Has
OOP Failed?

By Richard Mansfield
September 2005

Computer programming today is in serious difficulty. It is controlled by what amounts to a quasi-religious cult-- Object Oriented Programming (OOP). As a result, productivity is too often the last consideration when programmers are hired to help a business computerize its operations.

There's no evidence that the OOP approach is efficient for most programming jobs. Indeed I know of no serious study comparing traditional, procedure-oriented programming with OOP. But there's plenty of anecdotal evidence that OOP retards programming efforts. Guarantee confidentiality and programmers will usually tell you that OOP often just makes their job harder.

Excuses for OOP failures abound in the workplace: we are "still working on it"; "our databases haven't yet been reorganized to conform to OOP structural requirements"; "our best OOP guy left a couple of years ago"; "you can't just read about OOP in a book, you need to work with it for quite a while before you can wrap your mind around it"; and so on and so on. If you question the wisdom of OOP, the response is some version of "you just don't *get* it." But what they're really saying is: "You just don't *believe in it.*"

All too often a company hires OOP consultants to solve IT problems, but then that company's real problems begin. OOP gurus frequently insist on rewriting a company's existing software according to OOP principles. And once the OOP takeover starts, it can become difficult, sometimes impossible, to replace those OOP people. The company's programming and even its databases can become so distorted by OOP technology that switching to more efficient alternative programming approaches can be

Has OOP Failed?
By Richard Mansfield
A Four J's White Paper

Page 1 of 13
September 2005

costly but necessary. Bringing in a new group of OOP experts isn't a solution. They are likely to find it hard to understand what's going on in the code written by the earlier OOP team. OOP encapsulation（hiding code）and sloppy taxonomic naming practices result in lots of incomprehensible source code. Does anyone benefit from this confusion and inefficiency?

When the Java language was first designed, a choice had to be made. Should they mimic the complicated, counter-intuitive punctuation, diction, and syntax used in C++, or should they create an understandable, clear language--as much as possible like English? A "natural language."

They apparently decided to deliberately avoid straightforward, clear syntax because many professional programmers wouldn't take such a language seriously. Why? Job protection. Most professions have their jargon: One benefit is that it serves to separate the pros from outsiders. Academic theorists are the second primary reason that OOP is popular, and they are also major contributors to its complexity and inefficiency.

OOP originated in high-variable systems modeling（such as weather forecasting）, and it can be useful in some kinds of programming. One notable OOP success is the application of some of its principles to graphic user interface（event-driven component）programming. However, university professors have extended OOP principles to *all* programming, not merely those specialized areas where aspects of OOP theory work well and make sense. Professors, bless them, all too often prefer to ignore facts that conflict with their favorite theories.

I'm not saying we don't need professors. Society often benefits from--and some kinds of progress depend on--people who flit around testing every new intellectual craze, taking ideas to extremes. Indeed, Peter Pan would be much the poorer without Tinkerbelle. But if I'm faced with a mission-critical project, I wouldn't seek advice from a pixie.

Although the effectiveness of Object-Oriented Programming is debatable, curiously there is almost no debate about it. It pervades contemporary computer programming. Each new generation of programmers marches out of the universities as committed OOP practitioners. Most are simply unaware of any alternative.

If you raise questions about OOP's effectiveness, a hue and cry goes up from the multitude that have invested many years and much money developing their expertise in the art of OOP.

I say *art* because --whatever else it might be--programming isn't science. Although subsumed into computer science departments at many universities, programming is fundamentally linguistic, an act of communication. Alternatively, computer programming is taught in the math department in some schools, which makes perhaps even less sense. Programming, like cooking, involves math only occasionally and tangentially. But this confusion about the purpose and character of computer programming is endemic in the academy, and spills out into the profession of programming in the world at large. To me, the greatest misunderstanding involves the highly dubious utility of OOP and the unnatural family of languages derived from C, such as Java and C++ that are associated with OOP.

Like many theories, OOP includes some attractive concepts. OOP's major principles--encapsulation, inheritance, and polymorphism--are of value in specialized programming contexts. However, these concepts often prove difficult to apply to *most* real-world programming tasks. Yet most programming shops today are relentless in their cult-like insistence on employing C/OOP for every job they tackle.

