute server programs; custodians, which manage the resource consumption of server programs; and parameters, which control stateful attributes of threads. The server programs also rely on Scheme's capabilities for manipulating continuations as first-class values. The paper shows which role each construct plays in the construction of the server.

Failure to Tell the Truth

Languages and paradigms have been dismissed in the past because they are too slow, programming in them is too hard, and applications are too large. Lisp, for example, was dismissed because of this. Others are Prolog, ML, Smalltalk, and Haskell.

Java, for example, is slow and large, and some would argue it is hard to program in since it takes the simple concepts of OO and blends them with static and other inflexible ideas from other languages. But Java is completely accepted in the current OO and mainstream worlds.

To be consistent would be nice.

From "Lisp as an Alternative to Java" by Erann Gatt:

> *Two striking results are immediately obvious from the figures. First, development time for the Lisp programs was significantly lower than the development time for the C, C+, and Java programs. It was also significantly less variable. Development time for Lisp ranged from a low of 2 hours to a high of 8.5, compared to a range of 3 to 25 hours for C and C++ and 4 to 63 hours for Java. Programmer experience cannot account for the difference. The experience level was lower for Lisp programmers than for both the other groups (an average of 6.2 years for Lisp versus 9.6 for C and C++ and 7.7 for Java). The Lisp programs were also significantly shorter than the C, C++, and Java programs. The Lisp programs ranged from 51 to 182 lines of code. The mean was 119, the median was 134, and the standard deviation was 10. The C, C++, and Java programs ranged from 107 to 614 lines, with a median of 244 and a mean of 277.*
>
> *Second, although execution times of the fastest C and C++ programs were faster than the fastest Lisp programs, the runtime performance of the Lisp programs in the aggregate was substantially better than C and C++ (and vastly better than Java). The median runtime for Lisp was 30 seconds versus 54 for C and C++. The mean runtime was 41 seconds versus 165 for C and C++. Even more striking is the low variability in the results. The standard deviation of the Lisp runtimes was 11 seconds versus 77 for C and C++. Furthermore, much of the variation in the Lisp data was due to a single outlier at 212 seconds (which was produced by the programmer with the least Lisp experience: less than a year). If this outlier is ignored, the mean is 29.8 seconds, essentially identical to the median, and the standard deviation is only 2.6 seconds.*
>
> *Memory consumption for Lisp was significantly higher than for C and C++ and roughly comparable to Java. However, this result is somewhat misleading for two reasons. First, Lisp and Java both perform internal memory management using garbage collection, so often Lisp and Java runtimes will allocate memory from the operating system that is not actually being used by the application program. Second, the memory consumption of Lisp programs includes memory used by the Lisp development environment, compiler, and runtime libraries. This allocation can be substantially reduced by removing from the Lisp image features that are not used by the application, an optimization we did not perform.*

Failure to Work with Databases Well

Databases: *we still haven't gotten OOP to interact well with relational databases. The problem is that OOP is all about encapsulating state and behavior, while databases are all about separating state and behavior. Some people (usually programming language people) say that databases will go away. They will not. But I don't think the DB people are worried—if you don't look out, it is more likely that programming languages will go away, or at least be diminished in scope. The programming language notion of "persistence" is an anti-pattern. Until we make room for both the OOP and RDBMS paradigms, there is going to continue to be wasted effort (and bruised noses).*

–William Cook

There's something about DBs taking advantage of an economy of scale that PLs don't address. Most PLs have no facilities for managing large numbers of objects, efficient iteration over them, or migrating them to a secondary store for efficiency, let alone long-term persistence. I think the thing about DBs is that they've focused on this to the exclusion of the language features.

–Warren Harris

Failures from William Cook

Has OOP Failed?

> Dealing with failure is easy: Work hard to improve. Success is also easy to handle: You've solved the wrong problem. Work hard to improve.
>
> –Alan Perlis

Failure is always relative to a game, a set of rules, or a goal, mission, or objective. So before you can determine if OOP has failed, you have to decide what game it was trying to play. For most of us, there was only one game in town: to be the ultimate programming paradigm, the one that solves the problem of reuse, is the obvious best tool for any programming problem, something that we can commit to and be fulfilled by forever.

*The tendency toward this kind of **totalizing discourse** in computer science is strong. We believe our paradigms are incompatible (ever tried to mix procedural and logic programming?), or perhaps we don't spend enough effort on making them work together (how long did it take to get Lisp and C to talk?). In either case, the lack of ability to mix paradigms leads to "lock in": the one paradigm you pick has to do everything. And in general we believe that there is an ultimate paradigm.*

We also have to figure out what OOP is. Rather than take a theoretical viewpoint, let's just simply say that it is what you do with languages like Java and C# (or Simula). I don't include Smalltalk, or Beta, or other research languages because that is what they are (even though including them wouldn't change the results, it would make the issues more complex).