Here's one common example of how OOP is used inappropriately: Database systems embedded within successful companies often must be highly adaptable-- responsive to the ever-changing needs of a dynamic, effective corporate culture. OOP database management by contrast often results in relatively inflexible schema, and future changes in the business model must be translated (mapped) back to the now-obsolete OOP structures. One feature of OOP encapsulation theory urges that data be embedded within objects rather than remaining in a distinctly separate system (as in a traditional database). This contributes to the rigidity of OOP data management, and makes it more difficult to respond rapidly and efficiently to changes in the corporation's, or its customers', behaviors and needs.

OOP usually isn't the best solution that many claim it is. Of course in the current workplace programmers and developers cannot freely raise questions about OOP's efficacy. OOP is the dominant theory. Questioning it can imperil one's job. What's more, few alternatives to C/OOP are taught in the schools anymore.

OOP has taken over computer "science" departments to the exclusion of any alternative programming system. A generation of programmers has graduated from schools that teach only the OOP way, so that's what they tend to prefer. It's really all they know. They memorize a set of "best practices" (abstract design rules) to help them grind out OOP without making too many mistakes. OOP can be effective to some extent during the initial design phase--when a program's broad goals and large-scale structure is being imagined and sketched in. However, for day-do-day code writing, OOP is generally the single biggest obstacle to contemporary programmer productivity. Are results achieved faster? No, creating programs usually takes considerably longer than it does with more natural languages and more sensible approaches to modular design. Have bugs been reduced? No, they usually increase. Is program maintenance easier? No, OOP and C languages are notoriously counter-intuitive and difficult to read.

Given the freedom to choose, the public--amateurs and small business programmers--greatly prefers Basic and 4GLs and all but ignores OOP. According to several studies of overall computer language use, Basic and 4GL programmers have greatly outnumbered C programmers since the 1980's. But the elite, professional programmer class learns C-language variants and OOP in their computer "science" classes, and then stick with OOP/C for the rest of their lives. It reminds me a bit of the priest class in ancient Egypt. They accepted a bowl of olives from an ordinary guy who wanted to send a prayer to Osiris, then the priest went into the temple to talk to the gods in a secret language that only priests, and gods, understood. It's a living, as they say.

OOP's Benefits Aren't Unique

OOP is employed in programming in two primary, somewhat unrelated, ways. First, it's used to organize code libraries, such as Microsoft's .NET framework, much the way a biologist classifies fauna. Second, OOP seeks to bring order to large-scale programming projects--to help create a sensible overall design and to keep programmers from stepping on each other's toes.

OOP does have some useful features for such group-programming jobs. Encapsulation seals off tested code from prying eyes and from modification that might introduce bugs. But hiding code isn't unique to OOP. Most languages permit

compilation into inviolate code libraries, black-box functions, and other ways to lock code.

Some even argue that OOP is best even for programmers working solo. This claim often rests on OOP's code reusability features--the idea that with OOP you can easily reuse objects you've previously written for other programs, or more easily update programs if they need modification later on.

This, too, is a largely bogus claim. From the simplest copying and pasting on the low end, all the way up to classic code libraries--every computer language offers code reuse and code maintenance features in some form or another. And the considerable effort required to superimpose OOP on more natural programming languages and techniques often requires far more programmer time, than any time saved during future code maintenance. What's more, it seems obvious that when you go back to modify existing code, the easier that code is to read and understand, the more quickly you can edit it. OOP and C are not known for readability. Quite the opposite.

Runaway Linguistic Inflation

Linguistic inflation is a major, unpleasant side-effect of OOP/C and the drive toward elite jargon for professionals. Partly as a result of attempts to incorporate the idea of "object" into the programmer's vocabulary, programs and code libraries have gotten very messy and confusing. However, inured as they are in the One True Theory, few professional programmers dare complain about the highly redundant, needlessly complex code, and the breakdown of logical categories in code libraries that is a direct result of the application of OOP/C principles to otherwise straightforward computing tasks.

OOP inflates and damages programming languages in several ways:

- Key terms such as *object, method* and *property* have lost their descriptive value almost entirely because they've become largely indistinguishable in meaning. And the term *object* itself is applied to almost every element in OOP code libraries.

- The vocabulary that defines OOP features has become highly redundant.

Has OOP Failed?
By Richard Mansfield
A Four J's White Paper

Page 5 of 13
September 2005

- Programming tasks such as printing a document that used to require only three or four words in the source code, can now require hundreds of words.

- The diction in OOP languages grows enormous: The Basic language in 1990 consisted of fewer than 400 terms, and of these only 50 were usually employed for most common programming tasks; Basic .NET has many thousands of terms.

- OOP code libraries become huge and difficult to use: Programmers now often find themselves spending less time writing actual source code than they spend trying to find, and correctly address, classes in the .NET framework.

Let me elaborate on these points. First, the OOP classification system used to organize the elements of the programming language breaks down quickly into essentially meaningless, sometimes conflicting, categories.

OOP superimposes complex taxonomies on top of classic procedures and natural language technologies. And even the primary OOP categories--such as objects and their members--are frequently illusory on close inspection. With OOP the "distinction" between object, property, and method is always up for debate. The categories are quite vague and often interchangeable.

Consider that OOP's most fundamental grammatical relationship is between an object (similar to a noun in human language grammar) and its methods (verbs) and properties (adjectives). This grammar, though, is merely theoretical. In practice, the categorizations usually collapse in real-world source code. For example, a single OOP property can simultaneously be an object and a method in the same "sentence" (logical line of code). Put another way: an adjective is a noun is a verb. You can memorize each specific variation of this kind of nonsense, but you cannot diagram it or learn rules from it for future programming jobs.

What's more, in Microsoft's OOP .NET suite of languages, *everything* is an object, even fundamental components such as the integer variable type and logical operators. When everything is something, you have the fish-in-water problem: the concept of water has no meaning to a fish because it's wholly pervasive in their world. The term *object* is so omnipresent in OOP libraries that it, too, has lost much of its meaning.

With OOP-inflected programming languages, computer software becomes more verbose, less readable, less descriptive, and harder to modify and maintain. OOP apologists claim the opposite: that OOP is more efficient than traditional programming approaches. However, I've never seen any serious, reliable study comparing OOP with procedural programming. Such a study is long overdue.

Dozens of terms where one will do

Another unpleasant side effect of OOP is its elite patois--a highly redundant jargon only professionals understand. OOP computer language features now have multiple names, names that often have no distinctions in meaning--not even subtle distinctions. For example, consider the many synonyms for what is probably best described simply as a *code library* (i.e., a collection of procedures): *Assembly, control library, control type library, class library, object library, namespace, project model, object model, host object model, proxylib, type library, plug-in, plugin type library, services, services library, development environment, core type library, extensibility, runtime library, runtime execution library, runtime execution engine, kernel, helper*, and *dynamic link library*.

Some will argue that there are distinctions here. Indeed, in some cases, adjectives such as *core, control*, or *plug-in* do shade the meaning a bit. But nearly all the terms in the previous paragraph are merely synonyms--without even the slightest distinction between them. And some of the terms are simply inept. One popular usage, "object library," makes no sense on closer examination. OOP asserts that "objects" only exist during runtime, during execution of a program. Objects must be instantiated, they cannot be collected in a library--only classes can exist in a library. The proper term is "class library," for the same reason that you don't confuse a cookbook with a restaurant.

Has OOP Failed?
By Richard Mansfield
A Four J's White Paper

Page 7 of 13
September 2005

Bloated source code

Perhaps the worst form of inflation resulting from OOP is the redundancy in the programming code now required to do even common, previously simple, tasks. Consider the difference between the non-OOP instructions to print a document, and the OOP version. Here's how you traditionally print some text in Visual Basic Version 6 (pre-OOP):

```
Printer.Print Text1
```

In the VB.NET OOP version, you must replace these three words with 80 *lines* of source code (277 words total). You must use a group of members, import several libraries, and muster a fair amount of information (such as the "brush" color, line height, running word count, and so on). You must supervise several other aspects of the printing process as well. For example, if you don't correctly specify and keep track of how several elements of the printing are carried out--every few pages only the top half of the last line of text appears on the paper, or characters are chopped off at the right ends of lines. If you're interested in the gory details, the OOP version can be found in Book 6, Chapter 3 of my book *Visual Basic .NET All-In-One Desk Reference For Dummies* (Wiley).