If this is the game, then I can say that OOP has clearly failed. OOP is not the ultimate paradigm. It hasn't actually solved the problem of reuse. It isn't the best solution for every kind of problem. It is a useful technique, probably about as important as Structured Programming was in the '70s (there were a few conferences on structured programming back then, but they stopped after a while).

So, here are problem areas:

- ***Macros:** the basic problem is that a component that is sufficiently parameterized to be re-usable will in practice be un-usable, either because its API is too huge, or it will be too slow, or both. The Java Swing API is a good example. The only possible solution is to allow some of the parameters (or configuration information) to be resolved at compile time. When connected at runtime, components require too much glue. Think of macros as allowing a programmer to write compile-time code, not just run-time code. Reflection is a fine idea, but being forced to do it at runtime is a bad idea. In this sense, Yacc is a macro language that we bolt onto the side of our OO languages. We also had Lisp Macros, CPP macros, and these were useful but messy. There is active research to clean them up and make them better. But modern OO languages have thrown them out as unclean, and so it will be a while before progress can be made.*
- ***Databases:** we still haven't gotten OOP to interact well with relational databases. The problem is that OOP is all about encapsulating state and behavior, while databases are all about separating state and behavior. Some people (usually programming language people) say that databases will go away. They will not. But I don't think the DB people are worried—if you don't look out, it is more likely that programming languages will go away, or at least be diminished in scope. The programming lan-*

guage notion of "persistence" is an anti-pattern. Until we make room for both the OOP and RDBMS paradigms, there is going to continue to be wasted effort (and bruised noses).

> There's something about DBs taking advantage of an economy of scale that PLs don't address. Most PLs have no facilities for managing large numbers of objects, efficient iteration over them, or migrating them to a secondary store for efficiency, let alone long-term persistence. I think the thing about DBs is that they've focused on this to the exclusion of the language features.
>
> –Warren Harris

- ***RPC:*** *the problem is that taking lots of fine-grained objects and trying to distribute them is a bad idea. Round-trips are never going to be fast enough. The solution is at hand: XML web services, where we send large messages, which area really large documents describing what remote operation needs to be performed, over the wire—one round trip. Oddly enough, this is very similar to the standard way that systems communicate with RDBMS: SQL is effectively a high-level message API to the database server.*
- ***Modeling:*** *One of the great things to be driven by OOP is the idea of software modeling languages, like UML. People have noticed that a lot of the information in these models is structural, concerning the large-scale structure of information, presentation, and workflow. Within this structure is a more complex layer of information about behavior, detailed operational specifications, etc. But just setting up the structure involves huge amounts of typing in OOP. For example, a typical enterprise application might have 500 tables and 10,000 attributes. That translates to something like 200,000 lines of code before you have even done any real work. A similar thing happens with the user interface. People are working on round-trip template generation to produce all this code automatically, but it begs the question of why it needs to be generated at all? While some people dream of automatically generating full applications from specifications, this is not likely. But what will happen is that the structural aspects of the specifications will be compiled automatically, leaving the true complex behaviors to be coded and plugged into this structure. How will these plug-ins be coded? Probably using a variety of paradigms: functional, logic-based, and object-oriented.*
- ***Encapsulation:*** *the problem is that encapsulation is fantastic in places where it is needed, but it is terrible when applied in places where it isn't needed. Since OOP enforced encapsulation no matter what, you are stuck. For example, there are many properties of objects that are non-local, for example, any kind of global consistency. What tends to happen in OOP is that every object has to encode its view of the global consistency condition, and do its part to help maintain the right global properties. This can be fun if you really need the encapsulation, to allow alternative implementations. But if you don't need it, you end up writing lots of very tricky code in multiple places that basically does the same thing. Everything seems encapsulated, but is in fact completely interdependent. This is related to the notion of aspects.*

> Encapsulation is the most miserable failure of all. It's utterly the wrong idea.
>
> –Warren Harris

- ***Postmodernism:*** *the tendency toward totalizing discourse (can I use that phrase with an ironic tone?) has to stop. We need to give ourselves the freedom to use (or create) the most effective means of description for a given situation. We must find ways to let these different forms work together. Do not take a simplistic view of postmodernism and try to turn it into the ultimate paradigm. Don't fall into that trap...*

Summary: OOP succeeded in solving the wrong problem, and we are busy trying to make the solution more and more pure and clean in the hope that it will eventually work.