True, all these extra lines of code do give you more flexibility than the simple pre-OOP version. But that flexibility carries a terrible price. And this "advantage" is counterfeit. Non-OOP languages permit you the same flexibility if you want it: You can employ code libraries (API's) to manipulate any printer features you need to control. The difference is, with OOP you *must* write those 80 extra lines of code for every printing job.

OOP Makes Many Simple Tasks a Struggle

Even highly experienced OOP programmers continually struggle with even simple tasks. You'd think that years of experience writing in OOP or C-languages would result in greater facility and productivity. Unfortunately, for all but the brilliantly talented few--where OOP and C seem harmonic with their brain patterns--experience seems to have relatively little impact on overall productivity. The main reason for this is that

Has OOP Failed?
By Richard Mansfield
A Four J's White Paper

Page 8 of 13
September 2005

nobody is in charge of OOP or the C languages, so there is little consistency and very few reliable syntactic rules you can learn.

Humanity had a unique opportunity to design a sensible language that would permit efficient communication between man and machine. Other than Esperanto, this was the first time in human history that a language could be thoughtfully designed from the ground up rather than merely accreting like all other human languages. We failed. Various committees of academic theorists along with programming gurus in companies competing to control market share have contributed to the current muddle.

Inconsistent Grammar

To see the inconsistency in OOP grammar, consider an elementary programming task like changing the size of some text. To change the font size of text to 11 points, classic (non-OOP) code is straightforward and quite similar to the English language:

```
TextBox1.FontSize = 11
```

Compare that to the new "improved" OOP version:

```
Dim fnt As New System.Text.Font("Arial", 11)
TextBox1.Font = fnt
```

Which version seems more efficient, easier to remember and use, more readable, and ultimately more sensible? OOP claims to simplify and to improve efficiency. It rarely does.

One major problem with OOP libraries is that you have to learn new, unique taxonomic "addresses," interrelationships, and coding approaches for *each* programming task. There's little regularity, so there are few rules you can learn and apply across tasks. It's as if a card catalog had exploded in the library, and you had to look through the pile of cards, only by chance finding the address of a book quickly.

OOP libraries (and the resulting way that you invoke or "address" the functions contained therein) *are* organized, but the organization is inconsistent, haphazard, often illogical. For example, in Visual Basic .NET (the OOP version of Basic), one committee of Microsoft programmers decided that to change the *size* of text you must use the syntax in the previous example.

But another committee--for no discernable reason at all--specified that programmers must use *an entirely different syntax* to change the *color* of text. Here's how you must change the color property of text:

```
Dim c As Color
c = System.Drawing.Color.FromName("blue")
TextBox1.ForeColor = c
```

Mind you, you're doing *exactly* the same thing in both situations, namely changing a property of text. But you change these properties using vastly different source code.

Has OOP Failed?
By Richard Mansfield
A Four J's White Paper

Page 10 of 13
September 2005

With OOP taxonomies, there simply are very few conventions. The syntax for each task must be individually memorized. This is awfully inefficient, and programmers often spend more time trying to figure out the correct OOP syntax than they do actually writing the programming code itself.

To change the size of text:

1. Use the "New" command.
2. You must provide an argument list.
3. You don't provide a namespace.
4. Change the object's property.

To change the color of text:

1. *Don't* use the "New" command.
2. *Don't* provide an argument list.
3. You *must* provide a namespace.
4. Change the object's *method*, not a property.

There's a marvelous lack of common sense at work here. It represents a serious corruption of other, far more sensible, approaches to programming.

OOP's Features Don't Deliver What They Promise

OOP programming also requires that quite a bit of time and effort be spent wrestling with scoping rules and other essentially clerical issues. For example, among the OOP scoping commands are: Protected, Friend, and Shared. You even find combination scoping, using two scope declarations at the same time, such as Protected Friend, ReadOnly Public, and WriteOnly Friend. This kind of inflation, and the resulting ambiguity, should be a clue that theory is triumphing over practicality.

Encapsulation--sealing off successful code from further modification--is a valuable component of OOP. (OOP's polymorphism and inheritance features are often so damaging, bug-inducing, and unnecessarily complex that it's generally sensible to avoid them whenever possible.) However, creating "black-box" code is hardly exclusive to OOP. Most computer languages permit you to compile code libraries that contain unmodifiable, finished procedures. Where OOP differs is that it encourages

Has OOP Failed?
By Richard Mansfield
A Four J's White Paper

Page 11 of 13
September 2005

encapsulation not just in libraries where code is being reused, but also promotes hiding source code within a currently active programming project. In other words--don't let one programmer see another programmer's source code.

Hiding code can of course be useful, particularly with large, complicated programs where you want to ensure that nobody tampers with other people's tested, finished procedures. However, OOP doesn't limit itself to this useful clerical task. It's a much larger, more ambitious system. Its most astonishingly messy feature is often claimed as its greatest strength: the attempt to automate the process of code reuse.

OOP encourages writing code that behaves differently in different contexts (polymorphism). Instead of creating two procedures, one named ChocolateShake and one named VanillaShake, you instead create one procedure named Shake that can be either chocolate or vanilla, depending on the data you feed to the procedure. While this sounds promising, in practice it's confusing. Code can become harder to read because you've removed the descriptive parts of the name--chocolate and vanilla-- from the procedure's name. You've created a more general-purpose procedure out of two, more specific, more understandable, procedures.

OOP also encourages code reuse via inheritance: expecting the programmer to modify invisible (encapsulated) code. The contribution of inheritance to code unreadability, slow program execution, and bugs in general has--even among some OOP apologists--been widely acknowledged.

Alternatives

Nobody should be ashamed to admit that they have problems with OOP or the C language and its derivatives. *Everyone* has problems with them. Even OOP experts bicker among themselves about the "proper" way to manage the various convoluted aspects of OOP theory.

Some OOP theorists claim that the only alternative to OOP is "spaghetti code," but this is a straw man argument. Long ago structured programming ended the abuse of the GOTO command. Nobody seriously suggests going back to the early days before subroutines were in common use.

The true alternative to OOP is procedure-based programming and 4GL (fourth-generation) languages that are deliberately constructed to resemble natural human language as much as possible.

In sum: like countless other intellectual fads over the years ("relevance," communism, "modernism," and so on--history is littered with them) OOP will be with us until eventually reality asserts itself. But considering how OOP currently pervades both universities and workplaces, OOP may well prove to be a durable delusion. Entire generations of indoctrinated programmers continue to march out of the academy, committed to OOP and nothing but OOP for the rest of their lives.

About the Author

Richard Mansfield has been a prominent figure in the computer field for over two decades. From 1981 through 1987, he was editor of Compute! Magazine. He has written hundreds of magazine articles and two columns, and he began writing books full-time in 1991. He's written 38 computer books altogether and several became bestsellers, including *Machine Language for Beginners* (Compute! Books), *The Visual Guide to Visual Basic* (Ventana), and *The Visual Basic Power Toolkit* (Ventana, with Evangelos Petroutsos).

His recent titles include *The Savvy Guide to Digital Music* (Sams), *CSS Web Design for Dummies* (Wiley), *Office 2003 Application Development All-in-One Desk Reference For Dummies* (Wiley), *Visual Basic .NET All-In-One Desk Reference for Dummies* (Wiley), *XML All-In-One Desk Reference for Dummies* (Wiley, with Richard Wagner), *Visual Basic .NET Database Programming for Dummies*, *Visual Basic 6 Database Programming for Dummies* (Hungry Minds), *and The Wi-Fi Experience: Everyone's Guide to 802.11b Wireless Networking* (Que).

Overall, his books have sold more than 500,000 copies worldwide, and have been translated into 12 languages.